[SQUEAKING]

[RUSTLING]

[CLICKING]

PROFESSOR: And I just-- since I think it's been over a week, let me just kind of remind you the toy problem that I put up about a week ago. And the problem was where I was going to input x and y. Maybe you all remember this.

And I had a kind of a program, if you will. I kind of think of this as sort of steps in some software if you like, where I take a was sine x, and b is-- nothing special here, just illustrating a number of things. c is b plus x. And I was going to return z as the answer, OK?

And maybe you all remember that there was this graph that I drew that kind of illustrates a little bit of what's going on. I put the inputs-- in fact, I put all the variables in circles. And so there a. There was b. And I'm going to end up with z.

And I just put the local derivatives. And, Steven, I want to ask you if you would like or hate this notation, a notation that I think I've seen some people use. But, again, nobody's consistent in this field right now.

I kind of like the idea that-- here, let's-- and I don't want to get into placeholder notation. But if I were to talk about-- if I were to talk about-- I want to make a distinction between dz dx, and partial z and partial x.

And to be clear, this one, I want to equal 1. And I want to think of it as a one-step derivative. I look at this formula alone, and I don't want to think of b as being dependent on x, which, in fact, it is. But I'm pretending b is not dependent on x.

And I'm just looking at what I would call the step 3 derivative or the one-step derivative as opposed to the actual derivative, the one that puts it all together. And we wrote it down last time. What did I say it was? z, if you put it all together, was sine x over y plus x.

And so this thing, the thing that we actually want to calculate, is cosine x over y plus 1. So my question for Steven was, what do you think of this notation? Do you like it? OK. Anybody want to object to the notation?

So I like the idea of partial being a partial-- it's not the whole set of steps. It's just like one step. It's partial truth, and then the derivative being the whole truth.

And so what I'm putting-- what I'm going to put here are all the partials. So I'm going to put down the cosine x, which is the derivative of a with respect to x. And the 1 over y goes here. And there's a 1 that goes here. But there's also a 1 coming from here.

And let's see. What else did we have? And we had a minus a over y squared over here.

And maybe it's also useful sometimes to write down what these things are equal to. I'll just stick this below the circle. I don't know if the printing is too small. But I'll just do it anyway-- cosine y, and b plus x was here.

All right, so this is where we were I think when I left off last time, just to show an example of how a program is a sequence of transformations of variables. And for a program to work, for a program to work, any time you're at a node, all of the inputs have to be available and known to you.

So you may never stop to think about this property of a computer program. But it's sort of obvious, right, that I can't execute b unless I know a and y. I can't execute z unless I know b and x.

And so the very nature of a computer program is that this thing is going to be a DAG-- that's the word. Maybe most of you know that word. It's a Directed Acyclic Graph, which, in simple English, means that wherever I am, all the inputs are moving left to right. They're moving from the beginning to the end.

That could actually be relaxed, and you could have an implicit method or something. And most of what I'm talking about will still work under such situations. But I think, for simplicity, let's stick to DAGs and things going left to right, OK? Yep?

AUDIENCE: Why does a depend on y?

PROFESSOR: Let's see. Oh, of course, it doesn't. Thank you. If you want the honest reason as to why a depends on y is I left my glasses home today.

Thank you. Better? Good. Good catch. All righty. That's why a depends on y.

All righty. Good. So that's where we were. And let's see. How far did we get in all this?

So I think we got as far as saying the bit of the general situation for the forward derivatives. Let's see. Yeah, how far did we get?

So the goal, as you remember, is for every input-- so the inputs are over here. And for every output, so in this example, we have two inputs and one output.

Somehow calculating the derivative can be thought of as basically multiplying the path weights. If these were matrices, you have to do it in the right order. If it's scalars, you get to ignore that a little bit.

But we want to multiply the path weights and add that up over all the paths. So whether it's forward or backwards or any other way, our goal is to look at all the paths between x and z and then multiply the weights. Here there's just one weight. And then add it up over all the paths.

And so it's really the sum over paths from input i to output j of the product over the path of the weights. So I'll just kind of say it like that. I hope it's clear what is meant.

That's really what's going on, right? And some people would tell you it's just the chain rule, and they'd be right. I'm not a big fan of just the chain rule because-- not that there's anything wrong about it. But I feel like you don't really see what's going on if you just say just the chain rule. But if you look at it on the graph, the computational graph, I think it's much easier to see what's going on.

And it really doesn't matter-- if you really have the graph in front of you-- that's sort of the human thing to do-- it really doesn't matter if you multiply this times this times this or if you did it in the other order. It just doesn't matter. The product of the weights can happen in any order, and you can use associativity to your benefit any way you like.

And just so it's completely clear, I drew this symbolically. But on any computer, when you're doing automatic differentiation, everything here is going to be, most of the time, a number-- I would say 99.9% of the time. All of the variable, everything that you would see would be a numerical value.

So I just don't want any confusion that-- I did it symbolically because if I put a whole lot of numbers here, I don't think I'd make the point as well. But on a computer, when you do automatic differentiation, it's all numerical.

It's not finite differences. The way you get to the edge weights are never finite differences. But it is all numbers, usually floating point numbers of double precision or something, or maybe not, but floating point numbers.

OK. So then we talked about how this thing really works. And which of these do I want to tell you? So-- doo-doo-doo.

STEVEN JOHNSON: You did forward last time.

