MachineLearning-Lecture17

Instructor (Andrew Ng):Okay, good morning. Welcome back. So I hope all of you had a good Thanksgiving break. After the problem sets, I suspect many of us needed one. Just one quick announcement so as I announced by email a few days ago, this afternoon we'll be doing another tape ahead of lecture, so I won't physically be here on Wednesday, and so we'll be taping this Wednesday's lecture ahead of time. If you're free this afternoon, please come to that; it'll be at 3:45 p.m. in the Skilling Auditorium in Skilling 193 at 3:45. But of course, you can also just show up in class as usual at the usual time or just watch it online as usual also.

Okay, welcome back. What I want to do today is continue our discussion on Reinforcement Learning in MDPs. Quite a long topic for me to go over today, so most of today's lecture will be on continuous state MDPs, and in particular, algorithms for solving continuous state MDPs, so I'll talk just very briefly about discretization. I'll spend a lot of time talking about models, assimilators of MDPs, and then talk about one algorithm called fitted value iteration and two functions which builds on that, and then hopefully, I'll have time to get to a second algorithm called, approximate policy iteration

Just to recap, right, in the previous lecture, I defined the Reinforcement Learning problem and I defined MDPs, so let me just recap the notation. I said that an MDP or a Markov Decision Process, was a ? tuple, comprising those things and the running example of those using last time was this one right, adapted from the Russell and Norvig AI textbook. So in this example MDP that I was using, it had 11 states, so that's where S was. The actions were compass directions: north, south, east and west.

The state transition probability is to capture chance of your transitioning to every state when you take any action in any other given state and so in our example that captured the stochastic dynamics of our robot wondering around [inaudible], and we said if you take the action north and the south, you have a .8 chance of actually going north and .1 chance of veering off, so that .1 chance of veering off to the right so said model of the robot's noisy dynamic with a [inaudible] and the reward function was that +/-1 at the absorbing states and -0.02 elsewhere. This is an example of an MDP, and that's what these five things were. Oh, and I used a discount factor G of usually a number slightly less than one, so that's the 0.99. And so our goal was to find the policy, the control policy and that's at ?, which is a function mapping from the states of the actions that tells us what action to take in every state, and our goal was to find a policy that maximizes the expected value of our total payoff. So we want to find a policy. Well, let's see. We define value functions Vp (s) to be equal to this. We said that the value of a policy ? from State S was given by the expected value of the sum of discounted rewards, conditioned on your executing the policy ? and you're stating off your [inaudible] to say in the State S, and so our strategy for finding the policy was sort of comprised of two steps. So the goal is to find a good policy that maximizes the suspected value of the sum of discounted rewards, and so I said last time that one strategy for finding the [inaudible] of a policy is to first compute the optimal value function which I denoted V*(s) and is defined like that. It's the maximum value that any policy can obtain, and for example, the optimal value function for that

MDP looks like this. So in other words, starting from any of these states, what's the expected value of the sum of discounted rewards you get, so this is V*. We also said that once you've found V*, you can compute the optimal policy using this.

And so once you've found V*, we can use this equation to find the optimal policy ?* and the last piece of this algorithm was Bellman's equations where we know that V*, the optimal sum of discounted rewards you can get for State S, is equal to the immediate reward you get just for starting off in that state +G(for the max over all the actions you could take)(your future sum of discounted rewards)(your future payoff starting from the State S(p) which is where you might transition to after 1(s). And so this gave us a value iteration algorithm, which was essentially V.I.

I'm abbreviating value iteration as V.I., so in the value iteration algorithm, in V.I., you just take Bellman's equations and you repeatedly do this. So initialize some guess of the value functions. Initialize a zero as the sum rounding guess and then repeatedly perform this update for all states, and I said last time that if you do this repeatedly, then V(s) will converge to the optimal value function, V*(s) and then having found V*(s), you can compute the optimal policy ?*.

Just one final thing I want to recap was the policy iteration algorithm in which we repeat the following two steps. So let's see, given a random initial policy, we'll solve for Vp. We'll solve for the value function for that specific policy. So this means for every state, compute the expected sum of discounted rewards for if you execute the policy ? from that state, and then the other step of policy iteration is having found the value function for your policy, you then update the policy pretending that you've already found the optimal value function, V*, and then you repeatedly perform these two steps where you solve for the value function for your current policy and then pretend that that's actually the optimal value function and solve for the policy given the value function, and you repeatedly update the value function or update the policy using that value function. And last time I said that this will also cause the estimated value function V to converge to V* and this will cause p to converge to ?*, the optimal policy.

