PROFESSOR STRANG: [INAUDIBLE] Professor Suvrit Sra from EECS who taught 6.036 and the graduate version. And maybe some of you had him in one or other of those classes. So he graciously agreed to come today and to talk about Stochastic Gradient Descent, SGD. And it's terrific. Yeah, yeah.

So we're not quite at 1:05, but close. If everything is ready, then we're off. OK. Good.

PROFESSOR SRA: And your cutoff is like 1:55?

PROFESSOR STRANG: Yeah.

PROFESSOR SRA: OK.

PROFESSOR STRANG: But this is not a sharp cutoff.

PROFESSOR SRA: Why is there [INAUDIBLE] fluctuation?

PROFESSOR STRANG: There you go.

PROFESSOR SRA: Somebody changed their resolution it seems, but that's fine. It doesn't bother us. So I'm going to tell you about, let's say, one of the most ancient optimization methods, much simpler than, in fact, the more advanced methods you have already seen in class. And interestingly, this more ancient method remains "the" method for training large scale machine learning systems.

So there's a little bit of history around that. I'm not going to go too much into the history. But the bottom line, which probably Gil has also mentioned to you in class, that at least four large data science problems, in the end, stuff reduces to solving an optimization problem. And in current times these optimization problems are pretty large. So people actually started liking stuff like gradient descent, which was invented by Cauchy back in the day.

And this is how I'm writing the abstract problem. And what I want to see is-- OK, is it fitting on the page? This is my implementation in MATLAB of gradient descent, just to set the stage that this stuff really looks simple. You've already seen gradient descent. And today, essentially, in a nutshell, what really changes in this implementation of gradient descent is this part. That's it.

So you've seen gradient descent. I'm only going to change this one line. And the change of that one line, surprisingly, is driving all the deep learning tool boxes and all of large scale machine learning, et cetera. This is an oversimplification, but morally, that's it. So let's look at what's happening.

So I will become very concrete pretty soon. But abstractly, what I want you to look at is the kinds of optimization problems we are solving in machine learning. And I'll give you very concrete examples of these optimization problems so that you can relate to them better. But I'm just writing this as the key topic, that all the optimization problems that I'm going to talk about today, they look like that. You're trying to find an x over a cost function, where the cost function can be written as a sum.

In modern day machine learning parlance these are also called finite sum problems, in case you run into that term. And they just call it finite because n is finite here. In pure optimization theory parlance, n can actually go to infinity. And then they're called stochastic optimization problems-- just for terminology, if while searching the internet you run into some such terminology so you kind of know what it means.

So here is our setup in machine learning. We have a bunch of training data. On this slide, I'm calling x1 through xn. These are the training data, the raw features. Later, actually, I'll stop writing x for them and write them with the letter a. But hopefully, that's OK.

So x1 through xn, these could be just raw images, for instance, in ImageNet or some other image data set. They could be text documents. They could be anything. y1 through yn, in classical machine learning, think of them as plus minus 1 labels-- cat, not cat-- or in a regression setup as some real number.

So that's our training data. We have d dimensional raw vectors, n of those. And we have corresponding labels which can be either plus or minus 1 in a classification setting or a real number in a regression setting. It's kind of immaterial for my lecture right now. So that's the input.

And whenever anybody says large scale machine learning, what do we really mean? What we mean is that both n and d can be large. So what does that mean in words? That n is the number of training data points. So n could be, these days, what? A million, 10 million, 100 million, depends on how big computers and data sets you've got. So n can be huge.

d, the dimensionality, the vectors that we are working with-- the raw vectors-- that can also be pretty large. Think of x is an image. If it's a megapixel image, wow, d's like a million already. If you are somebody like Criteo or Facebook or Google, and your serving web advertisements, d-- these are the features-- could be like in several hundred million, even a billion, where they encode all sorts of nasty stuff and information they collect about you as users. So many nasty things they can collect, right?

