MITOCW | Lecture 20 | MIT 6.832 Underactuated Robotics, Spring 2009

RUSS TEDRAKE: OK. So welcome back. We talked last time about the problem of policy evaluation, which was, given I'm executing some policy pi, estimate the cost to go, right? And we showed that it was sort of trivial to do if you have a model. And if you don't have a model, you can still do it with the temporal difference learning class of algorithms, which is TD, which is in the title there, OK?

And the temporal difference learning, the TD, lambda, in particular, was nice because it encapsulated all the algorithms we talked about last time. TD, if lambda is 0 was essentially the one-step bootstrapping idea, we said, where you use your current cost plus your expected value of all future costs as your new estimate of this. And the other limiting case was TD1, which resulted in just Monte Carlo evaluation.

And what I said quickly at the end of class is that-- we spent all our time last time on doing temporal distance learning just on the representation of solving a Markov chain, right? We did it for discrete state, discrete action, discrete time. And it's known that these algorithms converge to the correct estimate if you run enough times and your Markov chain is ergodic, if you visit all states at all times, OK? But that's pretty limiting because in order to do a Markov chain analysis of even an acrobot or something, you have to democratize four state variables-- theta, theta dot, theta 1, theta 2, theta 1 dot, theta 2 dot. So you do 25-- I mean, things get big fast. It'd be the 25 to the 4th power if you put 25 bins in each dimension, and that's not very many.

So today, we want to make the tools more relevant, more powerful, by breaking this assumption that we're in a Markov chain, where we've discretized everything, and try to do more general-- try to do temporal difference learning on a more general class of functions that are more natively continuous, OK? So we're going to do it with function approximation.

Now, to get there-- John did a little bit of function approximation in the reinforce lectures, so I want to basically pick up on the kind of examples he was showing you, and do a quick crash course on least squares function approximation, just to make sure that people are comfortable with that, and we'll build on that quickly.

OK, so let's talk a little bit about least squares function approximation. OK, so the canonical example, which is the same thing that John showed you in class, is we want to approximate, or we want to estimate, some unknown function that takes input x and spits out output y. And you're given input/output data, which we can write something like-- you're given pairs of data, samples of the input and output. So you're allowed to query the function. Given this input, what output should you do?

In the basic case, you're doing this passively. Someone just gives you a data set, and you're supposed to then do your best job at reconstructing F, OK? There are interesting cases that people look at where, if you're allowed to choose this, then how do you actively interrogate the system? How do you pick the x's to get the most information output? In the simple case, let's just say you're given a collection of input/output data, and you want to estimate F.

So the standard way to do this is to write down-- there are many cost functions you could use, but the one most people use is to write down a least squares cost function, where we're going to try to find a model where y hat is some F of, let's say, it depends on parameter vector alpha, hat of x. Just like in the policy gradient kind of case, I'm going to write down a set of functions where the actual function depends on both alpha and x, but we can think of it as taking x's input and generating an estimate y hat, OK?

And you can formulate the problem with the least squares metric, where you're going to minimize, over alpha, the squared error over my data minus my estimator. I actually have the estimator here. No big deal, but just to keep it consistent with my chicken scratch notes. OK?

So if we can write down a class of functions we want to search over, then we can turn it into the standard optimization we've been doing throughout, where we just try to find the parameter vector alpha, which makes our estimates as close as possible, in the least squares sense, to the actual data, yeah? Or y hat here is F hat alpha xi.

OK, so why do we care about doing that? The methods used-- you already know optimization methods. We know a lot of ways that we could potentially try to solve this. So for instance, we could take the gradient of this with respect to alpha and do gradient descent, which is exactly what John was doing in the reinforce case, except the gradient was not calculated explicitly. It was estimated by sampling in the reinforce case, right?

And in general, there are some cases-- there are some classes of functions that we can choose, and I'll make this more graphical and explicit in a second, where you can just solve it analytically, right? Same way in the optimal control derivations, there were some u's that we could just derive analytically and some that we had to do gradient descent for.

So let's dig in now and get more specific and say, what kind of models do we want to use as candidates for these functions we're trying to fit? How would we write down an F hat, which depends on both alpha and x? The literature is just filled with people's different models that they do this. One of the most popular ones of, let's say, 20 years ago, which actually are still popular today, and we call them function classes. One of them is neural networks.

So a lot of people believe that a good way to write down an arbitrary function is to take your inputs and try to do something that models a neuron. It adds things up. Adds up the different inputs, and then goes through a sigmoidal function and gives you an output y. Maybe just because that's what the brain sort of does, maybe because there's a lot of success stories from it, but a lot of people do this, right? And moreover, this is sort of the single layer neural network if you have lots of different input functions. And they potentially add up to y, for instance. And the parameters now are the weights of this connection.