PROFESSOR: Yeah, I did forward last time. And maybe I could some up forward. I didn't say it last time. But maybe it's worth saying that when you do forward, you might have a node coming at your next step.

So you're in the middle of-- you're in the middle of doing forward mode. And I'll draw a little picture like this. Here I only had a case where a point depended on two inputs. But I'll just draw a picture kind of more generally. I'll write it, like, five inputs. But this could also be dot dot dot of some sort.

And I don't know what these variable names are. Maybe I'll just call them something. I don't want to call them x's or y's or z's. Maybe I'll call them a's. Let's do that.

So I'm at a step, and I've got a bunch of a's. And I've got a b over here where b depends on a. And I'm really just using the fact that here I'm going to put all these derivatives over here. I'm going to put db da1.

This is what I'm going to put on the labels-- db da2. This is just the one-step derivatives over here-- db da4, and this is 5, and so forth. I'm just going to put these.

And I've got the fact-- let's say I'm starting at a particular x. Let's just call it x1. And I've worked my way over.

And so at this point in time, I know a1, and I know da1 dx with a-- not the partial. But I know all of these values here, you see. I know da1 up to da5 dx.

So what I know is the paths that take me from this input to here. That's the information I have right now. And what I'd like to do is figure out how to move forward on the graph.

And so I'm just going to say this symbolically now. I want to be able to calculate-- my computer program will give me b. But the question is, how do I get db dx from the information I have right now?

And exactly what's happening is I'm going to sum-- I'm going to be summing-- let's see, what is it?-- db dai times dai dx from i equals 1 to whatever, right?

And that is the multivariate chain rule. This is really what's happening, that this is equal to this. And that's kind of why it all works, that you can proceed forward and keep working your way forward to get the derivative all the way to the end.

When you do reverse mode, we're going to have arrows going the other way. And so we'll see how it all happens. But the arrows are going to be-- here, it's, what do we depend on?

But what it's going to turn out to be the other way is, what are we influenced by, or what do we influence later? But we'll see reverse mode. But this is all forward so far.

And an interesting fact about forward mode is-- a detail about forward mode is you don't need to create the graph as a data structure. Yes, there's a question?

AUDIENCE: Just compute the partials [INAUDIBLE]?

PROFESSOR: Oh, well, that's-- so, OK, maybe we should talk about that. Yeah, let's talk about computing these partials. So the first answer I'm going to give you is the most simple basic answer, the "no cheating" answer. And then, in the real world, I'll tell you what the cheating answer is.

So you could write every computer program in the world. So here, let's put this over here. I'm going to need this for reverse mode as well.

So every computer program in the world basically it can come down to plus, minus, times, and divide. I could probably even remove some of these. But every computer program in the world can be brought down to plus, minus, times, and divide.

When you do a sine function or an exponential function, there's some power series or maybe a ratio of power series, and it's not the one you learned in calculus. So it's not Taylor series. I don't care what your calculus teacher told you. It's never done by that way.

But nonetheless, there's some sort of power series or ratio, a power series, usually, somewhere in there. And in the end, everything you ever do on a computer can come down to this. There's also conditionals and other things. But as far as all the arithmetic operations, it can be reduced to plus, minus, times, and divide.

So in fact, all you need are the partials for these. So, for example, all you really need, theoretically, is the ability to do-- right? So if b is a1 plus a2, the only two partials you need are this. In fact, let's put a plus or minus, and I'll cover two of these cases right there.

And what else is there? There's times. So if b is equal to a1 a2, then all I need is this. So I just need to know that, right?

And this is, by the way, what I did in that little notebook where I kind of did the magic of forward mode differentiation. And then, finally, there's the division one. And the partials are-- I mean, this is maybe a slightly different point of view of the quotient rule and the product rule.

But if you actually step back and look at it, this is exactly the sum and difference rule, the product rule, and I'm about to write the quotient rule down. Let's see.

So the partial with respect to a1 is 1 over a2. And the partial with respect to a2 is minus a1 over a2 squared. And I hope you can see that the information that you see here is exactly the same information as the product rule. It's a2 da1 plus a1 da2.

It's 1 over a2. This is the denominator times the derivative of the numerator or whatever. I guess you might want to write this as a2 over a2 squared just to see that it's the same thing. But it's the denominator times the derivative of the numerator minus the numerator times derivative of-- you see, so it's all the information you've known in the old days of calculus. I'm just writing it in a slightly different way.

If you have these rules, and your computer program only has those things, that's all you need, right? If that's what your program breaks up to atomically, you can actually do forward mode differentiation completely.

Now, of course, that's not how it really works in the real world. But I want everybody to see that in theory, this is all you need. So in practice, we know lots of derivatives. We know the derivative of sine is cosine. We know the derivative of log x is 1 over x.

So we throw them in so you don't have to waste your time. And Julia-- I think Steven mentioned in the Piazza or somewhere about chainrules.jl. So lots and lots of derivatives that are known get stuck in so that you can do this at a higher level.

So instead of letting the computer derive the derivative of sine x with all those pluses and minuses, it could just know the derivative of sine is cosine.

STEVEN JOHNSON: And as you also find in your problem set--

PROFESSOR: Something was missing, right?

