RUSS TEDRAKE: OK, so every once in a while, I stop and try to do a little bit of reflection, since we've-- have so many methods flying through this semester that I want to just-- once again, let's say where we've been, where we're going, what we have, what we can do, what we can't do, and why we're going to do something different today-- so just a little reflection here.

We've been talking, obviously, about optimal control. There's two major approaches to optimal control that we've focused on-- well, three, I guess. In some cases, we've done analytical optimal control. And I think, by now, you appreciate that, although it was often-- only works in special cases-- linear quadratic regulators and things like that-- the lessons we learned from that help us design better algorithms.

And things like LQR can fit right into more complicated non-linear algorithms to make them click, so I think, absolutely, it's essential to understand the things we can do analytically in optimal control-- even though they crap out pretty early in the scale of complexity that we care about. So mostly, that's good for linear systems and even restricted there, linear systems with quadratic costs and things like that.

And then we talked about major direction number two was the dynamic programming and value iteration approach. And the big idea there right was that, because we've written our cost functions over time to be additive, the big idea really was that we're going to learn-- we're going to figure out the cost-to-go function-- the value function, the value iteration-- and from there, we can just extract that.

That captures all of the long-term reasoning we have to do about the system. From that, we can extract the optimal control decisions. And it's actually very efficient. I hope, by now, you agree with me that it's very efficient, because if you think about it, it's solving for an entire-- solving for the optimal policy for every possible initial condition in times that are comparable to what we're doing for single initial conditions in the loop cases. But it only works in low dimensions, and it has some discretization issues.

And then the third major approach-- I called it policy search. And we focused mostly, in the policy search, on loop trajectory optimization. But I tried to make the point early-- and I'm going to make the point again in a lecture or two-- that it's really not restricted to thinking about loop trajectories.

So when I first said policy search, I said we could be looking for parameters of a feedback controller of a-- the linear gain matrix of a linear feedback. We could do a lot of things, but we quickly started being specific in our algorithms in trying to optimize some loop tape with direct colocation with shooting. But the ideas really are more general than that, and I'm going to-- we're going to have a lecture soon about how to do these kind of things with function approximation, and do more general feedback controllers. These worked in higher dimensional systems, had local minima-- all the problems you know by now.

OK, good-- so for our model systems, we got pretty far with that. In the cases where we knew the model, we assumed that the model was deterministic and sensing was clean-- everything like that. We could make our simulations do pretty much what we wanted with those bag of tricks.

Then I threw in the stochastic optimal control case. We said, what happens if the models aren't deterministic? Analytical optimal control-- I didn't really talk about it, but there are still some cases where you can do analytical optimal control. The linear quadratic Gaussian systems are the clear example of that.

We said that value iteration for this-- although I was quickly challenged on it, I said, basically, it was no harder to do value iteration stochastic optimization, where now our goal is to minimize some expected value of a long-term cost. Value iteration we basically said and almost no harder to do the case with transition probabilities flying around.

And in fact, the barycentric grids that we used in the value duration way back there I told you actually has a more clean interpretation as taking a continuous-- you can think of it as being a continuous deterministic system, and converting it into a discrete state stochastic system.

Remember, the interpolation that you do in the barycentric actually takes exactly the same form as some transition probabilities when you're going from-- you've got some grid, and you want to know where you're going to go from simulating forward from this with some action for some dt, you can approximate that as being some fraction here, some fraction here, some fraction here.

And it turns out to be exactly equivalent to saying that there's some probability I get here, some probability I get here, some probability I get here, and the like. So value iteration really, in that sense, can solve stochastic problems nicely.

The other major approach, the policy search, can still work for stochastic problems. In some cases, you can compute the gradient of the expected value with respect to your parameters analytically, with a [INAUDIBLE] update. In other cases, you can do sampling based, Monte Carlo based estimates, and I'm going to get more into that.

We're going to talk more about that, but the takeaway messages, when things get stochastic, both of these methods still work. And they work in slightly different ways, but you can make both of those work And then, last week, John threw a major wrench into things. Sorry. That wasn't supposed to be a statement about John. John made your life better by telling you that some of these statements-- some of these algorithms work even if you don't know the model.

And he talked about doing policy search without a model. And the big idea there was that actually fairly simple looking algorithms, which just perturb the-- which try different parameters-- run a trial, try different parameters-- simple sampling algorithms can estimate the same thing we would do with our policy gradient, the gradient of the expected reward with respect to the parameters.