So d and n are huge. And it's because both d and n are huge, we are interested in thinking of optimization methods for large scale machine learning that can handle such big d and n. And this is driving a lot of research on some theoretical computer science, including the search for sublinear time algorithms and all sorts of data structures and hashing tricks just to deal with these two quantities.

So here is an example-- super toy example. And I hope really that I can squeeze in a little bit of proof later on towards the end. I'll take a vote here in class to see if you are interested. Let's look at the most classic question, least squares regression. A is a matrix of observations-- or sorry, measurements. b are the observations. You're trying to solve Ax minus b whole square. Of course, a linear system of equations, the most classical problem in linear algebra, can also be written like that, let's say.

This can be expanded. Hopefully, you are comfortable with this norm. So x2 squared, this is just defined as that's the definition of that notation. But I'll just write it only once now. I hope you are fully familiar with that.

So by expanding that, I managed to write least squares problem in terms of what I call the finite sum right. So it's just going over all the roles in a. The end roles, let's say. So that's the classical least squares problem. It assumes this finite sum form that we care about.

Another random example is something called Lasso. Maybe if anybody of you has played with machine learning or statistics toolkits, you may have seen something called Lasso. Lasso is essentially least squares, but there's another simple term at the end. That again, looks like f of i.

Support vector machines, once a workhorse of-- there's still a workhorse horse of people who work with small to medium sized data. Deep learning stuff requires huge amount of data. If you have small to medium amount of data, logistic regression support, vector machines, trees, et cetera, this will be your first go to methods. They are still very widely used.

These problems are, again, written in terms of a loss over training data. So this again, has this awesome format, which I'll just now record here. I may not even need to repeat it. Sometimes I write it with a normalization-- you may wonder at some point, why-- as that finite sum problem. And maybe the example that you wanted to see is something like that.

So deep neural networks that are very popular these days, they are just yet another example of this finite sum problem. How are they an example of that? So you have n training data points, there's a neural network loss, like cross entropy, or what have you, squared loss, cross-- any kind of loss. yi's are the labels-- cat not cat, or maybe a multiclass. And then you have a transfer function called a deep neural network which takes raw images as input and generates a prediction whether this is a dog or not. That whole thing I'm just calling DNN.

So it's a function of ai's which are the training data. X are the [INAUDIBLE] matrices of the neural network, so I've just compressed the whole neural network into this notation. Once again, it's nothing but an instance of that finite sum. So that fi in there captures the entire neural network architecture. But mathematically, it's still just one particular instance of this finite sum problem.

And then people who do a lot of statistics, maximum likelihood estimation. This is log likelihood over n observations. You want to maximize log likelihood. Once again, just a finite sum.

So pretty much most of the problems that we're interested in machine learning and statistics, when I write them down as an optimization problem, they look like these finite sum problems. And that's the reason to develop specialized optimization procedures to solve such finite some problems. And that's where SGD comes in.

OK. So that's kind of just the backdrop. Let's look at now how to go about solving these problems. So hopefully this iteration is familiar to you-- gradient descent, right? OK.

So just for notation, f of x refers to that entire summation. F sub i of x refers to a single component. So if you were to try to solve-- that is, to minimize this cost function, neural

network, SVM, what have you using gradient descent, that's what one iteration would look like. Because it's a finite sum, gradients are linear operators. Gradient of the sum is the sum of the gradient-- that's gradient descent for you.

And now, I'll just ask a rhetoric question that, if you put yourself in the shoes of you're [INAUDIBLE] algorithm designers, some things that you may want to think about-- what may you not like about this iteration, given that big n, big d story that I told you? So anybody have any reservations or about drawbacks of this iteration? Any comments?

AUDIENCE: It's a pretty big sum.

PROFESSOR SRA: It's a pretty big sum. Especially if n is say, a billion on some bigger, [INAUDIBLE] number. That is definitely a big drawback. Because that is the prime drawback for large scale, that n be huge. There can be variety of other drawbacks. Some of those you may have seen previously when people compare whether to the gradient or to do Newton, et. Cetera but for the purpose of today, for finite sums, the big drawback is computing gradient at a single point-- there's a subscript xk missing there-- involves computing the gradient of that entire sum.