STEVEN JOHNSON: And sometimes it fails because it calls out-- so if forwarddiff relies on this dual number type, it works as long as you have Julia code that works with any number type. If you call out to a FORTRAN library that only works for double precision floating point numbers, then it can't differentiate that. So then someone has to write a special rule. And if they didn't, then it fails and probably complains that you if you try to use forwarddiff on [? the matrix ?] [INAUDIBLE] focusing.

PROFESSOR: So that's the problem with calling-- if only we can get everybody to consistently write all their code in Julia, you would never have that problem.

AUDIENCE: I mean, one solution is if you're Google, and you just hire a small army of engineers to just re-implement all the math libraries you ever need in Jax, which is basically [INAUDIBLE] implement NumPy, [INAUDIBLE], SciPy, and so forth. And then, as long as you stay in that closed universe of things that people have written derivative rules for, you're fine.

PROFESSOR: So I'll tell a little story. Before I really grew to understand-- automatic differentiation has only gotten exciting in the last couple of years. And as I was just starting to poke my head into it, I differentiated the eigenvalue problem. Besides the FORTRAN libraries, the LAPACK that-- it's probably C now anyway.

But besides the LAPACK library, there's another implementation of the symmetric eigenvalue problem in genericlinearalgebra.jl. It's written in all Julia. And it actually works beyond-- it works for quaternions and weird number systems and finite fields and not just plain old floating point numbers.

And so you can differentiate through it because it's all Julia, and it got the right answer. And I was talking to a friend of mine. What was he what was the package he was excited about with its automatic differentiation?

Oh, it's a statistics thing. Oh, what was it? Oh, I'm just blanking on the name. But it's a very well-known one. Anyway, it doesn't matter.

I was talking to a friend of mine, and I said, can you differentiate the symmetric eigenvalue problem? And he said to me, oh, well, we can, but we have to teach the system. And I thought that was crazy. Having to teach the system-- we really can't right now. But if we threw in a new rule, then we can.

But I now realize that everybody does that, right? It's actually reasonable from a kind of a complexity point of view. So you don't have to actually drill down from every operation.

But I was impressed that Julia was able to just get the answer of the derivative of the symmetric eigenvalue problem without being taught how to do it. It actually just did it using just these rules. And I thought that was kind of cool when I first saw that.

OK, so this is really all that you need. But as I say, in practice, in practice, we've added lots of things. For example, if b is equal to the sine of a, then this edge with cosine a is just-- it comes on the edges.

OK, so-- what are you doing there? It's--

AUDIENCE: Just-- we're all right. It's too small to have the [INAUDIBLE].

PROFESSOR: Oh, yeah, that's probably true. OK, maybe the last thing I'll say before I take a break, and-- the break may be slightly longer because I'm going to put a sweater on this little guy. And that'll take me a couple of more minutes. But he's really cute in that sweater. So you'll all like that.

So the other thing I wanted to say is when you do forward mode, you might ask yourself, where is this graph on the computer? I mean, you know, of course, on the blackboard, you can visually look at this graph, and you can see the paths. So where is this graph on the computer?

Well, in some sense, when you do forward mode, one way to say it is you don't need the graph at all. Maybe a better way to say it is the graph is kind of being built up as software executes, in effect.

As you execute, as you execute sine x, there's nothing stopping you from creating that cosine x and then getting that second part of the dual number as you go. And so you see as you execute, we can start out with x1. You get to over here.

We've got a, which is sine x and then cosine x. And you just can keep moving along. So in some sense, I think this is worth thinking about because-- I don't know if it's completely obvious or not. But as the computer executes, as it actually calculates this from this, it can also keep that path, the product of the weights going forward, just by multiplying-- at this moment in time, you can get this. And just as you need it, you can multiply it by what you have here.

And so that's why overloading works when you do forward mode. You take your original computer program. You let those dual numbers propagate forward. And you are literally, as you get there, you're getting the right value for-- so you got this-- you have all the values for the sum over the paths-- of the product of the paths to this point.

And so you don't need to go crazy to actually build this as some weird data structure. You just go forward, and it's there. However, no such luck with reverse mode.

So after the break, after I put on his little sweater and come back, what I'm going to do is discuss reverse mode in some real detail. And I find it very interesting to see in what sense reverse mode is the symmetric version of forward mode. It's just looking at it in the mirror. And what is it about computer software which kind of makes it asymmetric?

From a pure math point of view, reverse mode and forward mode are just mirror images of each other. But because of the realities of how computers execute, then they are no longer exactly mirror images. And reverse mode seems a little bit harder to actually implement.

It's not hard, but it feels a little bit-- it just feels a little bit less intuitive because-- it's ultimately because computer programs execute forward. What else are they going to do? If you could only run computer programs backwards, then you wouldn't have to-- it doesn't mean anything but. But I just want to make that point.

All right. So we take a break. It might be slightly more than five minutes. But I'm going to take this young man outside.

So I think the next thing to emphasize as I go from forward to reverse is to really focus on, perhaps, this basic picture where we talk about this variable I'm calling b. And it's all of the things that I depend on.

And maybe I should actually think a little bit. I'm going to get confused in a second. Did I actually put this in the order I wanted or not?

So let's see. If these were matrices, I want them to-- I would want them to be multiplied. Oh, god, I'm going to get completely confused.

You know what? We're not doing matrices. I'll think about it offline and let you-- unless you know the answer off the top, but I wanted it this way or this way. But we're doing scalars today. So I won't have to think. All right, but the key point is--

AUDIENCE: Yeah, you [INAUDIBLE].

