MachineLearning-Lecture16

Instructor (Andrew Ng):Okay, let's see. Just some quick announcements. For those of you taking 221, in 221 and 229, I said that in supervised learning there was about one lecture that would overlap and everything else was much more advanced in 229. And some of the reinforcement or anything else in 221, there's about one vector of overlap between 221 and 229, but then right after that, we actually go much further in 229 than we did in 221.

All right, so welcome back. What I want to do today is start a new chapter, a new discussion on machine learning and in particular, I want to talk about a different type of learning problem called reinforcement learning, so that's Markov Decision Processes, value functions, value iteration, and policy iteration. Both of these last two items are algorithms for solving reinforcement learning problems.

As you can see, we're also taping a different room today, so the background is a bit different.

So just to put this in context, the first of the four major topics we had in this class was supervised learning and in supervised learning, we had the training set in which we were given sort of the "right" answer of every training example and it was then just a drop of the learning algorithms to replicate more of the right answers.

And then that was learning theory and then we talked about unsupervised learning, and in unsupervised learning, we had just a bunch of unlabeled data, just the x's, and it was the job in the learning algorithm to discover so-called structure in the data and several algorithms like cluster analysis, K-means, a mixture of all the sort of the PCA, ICA, and so on.

Today, I want to talk about a different class of learning algorithms that's sort of in between supervised and unsupervised, so there will be a class of problems where there's a level of supervision that's also much less supervision than what we saw in supervised learning. And this is a problem in formalism called reinforcement learning. So next up here are slides. Let me show you. As a moving example, here's an example of the sorts of things we do with reinforcement learning.

So here's a picture of – some of this I talked about in Lecture 1 as well, but here's a picture of the – we have an autonomous helicopter we have at Stanford. So how would you write a program to make a helicopter like this fly by itself? I'll show you a fun video. This is actually, I think, the same video that I showed in class in the first lecture, but here's a video taken in the football field at Stanford of using machine learning algorithm to fly the helicopter. So let's just play the video.

You can zoom in the camera and see the trees in the sky. So in terms of autonomous helicopter flight, this is written then by some of my students and me. In terms of autonomous helicopter flight, this is one of the most difficult aerobatic maneuvers flown

and it's actually very hard to write a program to make a helicopter do this and the way this was done was with what's called a reinforcement learning algorithm.

So just to make this more concrete, right, the learning problem in helicopter flight is ten times per second, say. Your sensors on the helicopter gives you a very accurate estimate of the position of orientation of the helicopter and so you know where the helicopter is pretty accurately at all points in time. And your job is to take this input, the position orientation, and to output a set of numbers that correspond to where to move the control sticks to control the helicopter, to make it fly, to right side up, fly upside down, actually whatever maneuver you want.

And this is different from supervised learning because usually we actually don't know what the "right" control action is. And more specifically, if the helicopter is in a certain position orientation, it's actually very hard to say when the helicopter is doing this, you should move the control sticks exactly these positions. So it's very hard to apply supervised learning algorithms to this problem because we can't come up with a training set where the inputs of the position and the output is all the "right" control actions. It's really hard to come up with a training set like that.

Instead of reinforcement learning, we'll give the learning algorithm a different type of feedback, basically called a reward signal, which will tell the helicopter when it's doing well and when it's doing poorly. So what we'll end up doing is we're coming up with something called a reward signal and I'll formalize this later, which will be a measure of how well the helicopter is doing, and then it will be the job of the learning algorithm to take just this reward function as input and try to fly well.

Another good example of reinforcement learning is thinking about getting a program to play a game, to play chess, a game of chess. At any stage in the game, we actually don't know what the "optimal" move is, so it's very hard to pose playing chess as a supervised learning problem because we can't say the x's are the board positions and the y's are the optimum moves because we just don't know how we create any training examples of optimum moves of chess.

But what we do know is if you have a computer playing games of chess, we know when its won a game and when its lost a game, so what we do is we give it a reward signal, so give it a positive reward when it wins a game of chess and give it a negative reward whenever it loses, and hopefully have it learn to win more and more games by itself over time.

So what I'd like you to think about reinforcement learning is think about training a dog. Every time your dog does something good, you sort of tell it, "Good dog," and every time it does something bad, you tell it, "Bad dog," and over time, your dog learns to do more and more good things over time.