That sum is some is huge. So getting a single gradient to do a single step of gradient descent for a large data set could take you hours or days. So that's a major drawback. But then if you identify that drawback, anybody have any ideas how to counter that drawback, at least, say, purely from an engineering perspective?

I heard something. Can you speak up?

AUDIENCE: [INAUDIBLE]

PROFESSOR SRA: Using some kind of badge?

AUDIENCE: Yeah.

PROFESSOR SRA: You are well ahead of my slides. We are coming to that. And maybe somebody else has, essentially, the same idea. Anybody want to suggest how to circumvent that big n stuff in there? Anything-- suppose you are implementing this. What would you do?

AUDIENCE: One example at a time.

PROFESSOR SRA: One example at a time.

AUDIENCE: [INAUDIBLE] a random sample full set of n.

PROFESSOR SRA: Random sample of the full n. So these are all excellent ideas. And hence, you folks in the class have discovered the most important method for optimizing machine learning problems, sitting here in a few moments. Isn't that great? So the part that is missing is of course to make sense of, does this idea work? Why does it work?

So this idea is really at the heart of stochastic gradient descent. So let's see. Maybe I can show you an example, actually, that I-- I'll show you a simulation I found on somebody's nice web page about that. So exactly your idea, just put in slight mathematical notation, that what if at each iteration, we randomly pick some integer, i k out of the n training data points, and we instead just perform this update.

So instead of using the full gradient, you just compute the gradient of a single randomly chosen data point. So what have you done with that? One iteration is now n times faster. If n were a million or a billion wow, that's super fast. But why should this work right?

I could have done many other things. I could have not done any update and just output the 0 vector. That would take even less time. That's also an idea. It's a bad idea, but it's an idea in a similar league.

I could have done a variety of other things. Why would you think that just replacing that sum with just one random example may work? Let's see a little bit more about that.

So of course, it's n times faster, and the key question for us here, right now-- the scientific question-- is does this make sense? It makes great engineering sense. Does it make algorithmic or mathematical sense? So this idea of doing stuff in the stochastic manner was actually originally proposed by Robbins and Monro, somewhere, I think, around 1951. And that's the most advanced method that we are essentially using currently.

So I'll show you that this idea makes sense. But maybe let's first just look at a comparison of SGD with gradient descent in this guy's simulation. So this is that MATLAB code of gradient descent, and this is just a simulation of gradient descent. As you pick a different step size, that gamma in there, you move towards the optimum. If the step size is small, you make many small steps, and you keep making slow progress, and you reach there.

That's for a well-conditioned problem and an ill-conditioned problem. It takes you even larger. In a neural network type problem which is nonconvex, you have to typically work with smaller

step sizes. And if you take bigger ones, you can get crazy oscillations. But that's gradient descent.

In comparison, let's hope that this loads correctly. Well, there's even a picture of Robbins, who was a co-discoverer of the stochastic gradient method. There's a nice simulation, that instead of making that kind of deterministic descent-- after all, gradient descent is called "gradient descent." At every step it descends-- it decreases the cost function.

Stochastic gradient descent is actually a misnomer. At every step it doesn't do any descent. It does not decrease the cost function. So you see, at every step, those are the contours of the cost function. Sometimes it goes up, sometimes it goes down. It fluctuates around, but it kind of stochastically still seems to be making progress towards the optimum.

And stochastic gradient descent, because it's not using exact gradients, just working with these random examples, it actually is much more sensitive to step sizes. And you can see, as I increase the step size, its behavior. This is actually full simulation for [INAUDIBLE] problem.

So initially, what I want you to notice is-- let me go through this a few times-- keep looking at what patterns you may notice in how that line is fluctuating. Hopefully this is big enough for everybody to see. So this slider that I'm shifting is just the step size. So let me just remind you, in case you forgot, the iteration. We are running x k plus 1 is x k minus some eta k-- It's called alpha there-- times some randomly chosen data point.