PROFESSOR: Oh, I got lucky. OK. All right. So here it's every node that b depends on.

So when you're going backwards, what you need is everybody that you influence. So to draw that as a picture, I have a node a. And it's going to influence a whole bunch of nodes. Let's call these b1, b2. I'll just stick with my-- I'll stick with my running 5. But, no, there's nothing special about 5 here.

So in your graph, the graph is still going left to right. But you have a node, and somewhere along the line, you can say all of the nodes that a influences-- for example, influences in one step, right?

So x is influencing in one step here, and x is influencing z in one step. And so if that node over there was x, I would have two output arrows.

This is every node-- so every node, I'll say, influenced by a or, maybe more simply, every node that depends on a. So that's what one has to look at there.

And what we're going to do is we're going to have a final output variable z, where everything is going to start. And it wouldn't surprise you all that we're going to start-- we're still going to have something that looks like an ordered pair of numbers. But we're going to work our way this way.

So we're going to do all the multiplications from right to left. And at the end, over here, we're going to have-- let's say I had one x over here. When we're finally finished, we're going to compute x and dz dx.

So that's our goal is to start at this end and compute the derivative-- in other words, compute the exact same thing. But just to remind you, what we did here was we started with x, comma, 1 this way. And at the very end, we ended up with z and dz dx over here, OK? So that's your first kind of peek at how reverse mode will actually work.

And the way it's going to work is here we're going to presume that we have the-- we're going to have the b's and the-- we're going to have the full derivatives-- so dz db2. So in the middle of the algorithm, we're assuming that we have all of these things.

Sorry, these are not the partials. We have the derivatives all the way from z to where we are now. And our goal is to figure out what to put here.

And so I hope you could see that if you can always go-- if you just know what to put here, then you can actually go from right to left. And so our goal is to figure out dz da. So if we can only have a good formula for dz da in terms of all of these pieces, we would have no trouble being able to do this.

And, of course, the way this is going to work is-- what we're going to do is we're going to sum up now-- let's see. How is this going to go? Is it going to be dz dbi, dbi da-- from i equals 1 to 5 or something.

AUDIENCE: Do you need a transpose [INAUDIBLE]?

PROFESSOR: No, I'm not in scalars. But, yes, I mean, ultimately, there'll be some sort of transposer. Or I have to write it in the right direction as an alt--

AUDIENCE: That's from the outputs on the left, right? [INAUDIBLE].

PROFESSOR: Yeah. But I'm going to skip that for now, just to not complicate it. But Steven is just right.

And you see that this is also a chain rule. But also, I want you to really see that from a graph point of view, all that's happening is we're making sure that what we put here is the path all the way to the right multiplied by what's on this edge. And then just add them all up.

So you could see it the calculus way as these are two versions of the multivariate chain rule. I don't prefer that view, by the way. But some people do. And it's not wrong.

Or you could see it as we're just multiplying the sums of all these weights, and you notice, huh, by the time I get here, I'm actually calculating the same thing. I'm just getting the sum over all the paths of all the weights. And, again, you get to look at it both ways, whatever you like. But both are absolutely correct.

And so you have no problem. So you just keep moving your way through the graph, and you form these sums. So any questions about that?

So the next thing I want to talk about is a bit of a detail as to, how do you actually, in practice, compute these sums?

You see a sum like this, and perhaps your temptation is to form this entire sum at once. But you don't have to. You can traverse these arrows one at a time.

So, for example, you start out with-- so in practice-- let me put this over here. In practice, at some point, you'll initialize everything with a, 0. And you traverse this arrow, and maybe you'll get a, comma, oh I don't know, db3 da times dz db3.

You'll get one of your sum ends, and there's four more to go. And then, at some later time, maybe what you'll do is fold in-- there's no rule that says you have to form this entire sum at once, right? So maybe you'll add the first one later-- so db1 da dz db1.

And this is close to what's happening in practice, that you traverse an arrow, and then you just add the answer to whatever is here. Once you're finally done, you'll compute this entire sum. But you don't have to do it at once.

And so I think the secret to every good algorithm I've ever seen is to recognize that you don't have to implement something the way the formula looks. You see a sum, you can do it in any order. You can do it at any time.

So what actually happens in practice is one way or another, if you're going to do reverse mode, you have to have-- the first step is you have to go forward to-- you're going to go forward to actually compute all the nodes. Even if you did this one backwards, you're going to have to have this forward step where you compute a, b, and z.

And during those forward steps, you actually build up this graph. There's no way around it. You build up the graph, and you put the values on the edges.

And so unlike forward mode, which is-- so let me say this in a better way. So in terms of implementation, for forward mode-- so you don't build the graph.

It's kind of implicit. In reverse mode, there's an explicit graph that has to be built up. So in reverse mode, you need to calculate one or another-- you actually have to build the graph. So you need some sort of data structure to be able to build up this graph.

Oh, I just remembered, I have a nice demo of all this. I wonder if I'll find it fast enough. I forgot I had a good demo. You've never seen it because it was since last year. I'll try to find it.

But we have an explicit graph. And so what happens is you'll have a forward pass where you actually compute the primals, or the variables, and you build the graph. And then, once you have the graph, you can do a reverse pass, where you start evaluating the derivatives, where you evaluate the path products.