So your function might be some squashing function, this nonlinear function, times a weighted combination of the inputs, potentially plus some baseline or something like this, where this is tanh or some function like that that does a sigmoid.

OK. And if you do this, then you can stack things up and make multiple-layer neural networks. And people believe, and people certainly in the '80s very strongly believed, that this is a representative class of functions that maybe looked a little bit like what was going on in the brain.

And people know that if I have-- it's a general function approximator. If I have enough of these neurons and enough layers, then I can represent any function arbitrarily finally.

So it's kind of a cool result state. Using this thing that looks like the brain, if I stack up these elements, then I can represent any function with enough neurons. And if I want to solve a least squares optimization problem to make the input and output of this neural network work the same, then I can just solve this optimization problem with gradient descent, and that's roughly what everybody did in the '80s. I guess that's not quite fair, but-- right?

And it works, actually. People still use these today to do face recognizers, right? They use it to-- Yann LeCun down at NYU has got this text recognizer that they use actually to scan-- to read checks at the bank, right? That's still neural network-based, OK? A lot of people got a lot of things to work with these neural networks. Gerry Tesaro, in the optimal control sense, got the TD gammon-- the reinforcement learning system to play gammon, where the board configurations were put in as inputs to the neural network, and the output was what move you should take, and he got that to work, right? And actually, there was a value function that came out. He would even explicitly estimate the value function.

But today, this is a strawman because I don't like neural networks. They're not a particularly nice least squares thing to optimize. We can do better these days. I just wanted to put it up there is an important element of the class, but not one we'll tend to use, OK?

If you care about least squares function approximation, there are a lot of other choices you can have for your function f, which you can parameterized by alpha, which maps x to y. A lot of choices, OK? The ones that people in reinforcement learning tend to use most effectively are the linear function approximators. Linear here, meaning that they're linear in the parameters, but not necessarily linear in the outputs.

So I want yi hat to be some linear combination of potentially non-linear basis functions of xi. OK. So the neural networks had the problem that they were rich because they have nonlinearities in them, and if you cascade nonlinearities, you can get arbitrarily rich things. But the parameters that change the function are buried deep inside the nonlinearities, OK? It turns out, if you come up with policy and function classes that can represent your problem nicely, where the parameters are all at the output, then life gets a lot better, OK?

Why does life get a lot better? Well, because now I can take this least squares metric and solve it explicitly for alpha, OK? There's two things. I can represent arbitrary non-linear functions if I have the right basis set. Also, compute alpha explicitly. Don't have to do gradient descent. I can just find the optimum.

Just to see how that goes-- you can probably see it, but it's so simple, we'll write it out here. If I want to minimize that squared error, I minimize over alpha, sum over i of yi minus alpha transpose. I'll write the vector form of that v of xi squared.

I could write this even more vector form if I choose big Y to be y-- if I put these things into a big vector, and big phi is in the other direction, then I can write this guy as min over alpha y minus phi alpha. If you take the gradient with respect to alpha, what do you get? You get negative phi transpose y minus phi alpha equals 0.

Assuming that this thing is well-behaved, then you can just say alpha is-- very straightforward least squares estimation with linear approximators, but let me now convince you just how rich that class of functions is, OK?

OK. Here's the game. There's some function that exists. This is my actual f, OK? I don't know it. My algorithm that I'm about to do doesn't know it, but that's the original f I'm trying to estimate, OK? And let's say I don't get to experience f, but I get to sample from f, and every time I sample from f, I get some noise, right? Then maybe my input/output data might look something like this. This is just-- I took a bunch of random x's, I evaluated what that function was, and I added a little bit of random noise to it, OK?

Now the question is, if I just have those red data points as input/output data, can I come up with an estimate in f hat, which reproduces the original f? And that's not many data points, right?

OK. I could do it with a neural network, but without telling you all the details, it's not quite as elegant. We did it with reinforce. It took a little bit of convergence. If we do it with a linear function approximator, we can do it just in one shot, just like that.

The first trick, though, is we have to pick our basis set, phi, OK? You've got lots of choices with phi. Some of the common ones-- let me do-- one common one is radial basis functions, where you assume phi of x is just some Gaussian, right? phi i of x is just a Gaussian function. The normalization doesn't matter. x minus mi squared to sigma i. It's just a collection of Gaussians where the means are different for every basis function, OK? The variances could be different too. That doesn't matter.