You compute its gradient. This is SGD. That's what we are running. And we threw away tons of information. We didn't use the full gradient. We're just using this crude gradient.

So this process is very sensitive to the other parameter in the system, which is the step size. Much more sensitive than gradient descent, in fact. And let's see. As I vary the step size, see if you can notice some patterns on how it tries to go towards an optimum.

There's a zoomed in version, also, of this later, here. I'll come to that shortly. I'll repeat again, and then I'll ask you for your observations-- if you notice some patterns. I don't know if they're necessarily apparent. That's the thing with patterns. Because I know the answer, so I see the pattern. If you don't know the answer, you may or may not see the pattern. But I want to see if you actually see the pattern as I change the step size.

So maybe that was enough simulation. Anybody have any comments on what kind of pattern

you may have observed? Yep.

AUDIENCE: It seems like the clustering in the middle is getting larger and more widespread.

PROFESSOR SRA: Yeah, definitely. That's a great observation. Any other comments?

There's one more interesting thing happening here, which is a very, very typical thing for SGD, and one of the reasons why people love SGD. Let me do that once again briefly. OK, this is tiny step size-- almost zero. Close to zero-- it's not exactly zero.

So you see what happens for a very tiny step size? It doesn't look that stochastic, right? But that's kind of obvious from there if eta k is very tiny, you'll hardly make any move. So things will look very stable. And in fact, the speed at which stochastic gradient converges, that's extremely sensitive to how you pick the step size. It's still an open research problem to come up with the best way to pick step sizes. So it's even that simple, it doesn't mean it's trivial.

And as I vary the step size, it make some progress, and it goes towards the solution. Are you now beginning to see that it seems to be making a more stable progress in the beginning? And when it comes close to the solution, it's fluctuating more. And the bigger the step size, the amount of fluctuation near the solution is wilder as he noticed back there.

But one very interesting thing is more or less constant. There is more fluctuation also on the outside, but you see that the initial part still seems to be making pretty good progress. And as you come close to the solution, it fluctuates more. And that is a very principally typical behavior of stochastic gradient descent, that in the beginning, it makes rapid strides. So you may see your training loss decrease super fast and then kind of peter out.

And it's this particular behavior which guard people super excited, that, hey, in machine learning, we are working with all sorts of big data. I just want a quick and dirty progress on my training. I don't care about getting to the best optimum. Because in machine learning, you don't just care about solving the optimization problem, you actually care about finding solutions that work well on unseen data.

So that means you don't want to over fit and solve the optimization problem supremely well. So it's great to make rapid initial progress. And if after that progress peters out, it's OK. This intuitionistic statement that I'm making, in some nice cases like convex optimization problems, one can mathematically fully quantify these. One can prove theorems to quantify each thing that I said in terms of how close, how fast, and so on. We'll see a little bit of that.

And this is what really happens to SGD. It makes great initial progress, and regardless of how you use step sizes, close to the optimum it can either get stuck, or enter some kind of chaos dynamics, or just behave like crazy. So that's typical of SGD.

And let's look at now slight mathematical insight into roughly why this behavior may happen. This is a trivial, one-dimensional optimization problem, but it conveys the crux of why this behavior is displayed by stochastic gradient methods. That it works really well in the beginning, and then, God knows what happens when it comes close to the optimum, anything can happen. So let's look at that.

OK. So let's look at a simple, one-dimensional optimization problem. I'll kind of draw it out maybe on the other side so that people on this side are not disadvantaged. So I'll just draw out at least squares problem-- x is one dimensional. Previously, I had ai transpose x Now, ai is also a scalar. So it's just 1D stuff-- everything is 1D

So this is our setup. Think of ai into x minus b i. These are quadratic functions, so they look like this. Corresponding to different eyes, there's like some different functions sitting and so on.