Even the simplest thing is let's change my parameters a little bit, see what happened. That gives me a sample from this gradient. And if I do it enough times, I pull enough samples, and I can-- I get a sample of the expected returns. If you think back, that's why we try to stick in stochastic optimal control before we got to that, because John also told you the nice interpretation of these algorithms in the stochastic-- that, even if the plant that you're measuring from is stochastic or if the sensors are-- there's noise, then actually, still, these same sampling algorithms can estimate these gradients for you nicely.

I think John also made the point-- and I want to make it again-- you would never use these algorithms if you had a model. They're beautiful, but probably, if you have a model-- maybe, if you have a model and you're a very patient person, but very lazy, then you might try to use this, because you can type it in in a few minutes, but it's going to take a lot longer to run.

And the reason for that is it's going to require many [INAUDIBLE] requires many more simulations. In fact, it just requires many simulations to estimate a single gradient-- policy gradient.

Now, the next thing I'm going to say is a little more controversial, but most people would say that the limiting case, the best you could possibly do with these reinforce type algorithms-- are you raising your hand or just--

AUDIENCE: [INAUDIBLE]

RUSS TEDRAKE: No-- sorry. The best thing you can do with these reinforced algorithms, if you really-- the best performance you could expect is a shooting algorithm. And it's really, I should say, a first-order shooting method. It's really just doing gradient descent by doing trials.

And when we talked about shooting methods, I actually said never do first-order shooting methods. I made a big point. I said never do this, never do this-- because if you go to the second-order methods, things converge faster. You don't have to pick learning rates. You can handle constraints.

So there are people that do a bit of more second-order policy gradient algorithms, but that's not the standard yet. So you should really think of those as cool systems that, if you-- cool algorithms that, if you don't have a model, you can almost do a shooting method. Why do I say that's a controversial statement? Could you imagine somebody standing up and saying, this is actually better than doing gradient descent?

AUDIENCE: [INAUDIBLE]

RUSS TEDRAKE: Yeah. So the one advantage is that it's doing stochastic gradient descent. And there are people out there that really believe stochastic gradient descent can outperform even higher order methods in certain cases, just because of their ability to, by virtue of being random-- this is not some magical property we've endowed. This is [? Because ?] the algorithm is a little crazy. It bounces out of local minima. So for that reason, it does have all the strong optimization claims that a stochastic gradient descent algorithm has.

There's another point to make, though, and I think John made this too. The performance of this-- and John's done nice-- written nice paper on this-- the performance you'd expect, meaning the number of trials it would take to learn to optimize your cost function-- the performance of these reinforced type algorithms-- it degrades with a number of parameters you're trying to tune.

So remember, the fundamental idea was-- and the way I like to think of it is, imagine you're at a mixer station in a sound recording studio, and you're looking through the glass, and you've got a robot over there. You've got all your knobs set in some place, and your robot does its behavior, and then you give it a score.

You turn your knobs a little bit, you see how the robot acts. You turn it off a little bit more. And your job is to just twist these knobs in a way that finds the way down the gradient, and gets your robot doing what you want to do. That maybe is a demystifying way to think about this everything, which is mathematically beautiful, but really, it's just turning knobs.

If you have a model and you can compute the gradient, then you don't have to guess the way you turn knobs. You should always use that model to turn the knobs in the right direction. And also, if you think about that analogy, the number of-- the length of time it's going to take you to optimize your function is going to depend on how many knobs you have to turn.

If I have 100 knobs in front of me and I change them all a little bit-- I see how my robot acted-- then it's going to be hard for me to figure out exactly which knob to assign credit to. The fewer knobs I have to change, the faster I can estimate which knobs were important, and climb down a gradient.

I still say, when you have a model, you should always use it, because you can estimate the gradients. You can turn the knobs in the right way. But in the case where you don't have a model, it's actually-- they're very nice classes of algorithms. This knob tuning thing sounds ridiculous. Maybe, if I have even an [INAUDIBLE], if you have a good model of the [INAUDIBLE], then maybe you shouldn't-- you should definitely be using it.

But if you have a very complicated system, and the performance only depends on the number of parameters, then it-- I just want to make the point that it's-- they're actually pretty powerful for some control problems. And the ones that we're working on in my group are fluid dynamics control problems, but specifically if you have problems where you can get away with the small number of parameters, but you have a very complicated unknown dynamics.

And actually, those algorithms are really-- make a lot of sense to me. So the performance of these randomized policy search algorithms-- it goes with the number of parameters you're trying to tune. I could be sitting in this mixing station, and I could be twittering four parameters and having a simple pendulum do its thing, or I could be sitting in there turning these four knobs and having a Navier-Stokes simulation, with some very complicated fluid doing something, and the amount of time it takes me to twiddle those parameters is the same.