So that might look like this. If I made 10 different basis functions for this problem, and I centered them evenly across the space that I sampled for, then that would look like a reasonable basis set for function approximation. OK. And then the question is, can I take a linear combination of that 10 Gaussians and turn it into a pretty good estimate of the original function just from looking at the red points? OK?

If I plug this equation in, then I can get a weighting on each of those individual phis. Alpha i is the weight on phi i, right? I'll do that with another click here. And it turns out it said, in order to represent this function, that one that was centered around 0 had better have some negative value. The one that was centered around 1 has got a pretty big positive value. This one's got a pretty big positive value and so on. So you can see all the same Gaussians are there. They're just weighted by a different amount, OK?

And if you sum all those up to estimate y, then what do you get? Pretty good, right? It's pretty sparse set of data points. Pretty sparse set of basis functions gets a really nice accurate-- in one shot. No gradient descent or anything. This is consistent with what John and I have been saying, don't do reinforce unless you have to. Because if you know the function and you can sample from the function, you can just explicitly get it.

OK. The barycentric interpolators that we talked about before-- remember, we interpolated between elements of the grid with barycentric interpolators. Those are linear function approximators, where it turns out, you can think of that as having nonlinear basis functions, which have something like this, the whole thing rectified.

Essentially, if I plot it in 2D here, then-- if I turn my radial basis functions off and do my barycentric interpolators, and I run that same demo, the barycentric interpolators are going to look like that, OK?

If I want to linearly interpolate between all these neighboring points, one way to view it is I take my distance between the points, I interpolate. Another way to view that is actually, those are basis functions that look like tents. And the same thing is true in 2D, they're 2D tents, OK?

It's the opposite way to think about it, but those barycentric interpolators we've been using the whole time are exactly linear basis functions. And again, I can sum these guys up, and I can make an approximation of the original non-linear function. This one, of course, has got to be piecewise linear, but that's OK. It did a pretty good job, considering it's piecewise linear and it's not a piecewise linear thing it's estimating.

OK, you could do basis functions based on Fourier decompositions, for instance. You could do polynomials basis functions, where phi i of x is x to the i minus 1, let's say. Something like that. All these things work.

I'll do the Fourier one here. Same function, noisy data. There's a Fourier basis over-- spatial Fourier basis. I can add those up. I get very large coefficients. These tend to cancel themselves out a lot, but still, I get a pretty nice representation. OK.

So this idea of using linear function approximators and then linear least square-- exact least squares solution is a very powerful idea. You can represent very complicated functions potentially with this. This was not the way people tended to do things 15, 20 years ago. It really tends to be the way people do things now.

In fact, machine learning got on this kick in statistical learning theory. People talk about kernel methods, and you might know these. I mean, the essential idea is there's two problems. I didn't say it, but sometimes people are trying to estimate scalar-- continuous-valued outputs. Sometimes, people are trying to do Boolean outputs. They would call that a classification problem. There's different forms of this problem.

But in machine learning, people realized that you can often take a fairly low-dimensional problem, blow it up into a very high-dimensional space using lots of basis functions, for instance, lots of kernel functions, and then do linear least squares in the high-dimensional space, and that works a lot better than doing some sort of nonlinear gradient descent-based estimation in the low-dimensional space, OK? So that's really a trend that happened in machine learning.

In reinforcement learning, it's the norm because essentially, when we have linear function approximators, we have some proofs that our optimal control algorithms are going to converge when we have almost nothing. In fact, in the more nonlinear function approximator case, we have lots of examples where things like TD on a function approximator just don't converge. There are simple examples where they do exactly the wrong thing.

Good. So just to convince you of one more basis function that is more relevant to this class-- that was just the crash course for people who haven't seen function approximators-- let me give you another example from system identification. Let's do nonlinear system identification as a least squares problem with linear function approximation.

So let's say we've got our equations of motion, which come from this. It's been a little while since I wrote these equations. Jeez. Let's say it's for the pendulum or for the acrobot, you name it. And let's say we know the form of the equation, but we don't know the parameters. We don't know how much the masses are, how long the link lengths are. Normally you can measure those. But inertias, things like that, these can be harder to get right.

So we can, for instance, run a bunch of trials with some dumb open loop controller. Just pick you randomly, let's say, and make the acrobot flail around a little bit and collect data that looked like this, right? q, q dot, even q double dot, and u at every instant in time. And this is going to be exactly like our input/output data in our least squares formulation, OK?

And here's the very amazing observation. This one really surprised me when I first got it. The manipulator equations for random robot arms, they're actually linear in their parameters. Very non-linear functions. They do very complicated dynamics. But they tend to be, for most robot manipulators-- robotic arms on the factory floor, walking robots, things like that-- those equations actually turn out to be linear in the parameters. They're not linear in q, but they're linear in the parameters.