So these are my n different loss functions, and I want to minimize those. We know-- we can actually explicitly compute the solution of that problem. So you set the derivative of f of x to 0 so. You set the gradient of f of x to 0. Hopefully, that's easy for you to do.

So if you do that differentiation, will get gradient of f of x will just given by. Well, you can do that in your head, I'll just write it out explicitly. aix minus bi times ai Is equal to zero, and you solve that for x. You get x star, the optimum of this least squares problem. So we actually know how to solve it pretty easily.

That's a really cool example actually. I got that from textbook by Professor Dimitry [INAUDIBLE]. Now, a very interesting thing. We are not going to use the full gradient, we are only going to use the gradients of individual components. So what does the minimum of an individual component look like?

Well, the minimum of an individual component is attained when we can set this thing to 0. And that thing becomes 0 if we just pick x equal to bi divided by ai, right? So a single component can be minimized by that choice.

So you can do a little bit of arithmetic mean, geometric mean type inequalities to draw this picture. So over all i from 1 through n, this is the minimum value of this ratio, ai by bi. And let's say this is the max value of ai by bi.

And we know that closed form solution, that is the true solution. So you can verify with some algebra that that solution will lie in this interval. So you may want to-- this is a tiny exercise for you. Hopefully some of you love inequalities like me. So this is hopefully not such a bad exercise. But you can verify that within this range of the individual mins and max is where the combined solution lies. So of course, intuitively, with a physics styles thinking, you would have guessed that right away.

That means when you're outside where the individual solutions, let's call this the far out zone. And also, this side is the far out zone. And this region, within which the true minimum can lie, you can say, OK, that's the region of confusion.

Why I'm calling it the region of confusion? Because there, by minimizing an individual fi, you're not going to be able to tell what is the combined x star. That's all. And a very interesting thing happens now, just to gain some mathematical insight into that simulation that I showed you, that if you have a scalar x that is outside this region of confusion, which states that if you're far from the region within which an optimum can lie. So you're far away.

So you've just started out your progress, you made a random initialization, most likely far away from where the solution is. So suppose that's where you are. What happens when you're in that far out region? So if you're in the far out region, you use a stochastic gradient of some i-th component.

So the full gradient will look like that. A stochastic gradient looks like just one component. And when you're far out, outside that min and max regime, then you can check by just looking at it, that a stochastic gradient, in the far away regime, has exactly the same sign as the full gradient.

What does gradient descent do? It says, well, walk in the direction of the negative gradient. And far away from the optimum, outside the region of confusion, you're stochastic gradient has the same sign as the true gradient. Maybe in more linear algebra terms, it makes an acute angle with your gradient. So that means if even though a stochastic gradient is not exactly the full gradient, it has some component in the direction of the true gradient.

This is one 1D. Here it is, exactly the same sign. In multiple dimensions, this is the idea that it'll have some component in the direction of true gradient when you're far away. Which means, if you then use that direction to make an update in that style, you will end up making solid progress. And the beauty is, in the time it would have taken you to do one single iteration of batch gradient descent, far away you can do millions stochastic steps, and, each step will make some progress.

And that's where we see this dramatic, initial-- again, in the 1D case this is explicit mathematically. In the high-D case, this is more intuitive. Without further assumptions about angles, et, we can't make such a broad claim. But intuitively, this is what's happening, and why you see this awesome initial speed.

And once you're inside the region of confusion, then this behavior breaks down. Some stochastic gradient may have the same sign as the full gradient, some may not. And that's why you can get crazy fluctuations. So this simple 1D example kind of exactly shows you what we saw in that picture.

And people really love this initial progress. Because, often we also do early stopping. You train for some time, and then you say, OK, I'm done.

So importantly, if you are purely an optimization person, not thinking so much in terms of machine learning, then please keep in mind that stochastic gradient descent or stochastic gradient method is not such a great optimization method. Because once in the region of confusion, it can just fluctuate all over forever. And in machine learning, you say, oh, the region of confusion, that's fine. It'll make my method robust. It'll make my neural network training more robust. It's generalize better, et cetera, er cetera-- we like that.