So those are based on our last lecture [inaudible] MDPs and introduced a lot of new notation symbols and just summarize all that again. What I'm about to do now, what I'm about to do for the rest of today's lecture is actually build on these two algorithms so I guess if you have any questions about this piece, ask now since I've got to go on please. Yeah.

Student:[Inaudible] how those two algorithms are very different?

Instructor (Andrew Ng):I see, right, so yeah, do you see that they're different? Okay, how it's different. Let's see. So well here's one difference. I didn't say this 'cause no longer use it today. So value iteration and policy iteration are different algorithms. In policy iteration in this step, you're given a fixed policy, and you're going to solve for the value function for that policy and so you're given some fixed policy ?, meaning some function mapping from the state's actions. So give you some policy and whatever. That's just some policy; it's not a great policy. And in that step that I circled, we have to find the ? of S which means that for every state you need to compute your expected sum of discounted rewards or if you execute this specific policy and starting off the MDP in that state S.

So I showed this last time. I won't go into details today, so I said last time that you can actually solve for Vp by solving a linear system of equations. There was a form of Bellman's equations for Vp, and it turned out to be, if you write this out, you end up with a linear system of 11 equations of 11 unknowns and so you can actually solve for the value function for a fixed policy by solving like a system of linear equations with 11 variables and 11 constraints, and so that's policy iteration; whereas, in value iteration, going back on board, in value iteration you sort of repeatedly perform this update where you update the value of a state as the [inaudible]. So I hope that makes sense that the algorithm of these is different.

Student:[Inaudible] on the atomic kits so is the assumption that we can never get out of those states?

Instructor (Andrew Ng):Yes. There's always things that you where you solve for this [inaudible], for example, and make the numbers come up nicely, but I don't wanna spend too much time on them, but yeah, so the assumption is that once you enter the absorbing state, then the world ends or there're no more rewards after that and you can think of another way to think of the absorbing states which is sort of mathematically equivalent. You can think of the absorbing states as transitioning with probability 1 to sum 12 state, and then once you're in that 12th state, you always remain in that 12th state, and there're no further rewards from there. If you want, you can think of this as actually an MDP with 12 states rather than 11 states, and the 12th state is this zero cost absorbing state that you get stuck in forever. Other questions? Yeah, please go.

Student: Where did the Bellman's equations [inaudible] to optimal value [inaudible]?

Instructor (Andrew Ng):Boy, yeah. Okay, this Bellman's equations, this equation that I'm pointing to, I sort of tried to give it justification for this last time. I'll say it in one sentence so that's that the expected total payoff I get, I expect to get something from the state as is equal to my immediate reward which is the reward I get for starting a state. Let's see. If I sum the state, I'm gonna get some first reward and then I can transition to some other state, and then from that other state, I'll get some additional rewards from then. So Bellman's equations breaks that sum into two pieces. It says the value of a state is equal to the reward you get right away is really, well. V*(s) is really equal to +G, so this is V*(s) is, and so Bellman's equations sort of breaks V* into two terms and says that there's this first term which is the immediate reward, that, and then +G(the rewards you get in the future) which it turns out to be equal to that second row.

I spent more time justifying this in the previous lecture, although yeah, hopefully, for the purposes of this lecture, if you're not sure where this is came, if you don't remember the justification of that, why don't you just maybe take my word for that this equation holds true since I use it a little bit later as well, and then the lecture notes sort of explain a little further the justification for why this equation might hold true. But for now, yeah, just for now take my word for it that this holds true ‘cause we’ll use it a little bit later today as well.

Student:[Inaudible] and would it be in sort of turn back into [inaudible].

Instructor (Andrew Ng):Actually, [inaudible] right question is if in policy iteration if we represent ? implicitly, using V(s), would it become equivalent to valuation, and the answer is sort of no. Let’s see. It’s true that policy iteration and value iteration are closely related algorithms, and there’s actually a continuum between them, but yeah, it actually turns out that, oh, no, the algorithms are not equivalent. It’s just in policy iteration, there is a step where you’re solving for the value function for the policy vehicle is V, solve for Vp. Usually, you can do this, for instance, by solving a linear system of equations. In value iteration, it is a different algorithm, yes. I hope it makes sense that at least cosmetically it’s different.

Student:[Inaudible] you have [inaudible] representing ? implicitly, then you won’t have to solve that to [inaudible] equations.