One of the strongest properties of these algorithms is that, by virtue of ignoring the model, they're actually insensitive to the model complexity. So in my group, we're really trying to push-- in some problems where the dynamics are unknown and very complicated and a lot of the community is trying to build better models of this, we're trying to say, well, maybe before you have perfect models, we can do some of these model-free search algorithms to build good controllers without perfect models.

Are people OK with that array of techniques? Yeah? You have a good arsenal of tools? Can you see the obvious place where I'm trying to go next, now that I've set it up like this? We did value methods and policy search methods for the simple case, then we did value methods and policy search methods for the stochastic case, then we did policy methods for the model-free case. So how about we do model-free value methods today?

But I know it's a complicated web of algorithms, so I want to make sure that I stop and say that kind of stuff every once in a while. So what's the difference between a policy method and a value method? So value duration-- like I said, it's very, very efficient.

The way we represented value iteration with a grid, and having to solve every possible state at every possible time, is the extreme form of value-- of the value methods. In general, we can try to build approximate value methods-- estimates of our value function that don't require the big discretization.

So actually, last week, at the meeting-- one of the meetings I was at, I met Gerry Tesauro. And Gerry Tesauro is the guy who did TD-Gammon. Anybody heard of TD-Gammon? Yeah? [INAUDIBLE] knows TD-Gammon. I don't know what year it was. It was 20 years ago now. One of the big success stories for reinforcement learning was that they built a game player based on reinforcement learning that could play backgammon with the experts and beat the experts at backgammon.

Now, backgammon's actually not a trivial game. It's got a huge state space-- huge state space. I don't play backgammon, but I know there's a lot of bits going around there. It's stochastic, because you roll a die every once in a while. So it's actually not some complicated-- not some simple game.

In some ways, it's surprising that it was solved before checkers and these others. Maybe it's just because not enough people play backgammon, so you can beat the experts easier. I don't know. But we were playing competition style-- beat the best humans at backgammon-- well before checkers and chess, because of a value-based-- model-free, value-based method for backgammon.

So Gerry Tesauro actually use neural networks, and he learned, from watching the game, a value function for the game. What does that mean? So what do you do when you play backgammon-- or whatever game you play? I'm not trying to dump on that game. I just haven't played it myself.

So if you look at a go board or a chess board, you don't think about every single state that's possibly in there, but you're able to quickly look at the board and get a sense of if you're winning or losing. If you were to make this move, my life should get better. And there are serious people that think that the natural representation for very complicated physical control processes or very complicated game playing scenarios is to not learn actually the policy directly, but to just learn a sense of what's good and what's bad directly, learn a value function directly.

And then we [INAUDIBLE] from value iteration. That captures all that's hard about-- that captures the entire long-term look ahead in the optimal control problem. Once I have a value function, if I have a value function I believe, if I want to make an action, all I have to do is think about, well, if I made this action, my value would get better by this much. If I made this action, my value would get better by this much. And I just pick the action that maximizes my expected value.

Now, the good thing about value-based methods is that they tend to be very efficient. You can simultaneously think about lots of different states at a time. Just like value duration, it's very efficient to learn value methods. And historically, in the reinforcement learning world, nobody ever really did policy search methods until the early '90s.

There was at least 15 years where people were doing cool things with robots, and game playing, and things like that, where almost everybody, every paper was talking about, how do you learn a value function? How do you learn a value function if you have to put in a function approximator? Or how do you do a value function if this, if this?

So really, even though I did it second, this was actually the core of reinforcement learning for a long time. How do you learn a value function? How do you estimate the cost-to-go-- ideally, the optimal cost-to-go-- given trial and error experience with the robot? So that's today's problem.

Good-- so we can make it easier by thinking about a sub-problem first. And that's really policy evaluation, which is the problem of, given I have-- I have my dynamics, of course, and some policy pi, I want to estimate or compute J of pi, the long-term potentially expected reward of executing that feedback policy on that robot, potentially from all states at all times.

So this is maybe equivalent to what I just said about chess. So my value function for chess might look different than somebody who knows how to play chess. I look at the board, and most of the time, I'm losing, and my actions are going to be chosen differently, because I wouldn't even know what to do if my rook ended up over there.

And the optimal value function, given I was acting optimally, might look very different. But for me, the first problem is just estimate what's my cost-- the cost of executing my current game playing strategy, my current control-- feedback controller on this robot, or this game?

Now, there is something culturally different from the reinforcement learning value-based communities, and I'm going to go ahead and make that switch now. Most of the time, these things are infinite horizon discounted problems. I'll say it's discrete time just to keep it clean, because then it's easy to write [INAUDIBLE] equals t to infinity here, gamma to the-- let me just do it like this.