And, again, this could all be done with just-- this, too, could also just be done with plus, minus, times, and divide. Or you could actually add pieces to it. So in a way, this is like a transpose. But I'll kind of show you how it would work.

So with plus, minus, times, and divide-- where can I put that? Maybe I have enough room here. And so-- let's see. I just want to get this straight. Let's see, doo-doo-doo.

So, yeah, so how should I do this? So if I had, at this point-- let's say I had b, comma-- what's a good notation? It doesn't matter. I'm going to call this p.

So the gradient of plus is just 1 1. The gradient-- or plus minus is plus or minus 1. The gradient of product is, like, a2, a1. And the divide is what's over here.

So in a way, what's going on is I need the transpose. This becomes 1 plus minus 1. This becomes a2, a1, if I take the gradient transpose.

And so basically, what happens is if I have a plus, this just becomes a, p. Well, the way to write it is a and-- I'm getting caught up in a little bit of notation. Let me say it in words.

Suppose you're moving along, and a plus operation happens. So maybe I already have some partial results. So here I have pb at the moment, and I have a, and I have a pa1 and an a2 and a pa2.

And here I am on the graph. This a1 comes from other things, not this edge. I want to process this edge now. These two edges have not been processed, and I'm going to process it.

So what's the operation? The operation is going to be pa1 plus equals pb. And then the other operation will be pb2 plus equals pb. And that's how you do an add backwards. You just move this along.

If you were doing a multiply, which might be a little bit more informative-- let's do the multiply. So here we have a, pa, a1, pa1, a2, pa2, we're going to multiply these two. And let's call this b and pb.

So what do you put on these edges? The edges are just the same derivatives. Nothing changes there. We're multiplying a1 and a2. So this is going to be a2 and a1.

And now what's the implementation? The implementation is just pa1 plus equals a2 times pb and pa2 plus equals a1 times pb.

So that's basically how you would do it. And division would be the same. You want to see division? It's not much different. You would just use these things.

And so I would multiply-- I would add the product with a2 over a1. Well, I would just call this 1 over a2.

But-- do I have time to find the demo or not? I should have thought of this. I've got a really cool demo, but it might take me a minute to find it. It's really good, though.

Let me see if I can find it quickly. If I could find it quickly-- while I'm looking, does anybody have any questions about reverse mode? It really is-- let me see if I know where this thing is.

STEVEN JOHNSON: The one thing to remind you of is the reason for this. So the forward mode basically-- suppose you have some function. So there's some cost of that function. Forward mode, the cost of that is proportional to the cost of the function times the number of inputs.

Reverse mode is proportional to the cost of the function times the number of outputs. So if you have lots and lots of inputs and few outputs, you want to use reverse mode. If you have lots and lots of outputs and a few inputs, you want to use forward mode.

If you have lots of both, then it sucks to be you. It's going to be expensive either way, right?

And basically, what happens is if you have lots of inputs and a few outputs, then forward mode ends up recomputing a lot of terms as it explores all the paths, whereas in reverse mode, you share computations as long as you can until things fan out. That's the intuitive thing.

The other perspective-- the explicit perspective is if you write it as a product of Jacobian matrices, then basically you want to arrange it so if you only have one output, then you have a row vector on the left. And you just want to do vector-- row vector matrix products are cheap. Matrix-matrix products are expensive. We did that perspective before.

But that perspective is a little harder to write down once you start getting general graphs. But it's the same basic idea, that you want to arrange it so that you're multiplying matrices times vector-- Jacobians times vectors and not Jacobians times other matrix Jacobians.

PROFESSOR: I think I found it. And there's also something else I can show you related to-- well, maybe not.

So regarding the Jacobians, I think it's instructive and maybe not obvious. How do you turn this picture into that Jacobian point of view? Or even if these are all vector-valued or matrix-valued functions, it gets a little bit confusing because what you have to do is actually-- you have to put all the variables together to be able to write down. So you get a very sparse Jacobian.

I mean, maybe this could have been a homework exercise. Maybe we should remember this for next year. But turn this, or something like this, into a matrix multiply of Jacobians. And see what that looks like.

I think it's not so obvious the first time you see that. In fact, I remember being a little bit confused because I was trying to compare this viewpoint with the Jacobian viewpoint. And in the end, the mathematics is the same. But it's not really obvious at first.

But regarding the number of outputs versus the number of inputs, I think it is sort of interesting. Let me just draw one quick example that I think is very illustrative with the graph point of view.

So suppose I could do this just by pathway. Maybe this is enough right here. Maybe this is enough.

So let me just put the-- I don't even need to put the nodes down. Let me just put-- I'm just going to call these things a, b, c, and d. And your goal in life, your only goal in life-- this could be done all-- no calculus need to be mentioned, purely graph theory question.

Your goal is to calculate-- these are your sources. I won't even say "inputs" anymore. I'll just say "sources." This is a source, and this is a sinc. And I just want all of the products of the graph rates from source to sinc.

And in other words, the answer I want you to give me is acd and bcd. Your only goal in life is to give me that. So the obvious first way, the naive way, is, all right, I'll start at this source, and I'll compute a, and then I'll multiply it by c, And then I'll multiply it by d.

Or I could start over here right, and I'll start with b, and I'll multiply by c, and I'll multiply by d. And if you count, I did four multiplies. But if you want to do it the reverse way-- if you want to do it the reverse way, I could do d, and I could multiply by c. And I could take this result, and I can go in two ways on it.