So it depends on which frame of mind you're in. So that's the awesome thing about the stochastic gradient method.

So I'll give you now key mathematical ideas behind the success of SGD. This was like little illustration. Very abstractly, this is an idea that [INAUDIBLE] throughout machine learning and throughout theoretical computer science and statistics, anytime you're faced with the need to compute an expensive quantity, resort to randomization to speed up the computation. SGD is one example. The true gradient was expensive to compute, so we create a randomized estimate of the true gradient. And the randomized estimate is much faster to compute.

And mathematically, what will start happening is, depending on how good your randomized estimate is, your method may or may not convert to the right answer. So of course, one has to be careful about what particular randomized estimate one makes. But really abstractly, even if I hadn't shown you, the main idea, this idea you can apply in many other settings. If you have a difficult quantity, come up with a randomized estimate and save on computation. This is a very important theme throughout machine learning and data science.

And this is the key property. So stochastic gradient descent, it uses stochastic gradients. Stochastic is, here, used very loosely. And it just means that some randomization. That's all it means.

And the property-- the key property that we have is in expectation. The expectation is over whatever randomness you used. So if you picked some random training data point out of the million, then the expectation is over the probability distribution over what kind of randomness you used. If you picked uniformly at random from a million points, then this expectation is over that uniform probability.

But the key property for SGD, or at least the version of SGD I'm talking about, is that that over that randomness. The thing that you're pretending to use, instead of the true gradient n expectation actually it is the true gradient. So in statistics language, this is called the stochastic gradient that we use is an unbiased estimate of the true gradient. And this is a very important property in the mathematical analysis of stochastic gradient descent, that it is an unbiased estimate, And

Intuitively speaking anytime you did any proof in class, or in the book, or lecture, or to wherever, where you were using true gradients, more or less, you can do those same proofs-- more or less, not always. Using stochastic gradients by encapsulating everything within expectations over the randomness. I'll show you an example of what I mean by that. I'm just trying to simplify that for you. And

In particular, the unbiasedness is great. So it means I can kind of plug-in these stochastic gradients in place of the true gradient, and I'm still doing something meaningful. So this is answering that earlier question, why this random stuff? Why should we think it may work?

But there's another very important aspect to why it works, beyond this unbiasedness, that the amount of noise, or the amount of stochasticity is controlled. So just because it is an unbiased estimate, doesn't mean that it's going to work that well.

Why? Because it could still fluctuate hugely, right? Essentially, plus infinity here, minus infinity here. You take an average, you get 0. So that is essentially unbiased, but the fluctuation is gigantic. So whenever talking about estimates, what's the other key quantity we need to care about beyond expectation?

AUDIENCE: Variance.

AUDIENCE: Variance.

PROFESSOR SRA: Variance. And really, the key thing that governs the speed at which stochastic gradient descent does the job that we want it to do is, how much variance do the stochastic gradients have?

Just this simple statistical point, in fact, is at the heart of a sequence of research progress in the past five years in the field of stochastic gradient, where people have worked really hard to come up with newer and newer, fancier and fancier versions of stochastic gradient which have the unbiasedness property, but have smaller and smaller variance. And the smaller the variance you have, the better your stochastic gradient is as a replacement of the true gradient. And of course, the better [INAUDIBLE] of the true gradient, then you truly get that n times up.

So the speed of convergence depends on how noisy the stochastic gradients are. It seems like I'm going too slow. I won't be able to do a proof, which sucks. But let me actually tell you then about, rather than the proof, I think I'll share the proof with Gil. Because the proof that I wanted to actually show you, gives a proof of stochastic gradient is well-behaved on both convex and nonconvex problems. And the proof I wanted to show was for the nonconvex case, because it applies to neural networks. So you may be curious about that proof. And remarkably, that proof is much simpler than the case of convex problems.