Let's assume that it's completely feedback. That'll just keep me writing less symbols for the rest of the lecture here. 0 to infinity, gamma to the n, xn, and then pi of xn, where my action is always pulled directly from pi-- I mentioned it once before, but why do people do discounted things?

Lots of reasons why people do discounted things-- first of all, if you have infinite horizon rewards, there's just a practical issue. If you're not careful, infinite horizon rewards will blow up on you. So if you put some sort of decaying factor gammas, typically, it's constrained to be less than 1 just so you don't have to worry about things blowing up in the long term.

But you can make it 1, and then you just have to be more careful that you get to a fixed point [INAUDIBLE] cost, or whatever it is. Let's just put some decaying cost on future experiences. Philosophically, some people really like this. So a lot of the problems we've talked about are very episodic in nature. We talked about designing trajectories from time 0 to time final. What's the optimal thing? What's the optimal thing?

If you just want to live your life-- presumably, you don't know exactly when you're going to die. You're going to maximize some long-term reward. You'd like it to be infinite, but realistically, the things that are going to happen to me tomorrow are more important to me that the things that are happening in the very far, distant future.

So some people, philosophically, just like having this as a cost function for a robot that's alive executing an online policy, worrying about short-term things a little bit more, but thinking about into the future. And that knob is controlled by gamma. Almost all of the RL tools can be made compatible with the episodic non-discounted cases, but culturally, like I said, they're almost always written in this form, so I thought it'd makes sense to switch to that form for a little bit.

So how do we estimate J pi of x, given that kind of a setup? Let's do the model-based case, just as a first case. Let's say I have a good model. I made it look deterministic here, but we can, in general, do this for stochastic things. Let me do the model-based Markov chain version first.

So you remember, in general, we said that the optimal control problem for discrete states, discrete actions, stochastic transitions looked like a Markov decision process, where we have some discrete state space, we have a probability transition matrix, where T-I-J is probability of transitioning from I to J. And we have some cost.

And in the graph sense, I tend to write-- we tend to write the cost as-- instead of being [INAUDIBLE] action, we can just write it as the probability of-- the cost of transitioning from state I to state J. Good-- now, in the Markov decision processes that we talked about before, the transition matrix was a function of the action you chose. Your goal was to choose the action, which made your transition matrices have the optimal-- choose the best transition matrices for your problem.

In policy evaluation, where we're saying, we're trying to figure out the probability of the cost-to-go of running this policy, then the actions are-- the parameterization by action disappears again. It's not a Markov decision process. It falls back right into being a Markov chain.

So it's a simple picture now. We have a graph there's some probabilities of transitioning from each state to each action, from each state, because my actions are predetermined. If I'm in some state, I'm going to take this action based on pi.

And each transition incurs some cost, and my goal is to move around the graph in a way that incurs minimal long-term cost and expected value. So that's a good way to start figuring out how to do policy evaluation.

So now, in this discrete state transition matrix [INAUDIBLE] this form, I'm going to rewrite J pi as being a function of i, where i is from-- i is drawn from S, some-- it's one discrete state. And it's the expected value of G-I-N to I [INAUDIBLE] plus 1.

I should say another funny example I just remembered. So I gave an analogy of playing a game. You might look at the board and figure out what's the value of being in certain states. People think it's relevant in your brains too. So there's actually a lot of work in neuroscience these days which probes activity of certain neurons in your brain, and finds neurons that basically respond with the expected value of your cost-to-go function.

They have monkeys doing these tasks, where they pull levers or blink at the right time, and get certain rewards. And there's neurons that fire correlated with their expected reward in ways that are-- they design an experiment so it doesn't look like something that's correlated with the action they're going to choose, but it does look like it's correlated with expected reward.

And interestingly, when they learn-- as the monkeys learn during the task, you can actually see that they start making predictions accurately when they're close to the reward. They're about to get juice, and then, a few minutes later, they can predict when they're a minute away from getting juice.

And then, if you look at it a couple of days in, they're able to predict when they're a half hour from getting juice or something like this. I think the structure of trying to learn the value function is very real, especially if you're a juice-deprived monkey. So let's continue on here.

How do you compute J pi, given this equation? J is a vector now. Well, first of all the dynamic programming recursion let's us write it like this. J of ik is the expected value of taking one step-- the one step cost plus the discount factor times the future.

The reason people choose this form for the discount factor is that the Bellman recursion just looks like that. You just put a gamma in front of everything. We can take the expected value of this with our Markov chain notation and say it's the sum over i k plus 1's of Tik ik plus 1 times g ik, ik plus 1. Keep putting pi everywhere so we remember that. Pi k plus 1.