Now, I can multiply this by a-- so a times this, a times this result. Or I could go b-- I could reuse. And now, all of a sudden, what do I have? I have three products instead of four.

So this is not a bad way to see why reverse mode could possibly be better than forward mode without thinking about matrices and whatnot.

This is a little bit like-- for those of you who know about dynamic programming, where people talk about reuse of precomputed quantities, this is what's going on here. When you start moving from right to left, if you have a lot of sources and maybe only one sinc, which is typical in machine learning, there's a lot of quantities that you can reuse.

So it's kind of nuts to go a times c times d and b times c times d when you can just store the result of c times d and then just do the two separate multiplies. And, maybe, I think this gives you-- to me, this gives the best view as to why reverse mode could be very, very valuable compared-- without derivatives, without matrices, just this simple example, I think, kind of does the trick very nicely. It's saying the same thing that Steven was saying.

All right. I found this thing. Now let's just see if it works. Yeah, if you wouldn't mind.

I haven't looked at this in a while. But it was kind of cool when it worked. So sorry I haven't checked. I should have checked before.

But here, let's see. Do we have the two? Did this thing come out?

OK. So this is really cute. So let's see. I think the computation that's being seen here, I think-- is this up right here? 2x-- yeah, 2x is going to get-- 2 and x are going to get multiplied, the black numbers, and then times y.

And then we're going to have a plus. And here we have the minus of x and 1 squared. So the actual computation that you see here is exactly what's in these parentheses.

And now let's see what. Are we going to do are we doing forward mode or backwards mode? Let's take a quick look. So let me just see how this works. Sorry, I should have looked at this, but I forgot I had it.

Oh. So I think in red, we're just going to assume the variables are x equals 3-- so first we're just going to do the expression, without any derivatives. So we're going to take x to be 3 for whatever reason and y to be 5.

And then we have these constants. So 2 is 2, and 1 is 1. So we're just going to take x to be 3 and y to be 5. Oh, if x equals 3 and y equals 5, then, OK, the derivatives are-- the derivatives are 14 and 6. So that's where we want to go.

All right, so let's take the forward pass. So 2 times 3 is 6. 6 times 5 is 30. Let's see. I think we're going to do this minus next, I would imagine.

So 3 minus 1 is 2. There it is. 2 squared is 4. And let's add 30 plus 4 is 4.

OK, so now what's going to happen next? Hopefully, we're going to start-- OK, we're going to start the reverse pass.

So the answer to this computation is 34. That's the z that I've been talking about, all right? And now I want you to just see the reverse pass.

And so we always start out as 1. I think that's what's behind over here. The funny thing is you can get good at doing this by hand. I mean, it's kind of actually not a bad-- I'm not sure I'm completely there yet, but I have done this in classes where I would take a little computation and do reverse mode. But it's also easy to get a little confused. So it's very dangerous.

OK, so let's see. So we have a plus, OK? So when you have a plus, then-- these derivatives are all starting out at 0. So what's going to happen-- what's going to happen for this plus?

What's going to be the second number here and here? I haven't done it yet. But if you're following how reverse mode works, what will be the derivative here?

So what should be the numbers in green as soon as I press the button here and here? Do you see what that is?

STEVEN JOHNSON: I heard somebody whisper 1.

PROFESSOR: Yeah, both of them are 1. And one way to say it-- if you don't remember what's going on here, you can remember that the partial derivative of the sum with respect to this number is always 1 here. There's like a little 1-- just to copy the way it was on the blackboard, there's a 1 sitting on this edge, and there's a 1 sitting on the edge. It's the partial derivative of the sum with respect to each of the two inputs. So this should just be 1 each, and there it is.

OK, now let's go a little fancier. Let's do this product. If you remember how the product works, what should I imagine is sitting on this edge? What number here is on this edge? And what's on this edge?

So the product rule for calculus-- it's the same old product rule. You just have to get used to it in this new circumstance. So what goes here? The 5, and here is the 6, right?

So what we're going to do is we're going to go 5 times the 1, add it to the 0, and we're going to get the 5. And now here we're going to go 6 times the 1, and add it to the 0, and we're going to get a 6.

And we have another multiply. So there's the 3 and the 2. So if I've got this right, we should get the 15 and the 10. Yeah, the 15 and 10 should show up, and it does.

And notice the 10, by the way, showed up here because it's the same variable. And so there's really-- in storage, there's just one x. So now what do we-- we need to do the square.

So the square-- we've got the 1 here, right? And the derivative of square is 2x, right? And so we would actually have a 4 over here. And so we have to take 1 times the 4 and get a 4 if I did this right. Yep.

And now we have a minus. So we have a 1 and a minus 1. The 4 times the 1 adds to the 10 and gives a 14. Yep.

Oh, I like that, 10 plus 4. And then this is 0 minus 4. And there we go. And then we actually have-- the derivative with respect to x of this quantity z, apparently, is 14 and 6. So we've got the 14 over here because we just did it with respect to the x. Oh, the y is there, too, the 6. Yeah, the y is there, too.

OK, so this is-- I mean, if you guys want this, I could give you-- we can link this. But I sort of go through here-- we're out of time now, but we actually talk about-- you can see the whole code from beginning to end. So it doesn't-- I don't know. Reverse mode seems more mysterious, I think, than forward mode.