Instructor (Andrew Ng):Yeah, the problem is - let’s see. To solve for Vp, this works only if you have a fixed policy, so once you change a value function, if ? changes as well, then it’s sort of hard to solve this. Yeah, so later on we’ll actually talk about some examples of when ? is implicitly represented but at least for now it’s I think there’s – yeah. Maybe there’s a way to redefine something, see a mapping onto value iteration but that’s not usually done. These are viewed as different algorithms.

Okay, cool, so all good questions. Let me move on and talk about how to generalize these ideas to continuous states. Everything we’ve done so far has been for discrete states or finite-state MDPs. Where, for example, here we had an MDP with a finite set of 11 states and so the value function or V(s) or our estimate for the value function, V(s), could then be represented using an array of 11 numbers ‘cause if you have 11 states, the value function needs to assign a real number to each of the 11 states and so to represent V(s) using an array of 11 numbers. What I want to do for [inaudible] today is talk about continuous states, so for example, if you want to control any of the number of real [inaudible], so for example, if you want to control a car, a car is positioned given by XYT, as position and orientation and if you want to Markov the velocity as well, then Xdot, Ydot, Tdot, so these are so depending on whether you want to model the kinematics and so just position, or whether you want to model the dynamics, meaning the velocity as well.

Earlier I showed you video of a helicopter that was flying, using a rain forest we’re learning algorithms, so the helicopter which can fly in three-dimensional space rather than just drive on the 2-D plane, the state will be given by XYZ position, FT?, which is ?[inaudible]. The FT? is sometimes used to note the P[inaudible] of the helicopter, just orientation, and if you want to control a helicopter, you pretty much have to model velocity as well which means both linear velocity as well as angular velocity, and so this would be a 12-dimensional state.

If you want an example that is kind of fun but unusual is, and I'm just gonna use this as an example and actually use this little bit example in today's lecture is the inverted pendulum problem which is sort of a long-running classic in reinforcement learning in which imagine that you have a little cart that's on a rail. The rail ends at some point and if you imagine that you have a pole attached to the cart, and this is a free hinge and so the pole here can rotate freely, and your goal is to control the cart and to move it back and forth on this rail so as to keep the pole balanced. Yeah, there's no long pole in this class but you know what I mean, so you can imagine. Oh, is there a long pole here?

Student: Back in the corner.

Instructor (Andrew Ng):Oh, thanks. Cool. So I did not practice this but you can take a long pole and sort of hold it up, balance, so imagine that you can do it better than I can. Imagine these are [inaudible] just moving back and forth to try to keep the pole balanced, so you can actually us the reinforcement learning algorithm to do that. This is actually one of the longstanding classic problems that people [inaudible] implement and play off using reinforcement learning algorithms, and so for this, the states would be X and T, so X would be the position of the cart, and T would be the orientation of the pole and also the linear velocity and the angular velocity of the pole, so I'll actually use this example a couple times.

So to read continuous state space, how can you apply an algorithm like value iteration and policy iteration to solve the MDP to control like the car or a helicopter or something like the inverted pendulum? So one thing you can do and this is maybe the most straightforward thing is, if you have say a two-dimensional continuous state space, a S-1 and S-2 are my state variables, and in all the examples there are I guess between 4-dimensional to 12-dimensional. I'll just draw 2-D here. The most straightforward thing to do would be to take the continuous state space and discretize it into a number of discrete cells.

And I use S-bar to denote they're discretized or they're discrete states, and so you can [inaudible] with this continuous state problem with a finite or discrete set of states and then you can use policy iteration or value iteration to solve for V*(s)-bar and ?*(s)-bar. And if you're robot is then in some state given by that dot, you would then figure out what discretized state it is in. In this case it's in, this discretized dygrid cell that's called S-bar, and then you execute. You choose the policy. You choose the action given by applied to that discrete state, so discretization is maybe the most straightforward way to turn a continuous state problem into a discrete state problem.

Sometimes you can sorta make this work but a couple of reasons why this does not work very well. One reason is the following, and for this picture, let's even temporarily put aside reinforcement learning. Let's just think about doing regression for now and so suppose you have some invariable X and suppose I have some data, and I want to fill a function. Y is the function of X, so discretization is saying that I'm going to take my horizontal Xs and chop it up into a number of intervals. Sometimes I call these intervals buckets as well. We chop my horizontal Xs up into a number of buckets and then we're approximate this function using something that's piecewise constant in each of these buckets. And just look at this.