All right. Take the simple pendulum. I can rewrite this dynamics as alpha transpose times theta double dot, theta dot, sine of theta equals u, where alpha is i, b, and mgl.

OK, after you think about it a little bit, it turns out to be not all that surprising. In all our robot manipulators, you see sine thetas everywhere. You see sine squared thetas. You see sine cosines. You see all these things. You never see sine l theta or something like that, OK? It turns out, in these problems, that the parameters don't end up inside your nonlinearities. Yeah.

AUDIENCE: Isn't it still nonlinear because the parameters are multiplied together?

RUSS TEDRAKE: OK, good. So it's not linear in every individual parameter, but I can get-- I can re-estimate this as a linear optimization. So that's exactly right. So sometimes, you have to settle for groups of parameters. But those groups of parameters are always enough to rewrite your initial dynamics, OK?

OK, so that actually makes it sound like sys ID is easy. If we have a complicated robot-- yeah, says Michael, who's been doing it for the last three months, six months, maybe. I don't know. (LAUGHS) You just shot daggers at me there.

Yeah, so sys ID should be easy for robots that are well-behaved. It turns out, if you have saturations in your motor and stuff like that, it gets more complicated. Michael could tell you all about it. But if I have a simulation of a pendulum, let's say, then it should be trivial, and it is trivial. So let me just show you that.

It turns out, it's exactly the same linear function approximation. This is my basis functions, right? These are my phi's. I've got three basis functions. One is theta double dot, one is theta dot, one is sine theta. These are my coefficients, alpha.

And how am I going to do it here? Where is it? Sys ID, yeah. Just a few lines there of the Matlab code, and that includes running the tests, OK? So most of this is just setting up my simulator.

I'm going to pick some small random actions, a random tape of torques to play out. I'm going to pick some random initial condition. I'm going to run my control system with just making that tape-- a zero-order hold on that tape, OK? And I'm going to collect the time x torque and x dot that came out of that simulation, and do exactly the math I showed you over here, where alpha is i, b, mgl-- in this case, it's the location of the center of mass-- and do my optimization like that and watch what happens. What's it called?

OK, so I started from random initial conditions. I play a small random tape for 10 seconds out, OK? The actual parameters that I started with were this. The ones that are estimated after 10 seconds of running my simulation and doing least squares are that. It's pretty good, right? And that was corrupted by noise. Not a lot of noise, I guess, but noise. It's easy. System identification's easy, right?

It's actually a very, very powerful observation that you can do system identification for these really hard systems just as a linear function approximation task. One shot. That's what makes adaptive control tick on a lot of these manipulators, right? It's a very fundamental observation.

AUDIENCE: What would happen in the case you assigned theta to a theta?

RUSS TEDRAKE: OK, it's not going to work as well, but let's-- it's running some random initial tape here. It's not catastrophically bad, actually.

AUDIENCE: We have this small angle.

RUSS TEDRAKE: Exactly. That one happened to be a small angle, right? Let's see if I get-- come on. That's a bigger one. That's worse, yeah?

There's another way I can break it, just to anticipate here. Let me put it back before I forget. This was sine here, right? What if I don't put enough control torque in, OK? I put a note to myself, if I make a 0.1 here, then suddenly, I'm not putting in very much control torque. And why could that be a problem?

AUDIENCE: Unprotected.

RUSS TEDRAKE: What's that?

AUDIENCE: You're unprotected. [INAUDIBLE]

RUSS TEDRAKE: Yeah, I'm sort of not-- exactly. I'm not simulating the system beyond the noise that I've added, and that can break things. Let's see. So now it's pretty much just falling almost as a passive pendulum, and that breaks things more. Although this is a pretty easy problem. That doesn't horribly break anything.

OK, that same code could have run for the acrobot, right? It couldn't have run for one of our airplanes because the aerodynamics tends to not be linear in the parameters. But rigid body dynamics tend to be linear in the parameters, right?

Doing it on the acrobot's a little bit harder because you have to be more clever in stimulating all the dynamics with your one actuator. So there are a lot of good problems left in system identification. Designing sufficiently rich inputs to stimulate all your dynamics is one of the big ones. But function approximation and least squares is basically what you need to do to do system identification.

AUDIENCE: So if you have, for example, [INAUDIBLE]--

RUSS TEDRAKE: Yeah.

AUDIENCE: And then presumably, you have sine theta minus as one of the parts in your equations. So if you want to get something good, you, should put that as one of the features, right? You have sine alpha minus gamma there, and then [INAUDIBLE].