The expected value just is a sum over probabilities of getting each of the outcomes. So you can use that transition matrix. And in vector form, since I have a finite number of discrete states, I can just write that as J is g plus gamma TJ, where the i-th element of g is Keep my pi's everywhere. Everybody agree with those steps? OK. So what's J-- J pi?

AUDIENCE: [INAUDIBLE]

RUSS TEDRAKE: Mm-hmm.

AUDIENCE: [INAUDIBLE]

RUSS TEDRAKE: I have to go to a vector form for J, so I just put it over here. I'm saying that the i-th element of the vector g-- this is my vector g now-- and the i-th element of my vector g has that T in there-- absolutely. Yep. So it's the expected value of g there. OK, so what's J pi?

So lo and behold, policy evaluation on a Markov chain with known probabilities is trivial. It's this. It's almost free to compute. I could tell you exactly what my long-term cost is going to be just by knowing my transition matrix. That's something I think we forget, because we're going to get into models that look more complicated than that, but remember, if the transition matrix, it's trivial to compute the long-term cost for a Markov chain.

So let me just show you why that's relevant, for instance. All right, so I told you about this the day the clock stopped. I kept telling you about it [INAUDIBLE] And for the record, do you know what happened that day? The clock physically stopped.

Michael debugged it. There was a little piece of paint that blocked it at exactly 3:05 the day I was giving that lecture. That was a hard one to catch, to be fair. So one of my favorite models of stochastic processes in discrete time, for instance, is taking our rimless wheels, our passive walking models, and putting them on rough terrain.

So this is the rimless wheel, where now, every time it takes a step, the ramp angle is drawn from some distribution. Now, in real life, maybe you don't roll rimless wheels on that kind of slope, but the contention in that paper was that actually, every floor is rough terrain, and you actually have to worry about the stochastic dynamics all the time.

And if you want to take-- you can take your compass-gait model and put it on rough terrain, and you could take the [? kneed ?] model and put it on rough terrain. These are the passive things, so they can't walk on very much rough terrain before they fall down. But they can. They can walk on rough terrain.

And then you want to ask complicated questions about this, maybe. You want to say, given my terrain was drawn from some distribution, how far should I expect my robot to walk before it falls down? That sounds like a hard question to answer. It's trivial to answer, actually.

So this equation is exactly what drove that work. We built the transition matrix on the [INAUDIBLE] map, saying, given it's passive-- there's no actions to choose from-- given it's passive, what's the probability of being in this new state, given the terrain's drawn from some distribution given it's at a current state.

The cost function was 1 if it keeps taking a step, 0 if it fell over. And you compute this, and it-- what does it tells you? It tells you the expected number of steps until you fall down-- period. On shot. Simple calculation. It's so simple. The bad part is you have to discretize your state space to do it. But if you're willing to discretize your state space, then you can make very long-term predictions about your model with-- just like that, to the point where we are trying to say that people who talk about stability-- people are coming up with metrics for stability and walking systems. They say, why not just do this?

Why not actually compute, given some model of the terrain, how many steps you'd expect to take until it falls down? That's what you'd like to compute, and it's not hard to compute, so you should do that. So that's a clear place where policy evaluation by itself-- there's lots of cases where you have a robot that's doing something, it's got a control system, and you just want to verify how well it works. If you're trying to verify it an expected value, it's easy. Just do the Monte Carlo-- or sorry-- the Markov chain thing.

But what happens if I don't have a model? That's what we're supposed to be talking about today. Can we do the same thing if we don't have a model? I had to know T. I had to know the-- all the transition probabilities in order to make that calculation. What happens if we don't have a model-- we just have a robot we can run a bunch of times? How do you do it?

What would you do, if I asked you-- I say, I like your robot. I want to know how long it tends to run before it fails. How would you do it? How would you do it? There's an easy answer. You could run it a bunch of times and take an average.

We know that these value functions are state-dependent, so it's a little more painful than that. Technically, you're going to have to run it a bunch of times from every single initial condition, but you could do that. And actually, that's not totally crazy.

So I want to know how much cost I'm going to incur-- in the case of the walking robot, how many steps it's going to take on average before it falls down. First thing to try-- don't have to know the transition matrices-- just run it a bunch of times. So if I say Jn i, the n-th time I run my robot, I incur-- I just keep track of the cost. I keep track of how many steps it took. I keep track of how much gold it found-- whatever your cost function is.

The thing I'm trying to estimate is the expected value of that long-term cost. But any one trial-- I get this thing out as a random variable. I could take the expected value of the random variable. I can make a nice estimate of J pi i by just running it a bunch of times and taking-- doesn't sound very elegant, but it works.