This is clearly not a very good representation, right, and when we talk about regression, you just choose some features of X and run linear regression or something. You get a much better fit to the function. And so the sense that discretization just isn't a very good source of piecewise constant functions. This just isn't a very good function for representing many things, and there's also the sense that there's no smoothing or there's no generalization across the different buckets. And in fact, back in regression, I would never have chosen to do regression using this sort of visualization. It's just really doesn't make sense.

And so in the same way, instead of X, V(s), instead of X and some hypothesis function of X, if you have the state here and you're trying to approximate the value function, then you can get discretization to work for many problems but maybe this isn't the best representation to represent a value function. The other problem with discretization and maybe the more serious problem is what's often somewhat fancifully called the curse of dimensionality. And just the observation that if the state space is in RN, and if you discretize each variable into K buckets, so if discretize each variable into K discrete values, then you get on the order of K to the power of N discrete states. In other words, the number of discrete states you end up with grows exponentially in the dimension of the problem, and so for a helicopter with 12-dimensional state space, this would be maybe like 100 to the power of 12, just huge, and it's not feasible. And so discretization doesn't scale well at all with two problems in high-dimensional state spaces, and this observation actually applies more generally than to just robotics and continuous state problems. For example, another fairly well-known applications of reinforcement learning has been to factory automations. If you imagine that you have 20 machines sitting in the factory and the machines lie in a assembly line and they all do something to a part on the assembly line, then they route the part onto a different machine. You want to use reinforcement learning algorithms, [inaudible] the order in which the different machines operate on your different things that are flowing through your assembly line and maybe different machines can do different things. So if you have N machines and each machine can be in K states, then if you do this sort of discretization, the total number of states would be K to N as well. If you have N machines and if each machine can be in K states, then again, you can get this huge number of states. Other well-known examples would be if you have a board game is another example. You'd want to use reinforcement learning to play chess. Then if you have N pieces on your board game, you have N pieces on the chessboard and if each piece can be in K positions, then this is a game sort of the curse of dimensionality thing where the number of discrete states you end up with goes exponentially with the number of pieces in your board game. So the curse of dimensionality means that discretization scales poorly to high-dimensional state spaces or at least discrete representations scale poorly to high-dimensional state spaces. In practice, discretization will usually, if you have a 2-dimensional problem, discretization will usually work great. If you have a 3-dimensional problem, you can often get discretization to work not too badly without too much trouble. With a 4-dimensional problem, you can still often get to where that they could be challenging and as you go to higher and higher dimensional state spaces, the odds and [inaudible] that you need to figure around to discretization and do things like non-uniform grids, so for example, what I've drawn for you is an example of a non-uniform discretization where I'm discretizing S-2 much more finally than S-1. If I think the value function is much more sensitive to the value of state variable S-2 than to S-1, and so as you get into higher dimensional state spaces, you may need to manually fiddle with choices like these with no uniform discretizations and so on. But the folk wisdom seems to be that if you have 2- or 3-dimensional problems, it work fine. With 4-dimensional problems, you can probably get it to work but it'd be just slightly challenging and you can sometimes by fooling around and being clever, you can often push discretization up to let's say about 6-dimensional problems but with some difficulty and problems higher than 6-dimensional would be extremely difficult to solve with discretization. So that's just rough folk wisdom order of managing problems you think about using for discretization. But what I want to spend most of today talking about is [inaudible] methods that often work much better than discretization and which we will approximate V* directly without resulting to these sort of discretizations. Before I jump to the specific representation let me just spend a few minutes talking about the problem setup then. For today's lecture, I'm going to focus on the problem of continuous states and just to keep things sort of very simple in this lecture, I want view of continuous actions, so I'm gonna see discrete actions A. So it turns out actually that is a critical fact also for many problems, it turns out that the state space is much larger than the states of actions. That just seems to have worked out that way for many problems, so for example, for driving a car the state space is 6-dimensional, so if XY T, Xdot, Ydot, Tdot. Whereas, your action has, you still have two actions. You have forward backward motion and steering the car, so you have 6-D states and 2-D actions, and so you can discretize the action much more easily than discretize the states. The only examples down for a helicopter you've 12-D states in a 4-dimensional action it turns out, and it's also often much easier to just discretize a continuous actions into a discrete sum of actions. And for the inverted pendulum, you have a 4-D state and a 1-D action. Whether you accelerate your cart to the left or the right is one D action and so for the rest of today, I'm gonna assume a continuous state but I'll assume that maybe you've already discretized your actions, just because in practice it turns out that not for all problems, with many problems large actions is just less of a difficulty than large state spaces. So I'm going to assume that we have a model or simulator of the MDP, and so this is really an assumption on how the state transition probabilities are represented. I'm gonna assume and I'm going to use the terms "model" and "simulator" pretty much synonymously, so specifically, what I'm going to assume is that we have a black box and a piece of code, so that I can input any state, input an action and it will output S prime, sample from the state transition distribution. Says that this is really my assumption on the representation I have for the state transition probabilities, so I'll assume I have a box that read take us in for the stated action and output in mixed state. And so since they're fairly common ways to get models of different MDPs you may work with, one is you might get a model from a physics simulator. So for example, if you're interested in controlling that inverted pendulum, so your action is A which is the magnitude of the force you exert on the cart to left or right, and your state is Xdot, T, Tdot. I'm just gonna write that in that order. And so I'm gonna write down a bunch of equations just for completeness but everything I'm gonna write below here is most of what I wanna write is a bit gratuitous, but so since I'll maybe flip open a textbook on physics, a textbook on mechanics, you can work out the equations of motion of a physical device like this, so you find that Sdot. The dot denotes derivative, so the derivative of the state with respect to time is given by Xdot, ?-L(B) cost B over M Tdot, B. And so on where A is the force is the action that you exert on the cart. L is the length of the pole. M is the total mass of the system and so on. So all these equations are good uses, just writing them down for completeness, but by flipping over, open like a physics textbook, you can work out these equations and notions yourself and this then gives you a model which can say that S-2+1. You're still one time step later will be equal to your previous state plus [inaudible], so in your simulator or in my model what happens to the cart every 10th of a second, so ? T would be within one second and then so plus ? T times that. And so that'd be one way to come up with a model of your MDP. And in this specific example, I've actually written down deterministic model because and by deterministic I mean that given an action in a state, the next state is not random, so would be an example of a deterministic model where I can compute the next state exactly as a function of the previous state and the previous action or it's a deterministic model because all the probability mass is on a single state given the previous stated action. You can also make this a stochastic model. A second way that is often used to attain a model is to learn one. And so again, just concretely what you do is you would imagine that you have a physical inverted pendulum system as you physically own an inverted pendulum robot. What you would do is you would then initialize your inverted pendulum robot to some state and then execute some policy, could be some random policy or some policy that you think is pretty good, or you could even try controlling yourself with a joystick or something. But so you set the system off in some state as zero. Then you take some action. Here's zero and the game could be chosen by some policy or chosen by you using a joystick tryina control your inverted pendulum or whatever. System would transition to some new state, S-1, and then you take some new action, A-1 and so on. Let's say you do this for two time steps and sometimes I call this one trajectory and you repeat this M times, so this is the first trial of the first trajectory, and then you do this again. Initialize it in some and so on. So you do this a bunch of times and then you would run the learning algorithm to estimate ST+1 as a function of ST and AT. And for sake of completeness, you should just think of this as inverted pendulum problem, so ST+1 is a 4-dimensional vector. ST, AT will be a 4-dimensional vector and that'll be a real number, and so you might run linear regression 4 times to predict each of these state variables as a function of each of these 5 real numbers and so on.