But we actually have the whole thing self-contained in one notebook here where you could actually see the building up of the tree and actually how it works. And so we will-- in fact, you could actually see this tree coming up in a data structure. Yeah, so I'll link to this so you can see how it works.

But I think we're going to wrap it up now. Steven's going to make a list of things we haven't shown. Do you want me to-- do you want to start talking, and I could do that, Steven? I'll get the lights on.

STEVEN JOHNSON: So, yeah, so it'd be great if we had another couple of weeks in this course. It's actually amazing how much there is to say just about taking derivatives, even though you come in thinking you know how to take derivatives, and there's so much left.

PROFESSOR: The problem is linear algebra was evolved lately in math departments. People thought it was all easy.

STEVEN JOHNSON: Yeah.

PROFESSOR: And so they actually never pursued it, and they missed the boat. I really believe that.

STEVEN JOHNSON: So, for example, one very practical thing that I was hoping to cover, but we didn't get to, is how to do these custom AD rules. So suppose you have a box. This is your program, or maybe just a subroutine in your program. Or let me just say a function that takes in an x1, and then say this is a vector, and an x2, or let's say x and y-- x and y-- and outputs, say, an f and a g. And then suppose that AD fails.

PROFESSOR: As it did in the first version of the homework.

STEVEN JOHNSON: As it did in the first version. You called out to some [INAUDIBLE] or something that doesn't know how to differentiate. And there are, unfortunately, lots of things that these systems don't know how to differentiate. Or maybe it differentiates, but it differentiates it very, very badly. So that happens in a very inefficient-- that can happen if this function is computing something approximately.

And it spends a lot of effort trying to exactly differentiate the error in the approximations instead of realizing, oh, we're really doing an integral, and we're doing it accurate to 13 decimal places. I don't need to get the derivative of the last three digits.

And so what you do is if you're doing forward mode, what you have to do is supply a thing that computes, basically, a Jacobian vector product, a product, a JVP. It's also called an frule in chain rules. It's also called a pushforward, I think this thing--

PROFESSOR: Yeah, differential geometry, yeah.

STEVEN JOHNSON: So some things use this terminology. It comes from differential geometry, and it's called a pushforward. And so basically, you're supplying-- you just write a program that computes f, partial f, partial x times dx. And it computes a partial f, partial y dy, given dx dy.

So you implement this yourself by-- you know what your function is computing, and you work out manually with the derivative is. And if you're doing it in reverse mode, which is more common if you're doing any kind of optimization machine learning, you have to go in the other direction. You have to provide a vector-- and I should say a row Vector Jacobian Product, which is also called a VJP or an rrule in reverse rule, or a pullback in that funny differential geometry notation.

And so basically, equivalently, you're effectively providing a function that computes partial f, partial x, that Jacobian transposed, times a vector u and partial-- sorry, that should have been partial-- yeah, well same thing for-- and partial f, partial y, transposed, times u for an arbitrary u, and similarly for g.

PROFESSOR: And maybe just to say that what Steven's writing right now are exactly the rules in those boxes, except he's being careful about the fact that this is the multivariate version--

STEVEN JOHNSON: Yeah.

PROFESSOR: --when I was like, oh, I'm just going to do the scalars.

STEVEN JOHNSON: And so, in principle, this is straightforward. In practice, you need to think a little bit about the rules for what this even means. So, for example, these systems are designed-- written in such a way as to be quite general. And so because of that, the interfaces for supplying these can sometimes be a little bit confusing when you're not used to it.

Another thing that I would have liked to have time to cover is talk about complex numbers.

So, for example, suppose you have f of z where z is x plus iy is a complex number. Suppose you have that equals z squared, the magnitude squared. So this is the same thing as z times its complex conjugate.

So if you take complex analysis, they'll say, OK, this function, you can't differentiate at the end. It doesn't have a derivative with respect to z in the usual sense.

But, of course, if you change z by a little bit, the output does change by a little bit. And if you look, for example, in terms of x and y, the real and imaginary parts, this is a perfectly differentiable function of x and y.

And so suppose you're-- there are lots and lots of problems that are very naturally expressed in terms of complex numbers, especially in physics for example, physical quantities. You want to be complex, and you want to optimize some function of them, some real valued function.

And so, of course, if you split everything into its real and imaginary parts, it becomes differentiable in the usual way. And then you can take derivatives. But this is a very awkward way of-- if things are naturally expressed in terms of complex numbers.

PROFESSOR: At first it doesn't seem awkward right, but you only realize later that it's awkward.

STEVEN JOHNSON: Yeah.

PROFESSOR: I mean, it seems kind of natural at first.

STEVEN JOHNSON: It seems natural at first, but then it just turns into-- because then, basically, your code does not look like your math anymore. It doesn't look like your physical equations.

PROFESSOR: Right. Complex code, like z cubed, is much like-- other than all those--

STEVEN JOHNSON: It's like If you read Maxwell's original paper on Maxwell's equations in 1865, he didn't have vector notation. He just used a different letter of the alphabet for every component of every field. And you have, like, 20 variable equations and 20 variables. So this is a version of that. So now you really only have one quantity that's complex, but now you have to have two variables.

So if you just ask, what is df-- what is f of z plus dz minus f of z in terms of z-- this turns out to have a perfectly-- it's basically z times z bar plus z times d z bar.