So let me just mention some very important points about stochastic gradient. So even though this method has been around since 1951, every deep learning tool kit has it, and we are studying it in class, there are still gaps between what we can say theoretically and what happens in practice. And I'll show you those gaps already, and encourage you to think about those if you wish.

So let's look back at our problem and deliver two variants. So here are the two variants. I'm going to ask if any of you is familiar with these variants in some way or the other. So I just call

it feasible. Here, there are no constraints. So start with any random vector of your choice. In deep network training you have to work harder.

And then, this is the iteration you run-- option 1 and option 2. So option 1 says, that was the idea we had in class, randomly pick some training data point, use its stochastic gradient. What do we mean by randomly pick? The moment you use the word random, you have to define what's the randomness.

So one randomness is uniform probability over n training data points. That is one randomness. The other version is you pick a training data point without replacement. So with replacement means uniformly at random. Each time you draw a number from 1 through n, use their stochastic gradient, move on. Which means the same point can easily be picked twice, also.

And without replacement means, if you've picked a point number three, you're not going to pick it again until you've gone through the entire training data set. Those are two types of randomness. Which version would you use?

There is no right or wrong answer to this. I'm just taking a poll. What would you use? Think that you're writing a program for this, and maybe think really pragmatically, practically. So that's enough of a hint. Which one would you use-- I'm just curious.

Who would use 1? Please, raise hands. OK. And the exclusion-- the compliment thereof. I don't know. Maybe some people are undecided. Who would use 2? Very few people. Ooh, OK.

How many of you use neural network training toolkits like TensorFlow, PyTorch, whatnot? Which version are they using?

Actually, every person in the real world is using version 2. Are you really going to randomly go through your RAM each time to pick random points? That'll kill your GPU performance like anything.

What people do is take a data set, use a pre-shuffle operation, and then just stream through the data. What does streaming through the data mean? Without replacement. So all the toolkits actually are using the without replacement version, even though, intuitively, uniform random feels much nicer.

And that feeling is not ill-founded, because that's the only version we know how to analyze mathematically. So even for this method, everybody studies it. There are a million papers on it.

The version that is used in practice is not the version we know how to analyze. It's a major open problem in the field of stochastic gradient to actually analyze the version that we use in practice.

It's kind of embarrassing, but without replacement means non-IAD probability theory, and non-IAD probability theory is not so easy. That's the answer. OK.

So the other version is this mini-batch idea-- which you mentioned really early on-- is that rather than pick one random point, I'll pick a mini batch. So I had a million points-- each time, instead of picking one, maybe I'll pick 10, or. 100, or 1,000, or what have you. So this averages things. Averaging things reduces the variance. So this is actually a good thing, because the more quantities you average, the less noise you have. That's kind of what happened in probability.

So we pick a mini-batch, and the stochastic estimate now is this not just a single gradient, but averaged over a mini-batch. So a mini-batch of size 1 is the pure vanilla SGD. Mini-batch of size n is nothing other than pure gradient descent. Something in between is what people actually use. And again, the theoretical analysis only exists if the mini-batch is picked with replacement not without replacement.

So one of the reasons actually-- a very important thing-- in theory, you don't gain too much in terms of computational gains on convergent speed by using mini-batches. But mini-batches are really crucial, especially in the deep learning, GPU-style training, because they allow you to do things in parallel. Each thread or each core or subcore or small chip or what have, you depending on your hardware, can be working with one stochastic gradient. So mini-batches, the larger the mini batch the more things you can do in parallel.

So mini-batches are greatly exploited by people to give you a cheap version of parallelism. And where does the parallelism happen? You can think that each core computes a stochastic gradient. So the hard part is not adding these things up and making the update to x, the hard part is computing a stochastic gradient. So if you can compute 10,000 of those in parallel because you have 10,000 cores, great for you. And that's the reason people love using mini-batches.