Just for example, if you say that if you want to estimate your next state ST+1 as a linear function of your previous state in your action and so A here will be a 4 by 4 matrix, and B would be a 4-dimensional vector, then you would choose the values of A and B that minimize this. So if you want your model to be that ST+1 is some linear function of the previous stated action, then you might pose this optimization objective and choose A and B to minimize the sum of squares error in your predictive value for ST+1 as the linear function of ST and AT. I should say that this is one specific example where you're using a linear function of the previous stated action to predict the next state.

Of course, you can also use other algorithms like low [inaudible] weight to linear regression or linear regression with nonlinear features or kernel linear regression or whatever to predict the next state as a nonlinear function of the current state as well, so this is just [inaudible] linear problems. And it turns out that low [inaudible] weight to linear regression is for many robots turns out to be an effective method for this learning problem as well.

And so having learned to model, having learned the parameters A and B, you then have a model where you say that ST+1 is AST plus BAT, and so that would be an example of a deterministic model or having learned the parameters A and B, you might say that ST+1 is equal to AST + BAT + ?T. And so these would be very reasonable ways to come up with either a deterministic or a stochastic model for your inverted pendulum MDP. And so just to summarize, what we have now is a model, meaning a piece of code, where you can input a state and an action and get an ST+1. And so if you have a stochastic model, then to influence this model, you would actually sample ?T from this [inaudible] distribution in order to generate ST+1.