AUDIENCE: [INAUDIBLE]

RUSS TEDRAKE: What? Sum over k. Good. Thank you, thank you. Good. Sum over k. [INAUDIBLE] you've corrected both simultaneously. OK, so a couple of nuances here-- so first of all, I have an infinite horizon cost function. So this is only going to be an approximation, because I'm not going to run this forever 10 times. I'm going to run it for some finite duration 10 times.

So in practice, I'm actually going to run something that's big number. But that's OK because this discount factor means that a finite trial approximation should be a pretty good estimate of the long-term. And if I run it from initial condition i long enough, then I should be able to take an average and get the expected [INAUDIBLE].

There's lots of ways you can do that kind of thing if you don't want to do all the bookkeeping of remembering where you've been. You don't have to remember all of these things. You can do an online version, incremental. You can say that my J hat is just-- my J hat pi is just J hat pi-- I can guess an initial J hat, and then, every time I get a new trial, I'll just move my estimate towards that trial.

And this is actually an online version that approximates that in batch. This is just a standard [INAUDIBLE] that you can do it more carefully. I could choose these to be a perfect weighting but in general, this is actually a pretty good approximation, as the number of trials goes up, to this sum without keeping track of every J.

Every time, I'm just going to do moving average towards the new point, and by changing a small amount, it will converge. This is a low-pass filter. That's another way to say it. It's a low-pass filter that tries to get me to-- the mean of the J samples I'm getting in. So that gets rid of a little bit of bookkeeping. There's other things you can do.

Now, here's a really cool one. Think about this, and tell me if you think it's possible. I'm going to tell you in a minute how that-- if I have two policies, I can-- say, pi 1 and pi 2-- Do you believe that? It's going to take a little bit more machinery, but just to see where we go.

Say I have two control systems. I have the one that is risky, and I ran it once, and the thing fell down. So I don't actually want to run that 100 times. I might break my robot. Let's say I've got a different policy that I like a little better. It's a little safer to do evaluations on. Can you imagine running the safe policy, let's say, to learn about the risky policy? That's pretty cool idea, right?

What is wrong with this? Typically done with a q function. I'll show you how to say that in a second. So there's lots of ways you can do that. You can run trials, you can keep averages, you can try to learn about one trial by learning the other. What the fundamental idea here is is that it requires stochasticity. You need that, in policy, pi 1 and pi 2 have to change-- take the same actions with some non-zero probability.

Pi 2 might be my risky policy, and every once in a while, with some small probability, it takes a safe action, let's say. And pi 1 is my safe policy, but every once in a while, it takes a risky action. As long as these things have some non-zero overlap in probability space, then I can actually learn about what it would have been to do the more risky thing by taking the more conservative thing.

So policy evaluation's a really nice tool. But this feels slow. The Monte Carlo thing feels slow-- feels like I got to run a lot of trials from a lot of different initial conditions. And now you tell me what the cost-to-go is from this initial condition, and let's say I try this initial condition. What do I do? Do I just have to start over and run trials from the get-go again? Well, that doesn't seem very satisfying. Approach number two is bootstrapping.

I call it bootstrapping. If I learned about the cost of being in this state, and I spent a long time learning about the cost-to-go of being in this state, and then I go back and ask what's the cost of being in this state, if this one transitions into this one, then I should be able to reuse what I learned about the state to make it faster to learn about that state.

I didn't really plan to do it with the steps on the floor, but I hope that makes sense. Maybe I could do it on a graph. That's better, yeah? Let's say I figured out what J pi of this state is-- because I went from here, and I went around, and I did my stuff, and I learned pretty much what there is to learn about here [INAUDIBLE] policy.

And now I want to know about this state. Well, I should be able to reuse the fact that I've learned about that to help me learn this more quickly-- reasonable idea. Using your estimate to inform your future estimates is an idea about bootstrapping, reusing-- building on your current guess to build a better future guess.

And here's how it could look in the optimal control policy evaluation sense. What if I said my online rule used to be this, where I've got some estimate J pi hat? I'm going to run from 0 to some very large number to estimate this, and then make the update. What if, instead, I just took a single step and I did this update?

Does that make sense to you? Let's say I ask you to guess the long-term cost here. Instead of running all the way to the end, what if I just run a single step and then use as my cost my estimate for this, the cost of going here plus the gamma times the cost of doing all that? It's just using this one-step cost as an estimate for when I was going J-N of ik plus 1-- or sorry, J-N of ik.

Does that makes sense? If I find myself in a lot of different initial conditions, I could take one step and then use my guess for the cost-to-go from that step to the rest of the time. Now, this starts feeling a lot more appealing, actually, because now I don't have to think-- this actually got rid of that whole episodic problem.