RUSS TEDRAKE: There's always a step where you have to look at your equations and pull out the proper basis function to describe that class of systems. Absolutely. So if there's a sine theta 1 minus theta 2 floating around in there, it should be in one of your basis-- as one of your basis elements.

AUDIENCE: If you want to do this with this system but you don't have the knowledge, you think that it's linear, but you don't know the equation for it.

RUSS TEDRAKE: Good.

AUDIENCE: Then?

RUSS TEDRAKE: Then maybe you should do radial basis functions or polynomials or some other basis set. I think a more common thing would be, imagine there's something I'm not modeling well in my pendulum. Let's say there's some nonlinear friction in the joints or something like that. A common thing that people would do would be to add in here some slop terms-- let's say radial basis functions or something, just in my standard linear function approximator-- and do that, OK?

And then, now you've just added more representational power to this-- you've given the basis functions which you believe to be true, but you also add in some slop terms, right? And this is-- I mean, so Slotine certainly teaches this and does this in his robots, right? For his adaptive controller, he puts those in to capture the terms that he didn't expect to show up. Yeah.

AUDIENCE: How well does this work for things that aren't really smooth. Sometimes stick slope can-- it seems like if you plot it, it's not really a continuous function.

RUSS TEDRAKE: Continuous doesn't matter, right? If you were trying to fit a continuous basis set to a discontinuous function, then it'll only do as well as it can in the least square sense. But you can represent arbitrary static functions of-- if it's a function of x, then it should be right. I think the more common problem, maybe in a stick slope kind of model, is that there's hidden state--

AUDIENCE: OK, yeah.

RUSS TEDRAKE: --for instance, right? Maybe you don't know exactly what the state of the system is because there's some other-- and then you'd have to estimate that to put it into your basis set. OK, people feel OK with least squares estimation? Yeah?

Good. Now let's see if we can put it to use to do what we promised at the beginning, which is temporal difference learning. It's gone, isn't it? OK. So if you remember, and I hope the impression came across when I was talking about temporal difference learning, that we made a pretty complicated update, which was this weighted sum of one step, two step, three step, end step returns through some algebraic trick, and that's really probably the right way to think about it. Through some algebraic trick, you could do it with a very simple algorithm. So let me just remind you that that algorithm was--

OK. So if you remember, the picture we had last time was that I've got some Markov chain that I'm moving around. As I'm walking around this Markov chain, I'm trying to-- every time I visit a state, I want to update my estimate of the costs to go from being in that state, right? And I can do it with this very simple algorithm which keeps a decaying eligibility trace on each node. Every time I visit this node, it goes up. And then it starts decaying until the next time I visit and it goes up.

The rate at which it decays is given by the discount factor in my optimal control problem and the lambda, which is the amount that I want to weight Monte Carlo-- long-term evaluations-- versus short-term evaluations, right? If lambda is 1, I'm going to let that settle much more slowly, and it's going to be making very long-term updates, and if lambda is 0, then it's just going to make an update based on the one-step prediction, OK?

If I carry around this eligibility trace, this would be ei as a function of time for every i, and I do this very simple update, then there's this interpretation that j hat is doing something between Monte Carlo and one-step bootstrapping, depending on what lambda you pick. Right?

OK, so let's say we don't have a discrete state and action system, but we now have, let's say our barycentric interpolator is on a pendulum or something like that, right? And let's say I take a trajectory through here, which I-- that was a bad trajectory for a phase space, but I take some trajectory through my state space, OK? Let's say I'm willing to discretize the system in time still. And let's say my value estimate is a linear function approximator here, which is this barycentric grid, OK?

So at every point here, I'm going to say j hat is just a weighted combination of the four neighboring points weighted by the distance, right? Like I said, that actually is exactly of the form where I've got a scalar output, and I've just got-- you can think of this as being a tent, this being a basis function which has sort of a tent of region of applicability right here, and added with this one that has a tent of applicability here. And at every point, there's a-- at every cross-hair, there's a basis function that looks like a tent. Did you get that out of the previous one-- the previous picture? Linearly interpolating between those four points is the same as saying I've got four basis functions that are active, and each of them contributes in a way that diminishes from their point of attack.

OK. So now I've got a linear function approximator trying to estimate j hat. So how could I possibly do temporal distance learning-- yeah, John.

AUDIENCE: Is that volumetric interpolation? Barycentric has smaller--

RUSS TEDRAKE: OK, good. So you can do barycentric in three or four, actually.

AUDIENCE: And here, you have-- this would be-- you'd have three points, though, right, in this problem?