So it actually turns out that in a preview, I guess, in the next lecture it actually turns out that in the specific case of linear dynamical systems, in the specific case where the next state is a linear function of the current stated action, it actually turns out that there're very powerful algorithms you can use. So I'm actually not gonna talk about that today. I'll talk about that in the next lecture rather than right now but turns out that for many problems of inverted pendulum go if you use low [inaudible] weights and linear regressionals and long linear algorithm 'cause many systems aren't really linear. You can build a nonlinear model.

So what I wanna do now is talk about given a model, given a simulator for your MDP, how to come up with an algorithm to approximate the alpha value function piece. Before I move on, let me check if there're questions on this. Okay, cool. So here's the idea. Back when we talked about linear regression, we said that given some inputs X in supervised learning, given the input feature is X, we may choose some features of X and then approximate the type of variable as a linear function of various features of X, and so just do exactly the same thing to approximate the optimal value function, and in particular, we'll choose some features 5-S of a state S.

And so you could actually choose 5-S equals S. That would be one reasonable choice, if you want to approximate the value function as a linear function of the states, but you can also choose other things, so for example, for the inverted pendulum example, you may choose 5-S to be equal to a vector of features that may be [inaudible]1 or you may have Xdot2, Xdot maybe some cross terms, maybe times X, maybe dot2 and so on. So you choose some vector or features and then approximate the value function as the value of the state as is equal to data transfers times the features. And I should apologize in advance; I'm overloading notation here. It's unfortunate. I use data both to denote the angle of the cart of the pole inverted pendulum. So this is known as the angle T but also using T to denote the vector of parameters in my [inaudible] algorithm. So sorry about the overloading notation.

Just like we did in linear regression, my goal is to come up with a linear combination of the features that gives me a good approximation to the value function and this is completely analogous to when we said that in linear regression our estimate, my response there but Y as a linear function a feature is at the input. That's what we have in linear regression. Let me just write down value iteration again and then I'll written down an approximation to value iteration, so for discrete states, this is the idea behind value iteration and we said that V(s) will be updated as R(s) + G [inaudible].

That was value iteration and in the case of continuous states, this would be the replaced by an [inaudible], an [inaudible] over states rather via sum over states. Let me just write this as R(s) + G([inaudible]) and then that sum over T's prime. That's really an expectation with respect to random state as prime drawn from the state transition probabilities piece SA of V(s) prime. So this is a sum of all states S prime with the probability of going to S prime (value), so that's really an expectation over the random state S prime flowing from PSA of that. And so what I'll do now is write down an algorithm called fitted value iteration that's in approximation to this but specifically for continuous states. I just wrote down the first two steps, and then I'll continue on the next board, so the first step of the algorithm is we'll sample. Choose some set of states at random. So sample S-1, S-2 through S-M randomly so choose a set of states randomly and initialize my parameter vector to be equal to zero. This is analogous to in value iteration where I might initialize the value function to be the function of all zeros. Then here's the end view for the algorithm. Got quite a lot to write actually. Let's see. And so that's the algorithm. Let me just adjust the writing. Give me a second. Give me a minute to finish and then I'll step through this. Actually, if some of my handwriting is eligible, let me know. So let me step through this and so briefly explain the rationale. So the hear of the algorithm is - let's see. In the original value iteration algorithm, we would take the value for each state, V(s)I, and we will overwrite it with this expression here. In the original, this discrete value iteration algorithm was to V(s)I and we will set V(s)I to be equal to that, I think. Now we have in the continuous state case, we have an infinite continuous set of states and so you can't discretely set the value of each of these to that. So what we'll do instead is choose the parameters T so that V(s)I is as close as possible to this thing on the right hand side instead. And this is what YI turns out to be. So completely, what I'm going to do is I'm going to construct estimates of this term, and then I'm going to choose the parameters of my function approximator. I'm gonna choose my parameter as T, so that V(s)I is as close as possible to these. That's what YI is, and specifically, what I'm going to do is I'll choose parameters data to minimize the sum of square differences between T [inaudible] plus 5SI. This thing here is just V(s)I because I'm approximating V(s)I is a linear function of 5SI and so choose the parameters data to minimize the sum of square differences. So this is last step is basically the approximation version of value iteration. What everything else above was doing was just coming up with an approximation to this term, to this thing here and which I was calling YI. And so confluently, for every state SI we want to estimate what the thing on the right hand side is and but there's an expectation here. There's an expectation over a continuous set of states, may be a very high dimensional state so I can't compute this expectation exactly. What I'll do instead is I'll use my simulator to sample a set of states from this distribution from this P substrip, SIA, from the state transition distribution of where I get to if I take the action A in the state as I, and then I'll average over that sample of states to compute this expectation. And so stepping through the algorithm just says that for each state and for each action, I'm going to sample a set of states. This S prime 1 through S prime K from that state transition distribution, still using the model, and then I'll set Q(a) to be equal to that average and so this is my estimate for R(s)I + G(this expected value for that specific action A). Then I'll take the maximum of actions A and this gives me YI, and so YI is for S for that. And finally, I'll run really run linear regression which is that last of the set [inaudible] to get V(s)I to be close to the YIs. And so this algorithm is called fitted value iteration and it actually often works quite well for continuous, for problems with anywhere from 6- to 10- to 20-dimensional state spaces if you can choose appropriate features. Can you raise a hand please if this algorithm makes sense? Some of you didn't have your hands up. Are there questions for those, yeah?