So in the same way, when we try to fly a helicopter, every time the helicopter does something good, you say, “Good helicopter,” and every time it crashes, you say, “Bad helicopter,” and then over time, it learns to do the right things more and more often.

The reason – one of the reasons that reinforcement learning is much harder than supervised learning is because this is not a one-shot decision making problem. So in supervised learning, if you have a classification, prediction if someone has cancer or not, you make a prediction and then you’re done, right? And your patient either has cancer or not, you’re either right or wrong, they live or die, whatever. You make a decision and then you’re done.

In reinforcement learning, you have to keep taking actions over time, so it’s called the sequential decision making. So concretely, suppose a program loses a game of chess on move No. 60. Then it has actually made 60 moves before it got this negative reward of losing a game of chess and the thing that makes it hard for the algorithm to learn from this is called the credit assignment problem. And just to state that informally, what this is if the program loses a game of chess in move 60, you’re actually not quite sure of all the moves he made which ones were the right moves and which ones were the bad moves, and so maybe it’s because you’ve blundered on move No. 23 and then everything else you did may have been perfect, but because you made a mistake on move 23 in your game of chess, you eventually end up losing on move 60.

So just to define very loosely for the assignment problem is whether you get a positive or negative reward, so figure out what you actually did right or did wrong to cause the reward so you can do more of the right things and less of the wrong things. And this is sort of one of the things that makes reinforcement learning hard.

And in the same way, if the helicopter crashes, you may not know. And in the same way, if the helicopter crashes, it may be something you did many minutes ago that causes the helicopter to crash. In fact, if you ever crash a car – and hopefully none of you ever get in a car accident – but when someone crashes a car, usually the things they’re doing right before they crash is step on the brakes to slow the car down before the impact. And usually stepping on the brakes does not cause a crash. Rather it makes the crash sort of hurt less.

But so, reinforcement algorithm, you see this pattern in that you step on the brakes, you crash, it’s not the reason you crash and it’s hard to figure out that it’s not actually your stepping on the brakes that caused the crash, but something you did long before that.

So let me go ahead and define the – formalize the reinforcement learning problem more, and as a preface, let me say algorithms are applied to a broad range of problems, but because robotics videos are easy to show in the lecture – I have a lot of them – throughout this lecture I use a bunch of robotics for examples, but later, we’ll talk about applications of these ideas, so broader ranges of problems as well. But the basic problem I’m facing is sequential decision making. We need to make many decisions and where your decisions perhaps have long term consequences.

So let's formalize the reinforcement learning problem. Reinforcement learning problems model the worlds using something called the MDP or the Markov Decision Process formalism. And let's see, MDP is a five tuple – I don't have enough space – well, comprising five things.

So let me say what these are. Actually, could you please raise the screen? I won't need the laptop anymore today. [Inaudible] more space. Yep, go, great. Thanks.

So an MDP comprises a five tuple. The first of these elements, s is a set of states and so for the helicopter example, the set of states would be the possible positions and orientations of a helicopter. A is a set of actions. So again, for the helicopter example, this would be the set of all possible positions that we could put our control sticks into. P, s, a are state transition distributions. So for each state and each action, this is a high probability distribution, so sum over s prime, Psa, s prime equals 1 and Psa s prime is created over zero.

And state transition distributions are – or state transition probabilities work as follows. P subscript (s, a) gives me the probability distribution over what state I will transition to, what state I wind up in, if I take an action a in a state s. So this is probability distribution over states s prime and then I get to when I take an action a in the state s. Now I'll read this in a second.

Gamma is the number called the discount factor. Don't worry about this yet. I'll say what this is in a second. And there's usually a number strictly rated, strictly less than 1 and rated equal to zero. And R is our reward function, so the reward function maps from the set of states to the real numbers and can be positive or negative. This is the set of real numbers.

So just to make these elements concrete, let me give a specific example of a MDP. Rather than talking about something as complicated as helicopters, I'm going to use a much smaller MDP as the running example for the rest of today's lecture. We'll look at much more complicated MDPs in subsequent lectures.

This is an example that I adapted from a textbook by Stuart Russell and Peter Norvig called Artificial Intelligence: A Modern Approach (Second Edition). And this is a small MDP that models a robot navigation task in which if you imagine you have a robot that lives all over the grid world where the shaded-in cell is an obstacle, so the robot can't go over this cell.