RUSS TEDRAKE: This would still be called barycentric, but it's true the barycentric we did before was just, you take the neighboring three points. It's true. So actually, you can do barycentric with the three neighboring points or the four neighboring points or whatever. The volumetric is actually also called-- is also a barycentric interpolator. But you're right. I should have been more careful.

The way we did barycentric before is we took the three closest points, not the four, right? Taking the four is also a good interpolator. It also is called a barycentric interpolator, actually, right? And the question is just how it grows with the state. Most people use the three closest points because then, in higher dimensions, it's always n plus 1 points instead of the powers of n. Good, thank you.

OK. So I guess, then, the domain of attraction is more like that or something, right? For every cross-hair, there's a-- if you're doing the triangle interpolation, then it's more like that. Yeah?

So what do you think? So how can we make an analogy between going through discrete states and increasing eligibility? This eligibility is really-- I just need to remember that that state was relevant in my history of costs. Can you think of an analogy of how this function approximator might play into those equations?

What I want to get to with function approximation, I want to get an update for alpha that has the same sort of form. This j, remember, was-- in the Markov chain case, that's a vector j, where each element of the vector was the cost to go estimate for the i-th node, right? Now my function approximator is-- again, I'm trying to estimate a vector alpha, but each alpha could potentially affect my estimate in multiple states. The power of function approximators is you don't have to consider every point individually. You get some generalization. One parameter affects multiple outputs. OK?

So what could possibly make this tick? How would you do temporal difference learning with function approximators? Yeah.

AUDIENCE: [INAUDIBLE] Before, we were basically doing it through basis functions that were deltas at every--

RUSS TEDRAKE: Good.

AUDIENCE: --point. So you can break j hat up into be something that is actually based on these other basis functions.

RUSS TEDRAKE: So this should be-- whatever we come up with, this should hopefully be the limiting case where, if my basis function was delta functions, you'd like to get back to that? I think that's, unfortunately, going to be a--

AUDIENCE: OK.

RUSS TEDRAKE: No, no. Well, it just happens I'm going to be taking gradients, so-- but yeah, that's very-- that's a very nice observation, actually. What's that?

AUDIENCE: I think you can't erase because that [INAUDIBLE] Kronecker delta, so it's--

RUSS TEDRAKE: Exactly, yeah.

AUDIENCE: By delta function, he meant infinity or 1?

RUSS TEDRAKE: That's what John was pointing out. So he thinks it's going to be a Kronecker delta because it's going to have-- I guess it could be 1, yeah?

AUDIENCE: So if it is 1, it's actually-- it would be mapping from a Tableau representation to the actual function approximation.

RUSS TEDRAKE: Yeah.

AUDIENCE: But having features that are just 1, and you say--

RUSS TEDRAKE: No, I think that's-- I think it's a very nice observation. If we think of each feature as having height 1 here and domain nothing, right, then that should be the limiting case where we get our Markov chain. I think that's right.

AUDIENCE: So if you just changed to taking, at each step, a weighted sum of nodes--

RUSS TEDRAKE: Yeah.

AUDIENCE: --then mapping that weighted sum, [INAUDIBLE].

RUSS TEDRAKE: Yeah, I mean, that's effectively right. So the way that you can do it turns out to be a little bit-- again, a little bit nice and magical, OK? So it turns out-- so there's a couple of ways to think about this. One of them is when I'm going through the Markov chain, every time I get here, I'm going to think forward about-- I'm going to do a one-step prediction, I'm going to do a two-step prediction, I'm going to do a three-step prediction. And what happens is that these eligibility traces are just this trick which makes all that work. If I just remember where I've been, then I can make an update, which is the same as looking forward. Instead, I'm looking back in time with my eligibility traces.

In the function approximator case, doing exactly what you said turns out to be equivalent to remembering how important that parameter alpha i was in your recent history. OK? Does that make sense? If I remember the contribution that alpha made-- let's say one of the elements of alpha, alpha i, alpha j-- made in my recent history, then I can update alpha in the same sort of trick that this eligibility trace worked on for.

AUDIENCE: Would you do some kind of decaying exponential?

RUSS TEDRAKE: Yeah, yeah.

AUDIENCE: That's kind of what it's doing there.

RUSS TEDRAKE: This is exactly what it's doing here. And here, we had the special case where when I visited the state, I got a 1, and when I didn't visit it, I get here. I got nothing. It just decayed. And what we're going to get now is an eligibility trace on alpha. Do you have it? Yeah?

AUDIENCE: Well, I mean, does this thing in brackets need to be changed at all?

RUSS TEDRAKE: Excellent.

AUDIENCE: It seems like delta--

RUSS TEDRAKE: The eligibility trace changes.