Student:

Is there a recess [inaudible] function in this setup?

Instructor (Andrew Ng):Oh, yes. An MDP comprises SA, the state transition probabilities G and R and so for continuous state spaces, S would be like R4 for the inverted pendulum or something. Actions with discretized state transitions probabilities with specifying with the model or the simulator, G is just a real number like .99 and the real function is usually a function that's given to you.

And so the reward function is some function of your 4-dimensional state, and for example, you might choose a reward function to be minus – I don't know. Just for an example of simple reward function, if we want a -1 if the pole has fallen and there it depends on you choose your reward function to be -1 if the inverted pendulum falls over and to find that its angle is greater than 30° or something and zero otherwise. So that would be an example of a reward function that you can choose for the inverted pendulum, but yes, assuming a reward function is given to you so that you can compute R(s)I for any state. Are there other questions?

Actually, let me try asking a question, so everything I did here assume that we have a stochastic simulator. So it turns out I can simply this algorithm if I have a deterministic simulator, but deterministic simulator is given a stated action, my next state is always exactly determined. So let me ask you, if I have a deterministic simulator, how would I change this algorithm? How would I simplify this algorithm?

Student: Lower your samples that you're drawing [inaudible].

Instructor (Andrew Ng):Right, so Justin's going right. If I have a deterministic simulator, all my samples from those would be exactly the same, and so if I have a deterministic simulator, I can set K to be equal to 1, so I don't need to draw K different samples. I really only need one sample if I have a deterministic simulator, so you can simplify this by setting K=1 if you have a deterministic simulator. Yeah?

Student:I guess I'm really confused about the, yeah, we sorta turned this [inaudible] into something that looks like linear state regression or some' you know the data transpose times something that we're used to but I guess I'm a little. I don't know really know what question to ask but like when we did this before we had like discrete states and everything. We were determined with finding this optimal policy and I guess it doesn't look like we haven't said the word policy in a while so kinda difficult.

Instructor (Andrew Ng):Okay, yeah, so [inaudible] matters back to policy but maybe I should just say a couple words so let me actually try to get at some of what maybe what you're saying. Our strategy for finding optimal policy has been to find some way to find V*, find some way to find the optimal value function and then use that to compute ?* and some of approximations of ?*. So far everything I've been doing has been focused on how to find V*. I just want to say one more word. It actually turns out that for linear regression it's often very easy. It's often not terribly difficult to choose some resource of the features.

Choosing features for approximating the value function is often somewhat harder, so because the value of a state is how good is starting off in this state. What is my expected sum of discounted rewards? What if I start in a certain state? And so what the feature of the state have to measure is really how good is it to start in a certain state? And so for inverted pendulum you actually have that states where the poles are vertical and when a cart that's centered on your track or something, maybe better and so you can come up with features that measure the orientation of the pole and how close you are to the center of the track and so on and those will be reasonable features to use to approximate V*. Although in general it is true that choosing features, the value function approximation, is often slightly trickier than choosing good features for linear regression.

Okay and then Justin's questions of so given V*, how do you go back to actually find a policy? In the discrete case, so we have that ?*(s) is equal to all [inaudible] over A of that. So that's again, I used to write this as a sum over states [inaudible]. I'll just write this as an expectation and so then once you find the optimal value function V*, you can then find the optimal policy ?* by computing the [inaudible]. So if you're in a continuous state MDP, then you can't actually do this in advance for every single state because there's an infinite number of states and so you can't actually perform this computation in advance to every single state.