I don't have to go in and run some fixed length trial to approximate the long-term thing. I just take a single step, use this as my estimate, and I can just keep moving through my Markov chain. I don't have to ever reset. And potentially, if I visit states often enough-- I won't get into all the details-- roughly, it involves that Markov chain being-- having ergodicity. you have to be able to visit all the states with some non-zero probability as you go along.

But if you visit the states-- each state infinitely often is roughly the thing-- then this actually will converge to J pi of ik. So the ergodicity is actually bad news for my walking robot, because if my walking robot falls down, I'm going have to pick it back up if I want to get ergodicity back. There are robots that don't visit every state every arbitrarily often.

But in the Markov chain sense, that doesn't seem like such a bad assumption. And if I'm willing to take my robot when it falls down and pick it back up-- which, by the way, is about how I spent the last year of my PhD-- then actually, I can get ergodicity back.

OK, cool-- so that makes sense, right? I'm going to use my existing estimate of the cost-to-go to bootstrap my algorithm for estimating the cost-to-go. Yeah?

AUDIENCE: Does the transition [INAUDIBLE] come into play at all?

RUSS TEDRAKE: It does, because I'm getting this from sampled data. So this is actually drawn. The expected value of this update does the right thing. So this update doesn't have it, because this is from a real trials. But you should think about this as a sample from the real distribution. Now, that's actually a really good way for me to lead into the next step.

These algorithms tend to be a lot faster in practice than those algorithms-- not only are they a little bit more elegant, because you don't have to reset and run finite-length trials-- they tend to be a lot faster. And the reason for that is this here is really the-- it has the expected value of future costs built into it.

Let me say that in the pictures. There's two ways, I could estimate this. I could get here and then I could take a single path. Well, this one is not rich enough for me to make my point here, but OK-- so I could take a single path through here and get a single sample estimating the long-term cost. But if I instead use J pi, J pi is the expected value of going around and living in this.

So by using this update to bootstrap, or if I just take one step from here, I get for free the expected value of living over here for a long time. Does that make sense? So J is building up a map of the expected value, because it's visiting things often and it's-- drew this online algorithm with this low-pass filter. He's basically doing an expected value calculation.

By using my low-pass filtered [INAUDIBLE] in here, it's also-- it's getting the reward of-- maybe you could just say it's filtering faster. That's actually not a bad way to think about it. I've already got this thing filtered, so this one filters faster. That's a pretty reasonable way to think about it, actually. OK.

So this quantity here in the brackets, this whole guy right here, it's a very important quantity, it comes up a lot. It's called the temporal difference error. It's the difference that I get from executing my policy for one step and then using the long-term estimate compared to what I have is my long-term estimate, temporal difference error.

Now, if the system was deterministic and I had already converged, then that temporal difference there should be 0, because this thing should be exactly-- predict the long-term thing. If the system's stochastic, then this temporal difference error should be 0, on average. It's comparing my cost-to-go from ik, given my 1 step plus the cost-to-go from ik plus 1.

So those things-- you want those to match, right? You want that my 1 step plus long-term prediction should match my long-term prediction, if things are right. They should match an expected value. So that thing's called the temporal difference error, and it's an important quantity in reinforcement learning.

It makes sense to write that down. That's a reasonable estimate for J. n would be to take one step and then do the other one, but there's-- that seems a little bit arbitrary. Why don't I just do one step and then use my lookup? Thi sis the way Rich says it. Why not do two steps and then use my value function to look up? Or three steps-- why not take, similarly, three steps, and then use that to look it up-- or 14 steps or something like that.

Why should I arbitrarily pick this one real data and then look ahead, instead of two real pieces of data and my look ahead? Well, there's no reason that you actually have to pick like that. Can I just write the inside part here? I could have estimated Jn ik as gik ik plus 1 plus gamma gik plus 1 ik plus 2 plus gamma squared J pi at ik plus 2. That's a perfectly good approximation too. I could have done three-step. I could have done four-step.

AUDIENCE: [INAUDIBLE]

RUSS TEDRAKE: Say it again.

AUDIENCE: [INAUDIBLE]

RUSS TEDRAKE: Yeah I. I haven't been writing it that way because-- yeah. I would have been fine writing it that way. At some point, I decided to not write pi there, and I'll just stay consistent by not writing that. OK, so Rich [INAUDIBLE] came up with a clever algorithm that's-- basically allows you to seamlessly pick between the one step, two step, three step, n-step look ahead with a single knob. And it works. It's called the TD lambda algorithm.

And the basic idea is that you want to combine a lot of these different updates into a single update. It sounds really bizarre [INAUDIBLE] so let me just say it. Let's say I call my estimate J of Jn, with M step look ahead of ik. M equals 0 to M gamma of M gik plus M ik plus M plus 1. Big M.