And so in general, it turns out what you can do is you can write df as df dz times dz plus df d z bar times d z bar.

And you treat these-- the notation is a little confusing because it looks like you're treating z and z bar as independent variables when of course they're not. You should really think of this as a function of-- this is really a function of two variables. This is like f of a, b equals a times b where you just let a equals z and b equals z bar.

So you can think of it as a function of two variables, and at the end, you plug in the complex conjugates. And it's nice to-- so everything ends up being perfectly sensible. It's equivalent in some sense to differentiate in [? respect of ?] the real and imaginary parts.

PROFESSOR: The way I look at it-- I don't know if this helps or hurts. But in the complex plane, dx is eastward and dy is northward. And here you have dz and d z bar are these two symmetric directions. But it's still-- you need two dimensions to describe what's going on. So that's one way to look at it.

STEVEN JOHNSON: So then you can go through and work out, what does this mean for gradients? What does this mean for Newton methods? All of those kinds of things. And so the name for this is--

PROFESSOR: Does anyone in machine learning ever use complex numbers?

STEVEN JOHNSON: I don't know if it's very common in machine learning. Maybe if you're doing FFTs or something like that--

PROFESSOR: Oh, maybe.

STEVEN JOHNSON: --then it would be natural. But this is also called the Wirtinger calculus-- Wirting-- do you actually know what the C and the R stand for in CR calculus?

PROFESSOR: Not a clue.

STEVEN JOHNSON: So if you ever need to optimize a function of complex variables, it turns out there's a perfectly sensible way to do it that still works when you have complex conjugates and real parts and imaginary parts that you can't differentiate normally. So just look this up.

PROFESSOR: That's what I'm doing.

STEVEN JOHNSON: You do not have to drop into separate real and imaginary parts, which is always an option. But it can be a very awkward and unnatural option if that's not the language that your problem is naturally expressed in. So there's more things.

I know the first time we taught this course last January, Alan had ambitions of getting into exterior calculus and some differential geometry.

PROFESSOR: It's not so bad.

STEVEN JOHNSON: It's not so bad.

PROFESSOR: I could have done it in one half of one lecture. But I didn't.

STEVEN JOHNSON: Yeah, it's one of those things-- when we first started planning this, I was like, do we really have enough material to fill 16 hours of lectures? And then you realize, oh, every little thing-- there's all these topics and sensitivity analysis, and they're all more complicated than--

PROFESSOR: So I don't think it's any of these things, Calculus of Real Values, Calculus Readiness, which is an education term for students who are ready for calculus, Cost Reduced--

STEVEN JOHNSON: Yeah, I've even seen it written as blackboard bold. So I think this is the clue. CR, I've seen it written this way. So it's like the complex numbers and the real numbers. I think it might be just a reference to those two sets somehow. But you want to map complex derivatives into real derivatives or vice versa.

So, yeah, anyway, so this is-- there's a whole set of rules you can derive beyond this for CR calculus. If the function is real, for example, if the output is real, then these two terms have to be complex conjugates. And so this is twice the real part of that.

If it's analytic, then this time it's 0. So if it has an ordinary derivative in the 18.04 sense, in the complex analysis sense, then one of the terms drops out.

PROFESSOR: You don't think it's a chain rule of calculus? No, maybe not.

STEVEN JOHNSON: Oh, yeah.

PROFESSOR: Somebody threw that out there. But I don't think that's right.

STEVEN JOHNSON: But anyway, if you google it, you can find notes on these things. Alan, is there anything else you wanted to come-- what else did we leave out?

PROFESSOR: Oh, there's bazillions of things. But I think that's a pretty good list right there. We can call it a day with that, I think.

STEVEN JOHNSON: Yeah, yeah. There's so many more things. I mean, Alan did a little bit on derivatives of eigenvalues and so forth. But there's whole papers on how do you differentiate different matrix factorizations and--

PROFESSOR: I wrote chapters about this 20 years ago, but I never did anything with those chapters. And now, yeah, now everybody's discovering it.

STEVEN JOHNSON: And there's [INAUDIBLE] open research problems in how to efficiently take care of some of these things, especially if some of the terms are sparse, and how do you track that information through the process to take full advantage of the structure of the matrix.

PROFESSOR: But hopefully, you now all realize that calculus is way more interesting than your freshman teacher had you believe.

STEVEN JOHNSON: Just taking derivatives. You think, oh, they tell you that, oh, taking derivatives is trivial. You learn these five rules, and then you can handle everything.

PROFESSOR: And I think it's safe to say that most members of our department, they don't think about these things. They probably have never come across these things, by and large, I think. In other words, it's not something that's-- it's coming up more in computer science than mathematics, I believe. I think that's the truth.

So if you mention matrix calculus to many people in math, they-- it may not mean too much, I suppose. I'm not sure.

STEVEN JOHNSON: I mean, computer science, computational science--

PROFESSOR: It's coming up a lot more these days.

STEVEN JOHNSON: That's where you end up differentiating-- I mean, you say, I know the rules for the derivatives, and now let me differentiate the drag on an airplane wing that I solved by Navier-Stokes equation.

PROFESSOR: Well, give me that machine learning gradient to move along.

STEVEN JOHNSON: And then you realize it's not so easy to take derivatives.

PROFESSOR: Right. All right. I think that's it.

STEVEN JOHNSON: OK. All right.

PROFESSOR: So thanks, everybody.