AUDIENCE: [INAUDIBLE]

RUSS TEDRAKE: Perfect, right? This is gamma. It's going to do a decaying exponential still. You want to forget things. This thing is now-- the new eligibility trace here is the same size as alpha, not the number of nodes in the system. And the amount that I'm going to credit each alpha with is the gradient of my estimate. Does that sort of make sense? Yeah?

In the case of linear function approximators, the gradient of this is just phi of x. Partial j, partial alpha. Just gets rid of that alpha. OK?

So I remember the relative contribution of each of my alphas in the recent past, and then, based on this e, I make the same update that I did before. Just copy this down. But my e is now the eligibility on alpha.

AUDIENCE: [INAUDIBLE] based on that one to the one that gets visited and zero to the others.

RUSS TEDRAKE: Which is what? What's that? Yeah?

OK. So that is an intuitively correct algorithm, I think. So it seems pretty natural, using the eligibility argument, that this could work. Proving that it works turns out to be a pain. It's not actually an update like we would normally have. There's a special case.

So in one case, when lambda equals 1, then you can actually show that this TD update with linear function approximation is doing gradient descent on the difference between my j lambda-- what I call j lambda-- and my other j squared. OK?

So in the lambda versus 1 case, it actually is doing gradient descent on the Monte Carlo error. In every other setting of lambda, the algorithm is not a standard optimization framework of going down in some steepest descent kind of approach. But it tends-- in some cases, it works faster because it uses previous information in a heuristic sort of way, but it does it very effectively. And people have done the work. In fact, the work was done upstairs by Tsitsiklis and Van Roy in '97 or something like this to prove that, for all lambdas between 0 and 1 in linear function approximators, that this update will go to the true value estimate as your system runs. OK?

AUDIENCE: Did they mention that the [INAUDIBLE]?

RUSS TEDRAKE: They should.

AUDIENCE: [INAUDIBLE] I guess different algorithms can reach a different result, and lambda equals 1 only converges to the actual thing that you're looking for. As you start varying lambda, you converge toward different things, but it's still converging [INAUDIBLE].

RUSS TEDRAKE: That's possible, but--

AUDIENCE: Different lambdas?

RUSS TEDRAKE: I mean--

AUDIENCE: Once your [INAUDIBLE] is correct, doesn't lambda not matter anymore? Because it's going to project to the future--

RUSS TEDRAKE: Yeah, the only stationary point should be the true cost to go function. So if it converges-- I'd be surprised if that's true.

AUDIENCE: Your lambda [INAUDIBLE]--

RUSS TEDRAKE: I mean, Alborz has done the stuff, so you might know it better than I do. But I was under the impression it converges to the true value estimate. I mean, it's all based on these contraction mappings, and I think the stationary point of the contraction is the true value. If you find out differently, then tell us next class, definitely.

OK. So what just happened? So I made a trivial change to the algorithm. In fact, in some ways, it looks almost easier. It's one less line. And it now suddenly works with linear function approximators. So I don't have to feel like I discretized my state space. I can cover my state space with radial basis functions. That might be as painful, by the way, as discretizing the state space if you have to put a lot of radial basis functions down. But I could also do it with more complicated-- I could do it with polynomials, I could do it with Fourier bases, and potentially, things that work with less basis functions over a very large domain. And now suddenly, the tools scale up to little bit higher dimensional systems, as high as you can imagine. As creative as your basis set allows you to be.

To complain about these algorithms is that they're inefficient in their use of data. So if you think-- certainly when we're shooting planes and they break, or if we're using a walking robot and it falls down a lot, then every piece of data should be treated with reverence, right? This is hard-earned data.

These algorithms, as written like this, basically, every time they visit the data, they make some small incremental update, and a throw it away and keep moving. And so, by no means are they efficient in data. They are efficient only in the sense that they use previous estimates of j hat to bootstrap, but there's no sense of remembering all your data and reusing it.

So some people have thought about replaying trajectories. So if you store all your trajectories-- let's say I ran my plane 10 times, well, I'll remember all that data and I'll just run the TD update over and over again over those same 10 trajectories. That's a reasonable thing to do. But again, with linear function approximators, you can do better, right? You can do LSTD, least squares temporal difference learning. This is least squares temporal difference learning, yeah?

The argument basically goes like this. So in learning, there's always a difference-- people like to distinguish between online versus batch updates. So the online-- this is an online update. I took my piece of data in. I immediately changed alpha and then I spat it out, right?

You could imagine a "batch" update, which collected a bunch of these trajectories. Let's say it ran a whole trajectory with the plane. Stop. Now process that trajectory, make a change in alpha, and then repeat. So instead of doing it at every single time step, let's collect a little bit of data and then make an update, OK?