This is a big M. Big M. Everything else is little m's. That was the one step. This is the two step. In general, this is the M step look ahead. So it turns out there's actually an efficient way to compute this.

Let's call it something else-- p, p, p. This one takes a little time to digest. But it turns out it's pretty efficient to calculate a weighted sum of the one step, two step, three step-- onto forever-- sum of look-aheads parameterized by another parameter, lambda. So when lambda's 1, this thing turns out to be basically doing Monte Carlo. And when lambda's 0, this thing basically is doing just the one-step look ahead. And when lambda is somewhere in between, it's doing some look ahead using a few steps. Does that makes sense at all? It's a lot of terms flying around here.

Even if you don't completely love that, just think of my estimate, J of lambda being awaited, basically something that, where if lambda is 1, it's going to be the very long-term look ahead. If lambda is 0, it's going to be the very short-term look ahead. And there's a continuum in between, a continuous knob I can turn to say how far I'm going to look ahead into the future as my estimate that I'm going to use in that TD error. And there's a whole gamut in between.

AUDIENCE: [INAUDIBLE]

RUSS TEDRAKE: Can I say it again?

AUDIENCE: [INAUDIBLE]

RUSS TEDRAKE: Or can I read it? 1 minus lambda i just a the normalization factor. p equals 1 to infinity. Lambda the p minus 1 J p-- where this is the p step look ahead. So this is a very famous algorithm-- the TD lambda algorithm-- which allows you to do policy evaluation without knowing the transition matrix, doing bootstrapping or Monte Carlo in a simple single framework with just a parameter lambda to evaluate. So it's a tweak.

And it turns out it uses an eligibility trace, just like in reinforce. Did you get the eligibility traces, John?

AUDIENCE: [INAUDIBLE]

RUSS TEDRAKE: OK. Well, that's fine. So it turns out to have a really simple form. I'll write it, because it's so simple, but it'll also be in the notes, if you want to spend more time with it here.

OK, two observations-- first of all, this looks no harder [INAUDIBLE] than the original version I had, pretty much. It just requires one extra variable, which is this eligibility trace. What does the eligibility trace look like?

OK. It starts off at 0. There's an element for every note in the graph. Every time I visit the graph-- that node in the graph, I-- it goes up by 1, and then it starts forgetting based on gamma and lambda, as this discount factor. And then, the next time I visit it, it goes up by 1. If I visit it a lot, it can build up like this.

It's just a trace of memory of when I visited this cell. Does that makes sense, this dynamics here? Every time I visit the cell, it goes up by 1, and always, it's going down exponentially. It turns out, if you just remember that, the way that you've visited cells in the past, decade by this lambda-- as well as gamma-- but this lambda-- which is the new term-- then it's enough to [INAUDIBLE] this trivial update here, scaled by the-- how often I visited that cell recently.

Is it enough to accomplish this seemingly bizarre combination of short and long-term look-aheads. So it's a really simple, really beautiful algorithm. Just remember how-- when I visited these cells, and then make this TD error update scaled by that, and I've got the TD lambda algorithm.

And what people can do is they-- people can prove that TD lambda converges to the TD lambda update that J hat will go to J pi from any initial conditions. So you can just guess J randomly to begin with. And if I run it, as I visit all these states arbitrarily often, it still makes that ergodicity assumption.

Then I'll get my policy evaluation out. That's really cool-- simple algorithm. Now, what people also realize is that, when you start out, and J is randomly initialized, then it makes a lot of sense to set lambda close to 1, because bootstrapping has less value when I just start out. My estimate is bad everywhere, so why should I use my bad estimate as my predictor?

So you start off-- you keep lambda close to 1. It does very long-term. It does more Monte Carlo style updates. And as this estimate starts converging to the good estimate, you start turning lambda down. And with a cleverly tuned timing of lambda, you can get very fast convergence compared to the Monte Carlo algorithms. You more and more bootstrap.

Excellent. Well, time's up. The clock is still moving today, so I have to stop. So the really cool thing-- we only talked about policy evaluation today. The next step is, how do you do these value methods to improve your policy?

And it turns, out in many cases, if you make a current estimate of your value function and then, on every step, you try to do the greedy policy, epsilon greedy policy, you basically-- you mostly exploit your current estimate of the value function, then you can still prove that these things-- at least on the grid, the Markov chain case-- can get to their optimal-- the optimal value function and the optimal policy.

So we'll finish that up next time and get into the more interesting-- get rid of these Markov chains to try to get back to the real world. Good. OK, see you Tuesday.