What you do instead is whenever your robot is in some specific state S is only when your system is in some specific state S like your car is at some position orientation or your inverted pendulum is in some specific position, posed in some specific angle T. It's only when your system, be it a factor or a board game or a robot, is in some specific state S that you would then go ahead and compute this augmax, so it's only when you're in some state S that you then compute this augmax, and then you execute that action A and then as a result of your action, your robot would transition to some new state and then so it'll be given that specific new state that you compute as augmax using that specific state S that you're in.

There’re a few ways to do it. One way to do this is actually the same as in the inner loop of the fitted value iteration algorithm so because of an expectation of a large number of states, you’d need to sample some set of states from the simulator and then approximate this expectation using an average over your samples, so it’s actually as inner loop of the value iteration algorithm. So you could do that. That’s sometimes done. Sometimes it can also be a pain to have to sample a set of states to approximate those expectations every time you want to take an action in your MDP.

Couple of special cases where this can be done, one special case is if you have a deterministic simulator. If it’s a deterministic simulator, so in other words, if your similar is just some function, could be a linear or a nonlinear function. If it’s a deterministic simulator then the next state, ST+1, is just some function of your previous stated action. If that’s the case then this expectation, well, then this simplifies to augmax of A of V* of F of I guess S,A because this is really saying S prime=F(s),A. I switched back and forth between notation; I hope that’s okay. S to denote the current state, and S prime to deterministic state versus ST and ST+1 through the current state. Both of these are sorta standing notation and don’t mind my switching back and forth between them. But if it’s a deterministic simulator you can then just compute what the next state S prime would be for each action you might take from the current state, and then take the augmax of actions A, basically choose the action that gets you to the highest value state.

And so that’s one case where you can compute the augmax and we can compute that expectation without needing to sample an average over some sample. Another very common case actually it turns out is if you have a stochastic simulator, but if your similar happens to take on a very specific form of ST+1=F(s)T,AT+?T where this is galsie noise. The [inaudible] is a very common way to build simulators where you model the next state as a function of the current state and action plus some noise and so once specific example would be that sort of mini dynamical system that we talked about with linear function of the current state and action plus galsie noise. In this case, you can approximate augment over A, well.

In that case you take that expectation that you’re trying to approximate. The expected value of V* of S prime, we can approximate that with V* of the expected value of S prime, and this is approximation. Expected value of a function is usually not equal to the value of an expectation but it is often a reasonable approximation and so that would be another way to approximate that expectation and so you choose the actions according to watch we do the same formula as I wrote just now. And so this would be a way of approximating this augmax, ignoring the noise in the simulator essentially. And this often works pretty well as well just because many simulators turn out to be the form of some linear or some nonlinear function plus zero mean galsie noise, so and just that ignore the zero mean galsie noise, so that you can compute this quickly.

And just to complete about this, what that is, right, that V* F of SA, this you down rate as data transfers Fi of S prime where S prime=F of SA. Great, so this V* you would compute using the parameters data that you just learned using the fitted value iteration algorithm. Questions about this?

Student:[Inaudible] case, for real-time application is it possible to use that [inaudible], for example for [inaudible].

Instructor (Andrew Ng):Yes, in real-time applications is it possible to sample case phases use [inaudible] expectation. Computers today actually amazingly fast. I'm actually often surprised by how much you can do in real time so the helicopter we actually flying a helicopter using an algorithm different than this? I can't say. But my intuition is that you could actually do this with a helicopter. A helicopter would control at somewhere between 10hz and 50hz. You need to do this 10 times a second to 50 times a second, and that's actually plenty of time to sample 1,000 states and compute this expectation.

They're real difficult, helicopters because helicopters are mission critical, and you do something it's like fast. You can do serious damage and so maybe not for good reasons. We've actually tended to avoid tossing coins when we're in the air, so the ideal of letting our actions be some up with some random process is slightly scary and just tend not to do that. I should say that's prob'ly not a great reason because you average a large number of things here very well fine but just as a maybe overly conservative design choice, we actually don't, tend not to find anything randomized on which is prob'ly being over conservative. It's the choice we made 'cause other things are slightly safer. I think you can actually often do this.

So long as I see a model can be evaluated fast enough where you can sample 100 state transitions or 1,000 state transitions, and then do that at 10hz. They haven't said that. This is often attained which is why we often use the other approximations that don't require your drawing a large sample. Anything else? No, okay, cool. So now you know one algorithm [inaudible] reinforcement learning on continuous state spaces. Then we'll pick up with some more ideas on some even more powerful algorithms, the solving MDPs of continuous state spaces. Thanks. Let's close for today.

[End of Audio]

Duration: 77 minutes