So let's just write down what a batch update-- the batch update for this system, if I ran a trajectory and then made an update, that update would look like this, just by applying that rule a bunch of times but without changing alpha in between, right? So another way to say it is I'm going to hold alpha fixed, collect up all the changes I want to make at alpha, and then, at the end of the run, make one change to alpha, OK?

Well, then that change, if you do it in the batch mode, is just going to be a sum of all these guys. That's a scalar times a vector. So I can reorder it without any pain. Oops. I got to put that inside.

Let me write out the intermediate step here. j alpha ik plus 1 minus j alpha ik. Agree with that? That's the batch update of that. I just collect them all up, I sum them over k here, all my time steps. That's what my update's going to look like.

If I write this a little bit-- if I break into my function approximator there, I can write it like this. hm over k. Oh, boy, I forgot my ek over here. Sorry about that. There's an ek there too, right? j is just phi times alpha, so I'm going to break into that.

If I collect those terms, then I get something we can think about again. Sorry, a times alpha, where this is b and this guy here is a.

In other words, the long-term behavior of this system, if I collect the updates and then make them like this, well, this looks like a low-pass filter, really, that's going to this solution. Yeah. The steady state is alpha equals a inverse phi, where you do that inverse carefully. svd or something.

OK. So that observation led to this least squares temporal difference learning algorithm, which said instead of chewing up every piece of-- every data point and spitting it out, remember everything that you have experienced in the past. And instead of doing this sort of incremental step that goes epsilon towards the steady state, you can solve for that steady state every time you have-- if you collect b and a, you can just collect that with a bunch of data, go ahead and jump right to the solution.

So LSTD, what I think a lot of people would consider the state of the art if you just want to do policy evaluation, build up b and a as you run. Compute alpha is a inverse b whenever you need a new estimate of alpha, OK?

Why do you want to do that? It's more efficient with your data. You remember and replay your last data seamlessly. You don't have to guess some silly learning rate. And it doesn't even depend on your initial conditions anymore, your initial guess at alpha. OK?

So if I went to go and do-- if I were given a robot that's currently performing some feedback controller. Let's say it's stochastic. It's a stochastic system. There's noise and everything like that. And I wanted to just evaluate how well it performed on this cost function, some cost function that someone gave me. If [INAUDIBLE] says I got this robot, I wanted it to optimize this cost function. How is it doing? Tell me how it's doing.

I would look at that-- I would look at the state space. I'd try to come up with a linear tiling of radial basis functions, or some linear function approximator over that state space, and I'd start running trials and I'd keep those trials and store up these matrices, and do a least squares temporal difference update. And this result basically tells me that it's going to do the same thing as the TD. It's going to do it potentially more effectively because it's more efficient with data, and it's going to do it without having to guess initial vector or having some learning rate.

AUDIENCE: Do you actually have to define the length of an episode, for example, if it's a periodic system like a walking system? [INAUDIBLE]

RUSS TEDRAKE: That's really good. So Alborz here has actually written a paper on incremental LSTD. So you might think that doing that update is expensive, and it can be if you just naively do that. But you can do recursively squares, and you can sort of make a cheaper online update to try to do this, to make this a constant update of alpha. If you choose to update alpha at every step, which you could choose to do, and maybe you would choose to do on a non-episodic task, or maybe in walking, you do it once per step or whatever, then you can do it nicely with an incremental LSTD. And you can look at Alborz's paper to figure out how to do that, which is an approximation of the true LSTD, but a pretty good one in your experiments, right? And much more efficient, right?

Yeah, or there's a lot of time-- I mean, I think, in walking, it turns out I would do it probably once per step or something. There's natural discretizations. But there's nothing to say, if you have the computational horsepower, that you couldn't just do this every step too. It's just more expensive than doing an incremental version. OK?

So function approximation's very powerful. This is what's going to take our tabular based ideas and our Markov chain ideas and make them scale to the real world. This is the first year I did function approximation first in the temporal different learning case, but of course it's relevant for the policy gradient world too, right? John showed you different function approximators that were doing reinforce.

Instead of parameterizing my value function as a function approximator, I could have also just directly parameterized my feedback controller as a value function, and done gradient descent if I had a model, or reinforce if I didn't have a model. Function approximation is supposed to be the savior of reinforcement learning. The problem is there's limited results. I mean, the linear function approximation is really the only case we have strong results for most of these cases.

So I'm going to talk about doing the policy stuff with function approximation on Thursday, and then it culminates in actor-critic, where you do both function approximation in the policy and the value function simultaneously. That'll happen sometime next week. Excellent. I'll see you next week.