But a nice side remark here, this also brings us closer to the research edge of things again. That, well, you'd love to use very large mini-batches so that you can fully max out on the parallelism available to you. Maybe you have a multi-GPU system, if you're friends with nVidia

or Google. I only have two GPUs.

But it depends on how many GPU shows you have. You'd like to really max out on parallelism so that you can really crunch through big data sets as fast as possible. But you know what happens with very large mini-batches? So if you have very large mini-batches, stochastic gradient starts looking more like? Full gradient descent, which is also called batch gradient descent.

That's not a bad thing. That's awesome for optimization. But it is a weird conundrum that happens in training deep neural networks. This type of problem we wouldn't have for convex optimization. But in deep neural networks, this is really disturbing thing happens, that if you use this very large mini-batches, your method starts resembling gradient descent. That means it decreases noise so much so that this region of confusion shrinks so much-- which all sounds good, but it ends up being really bad for machine learning.

That's what I said, that in machine learning you want some region of uncertainty. And what it means actually is, a lot of people have been working on this, including at big companies, that if you reduce that region of uncertainty too much, you end up over-fitting your neural network. And then it starts sucking in its test data, unseen data performance. So even though for parallelism, programming, optimization theory, big mini-batch is awesome, unfortunately there's price to be paid, that it hurts your test error performance.

And there are all sorts of methods people are trying to cook up, including shrinking data accordingly, or chaining neural network architecture, and all sorts of ideas. You can cook up your ideas for your favorite architecture, how to make a large mini-batch without hurting the final performance. But it's still somewhat of an open question on how to optimally select how large your mini-batch should be. So even though these ideas are simple, you see that every simple idea leads to an entire sub area of SGD.

So here are practical challenges. People have various heuristics for solving these challenges. You can cook up your own, but it's not that one idea always works. So if you look at SGD, what are the moving parts? The moving parts in SGD-- the gradients, stochastic gradient, the step size, the mini batch. So how should I pick step sizes-- very non-trivial problem.

Different deep learning toolkits may have different ways of automating that tuning, but it's one of the painful things. Which mini batch to use? With replacement, without replacement I already showed you. But which mini batch should I use, how large that should be? Again, not

an easy question to answer.

How to compute stochastic gradients. Does anybody know how stochastic gradients are computed for deep network training? Anybody know? There is a very famous algorithm called back propagation. That back propagation algorithm is used to compute a single stochastic gradient.

Some people use the word back prop to mean SGD. But what back prop really means is some kind of algorithm which computes for you a single stochastic gradient. And hence this TensorFlow, et cetera-- these toolkits-- they come up with all sorts of ways to automate the computation of a gradient. Because, really, that's the main thing.

And then other ideas like gradient clipping, and momentum, et cetera. There's a bunch of other ideas. And the theoretical challenges, I mentioned to you already-- proving that it works, that it actually solves what it set out to do. Unfortunately, I was too slow. I couldn't show you the awesome five-line proof that I have that SGD works for neural networks. And theoretical analysis, as I said, it's really laggy.

My proof also uses the with replacement. And the without replacement version, which is the one that is actually implemented, there's very little progress on that. There is some progress. There's a bunch of papers, including from our colleagues at MIT, but it's quite unsolved.

And the biggest question, which most of the people in machine learning are currently excited about these days is stuff like, why does SGD work so well for neural networks? We use this crappy optimization method, it very rapidly does some fitting-- the data is large, neural network is large, and then this neural network ends up having great classification performance. Why is that happening? It's called trying to explain-- build a theory of generalization. Why does an SGD-trained neural network work better than neural networks train with more fancy optimization methods?

It's a mystery, and most of the people who take interest in theoretical machine learning and statistics, that is one of the mysteries they're trying to understand. So I think that's my story of SGD. And this is the part we skipped, but it's OK. The intuition behind SGD is much more important in this.

So I think we can close.

PROFESSOR STRANG: Thank you.

[APPLAUSE]

And maybe I can learn the proof for Monday's lecture.

PROFESSOR SRA: Exactly. Yeah, I think so. That'll be great.