And so, let's see. I would really like the robot to get to this upper right north cell, let's say, so I'm going to associate that cell with a +1 reward, and I'd really like it to avoid that grid cell, so I'm gonna associate that grid cell with -1 reward.

So let's actually iterate through the five elements of the MDP and so see what they are for this problem. So the robot can be in any of these eleven positions and so I have an MDP with 11 states, and it's a set capital S corresponding to the 11 places it could be in.

And let's say my robot in this set, highly simplified for a logical example, can try to move in each of the compass directions, so in this MDP, I'll have four actions corresponding to moving in each of the North, South, East, West compass directions.

And let's see. Let's say that my robot's dynamics are noisy. If you've worked in robotics before, you know that if you command a robot to go North, because of wheel slip or a core design in how you act or whatever, there's a small chance that your robot will be off side here. So you command your robot to move forward one meter, usually it will move forward somewhere between like 95 centimeters or to 105 centimeters.

So in this highly simplified grid world, I'm going to model the stochastic dynamics of my robot as follows. I'm going to say that if you command the robot to go north, there's actually a 10 percent chance that it will accidentally veer off to the left and a 10 percent chance it will veer off to the right and only a .8 chance that it will manage to go in the direction you commanded it. This is sort of a crude model, wheels slipping on the model robot. And if the robot bounces off a wall, then it just stays where it is and nothing happens.

So let's see. Concretely, we would write this down using the state transition probability. So for example, let's take the state – let me call it a free comma one state and let's say you command the robot to go north. To specify these noisy dynamics of the robot, you would write down state transition probabilities for the robot as follows. You say that if you're in the state free one and you take the action north, your chance of getting to free two is 0.8. If you're in the state of free one and you take the action north, the chance of getting to four 1 is open 1 and so on. And so on, okay?

This last line is that if you're in the state free one and you take the action north, the chance of you getting to the state free free is zero. And this is your chance of transitioning in one-time sets of the state free free is equal to zero. So these are the state transition probabilities for my MDP.

Let's see. The last two elements of my five tuple are gamma and the reward function. Let's not worry about gamma for now, but my reward function would be as follows, so I really want the robot to get to the fourth – I'm using four comma free. It's indexing to the states by using the numbers I wrote at the sides of the grid.

So my reward for getting to the fourth free state is +1 and my reward for getting to the fourth 2-state is -1, and as is common practice – let's see. As is fairly common practice in navigation tasks, for all other states, the terminal states, I'm going to associate sort of a small negative reward and you can think of this as a small negative reward that charges my robot for his battery consumption or his fuel consumption for one move around. And so a small negative reward like this that charges the robot for running around randomly tends to cause the system to compute solutions that don't waste time and make its way to the goal as quickly as possible because it's charged for fuel consumption.

Okay. So, well, let me just mention, there's actually one other complication that I'm gonna sort of not worry about. In this specific example, unless you're going to assume that when the robot gets to the +1 or the -1 reward, then the world ends and so you get to the +1 and then that's it. The world ends. There are no more rewards positive or negative after that, right? And so there are various ways to model that. One way to think about that is you may imagine there's actually a 12th state, something that's called the zero cost absorbing state, so that whenever you get to the +1 or the -1, you then transition the probability one to this 12th state and you stay in this 12th state forever with no more rewards. I just mention that, that when you get to the +1 or -1, think of the problems in finishing. The reason I do that is because it makes some of the numbers come up nicer and be easier to understand. It's the sort of state where you go in where sometimes you hear the term zero cost absorbing states. It's another state so that when you enter that state, there are no more rewards; you always stay in that state forever.

All right. So let's just see how an MDP works and it works as follows. At time 0, your robot starts off at some state as 0 and, depending on where you are, you get to choose an action a0 to decide to go North, South, East, or West. Depending on your choice, you get to some state s1, which is going to be randomly drawn from the state transition distribution index by state 0 and the action you just chose. So the next state you get to depends – well, it depends in the probabilistic way on the previous state and the action you just took. After you get to the state s1, you get to choose a new action a1, and then as a result of that, you get to some new state s2 sort of randomly from the state transition distributions and so on. Okay?

So after your robot does this for a while, it will have visited some sequence of states s0, s1, s2, and so on, and to evaluate how well we did, we'll take the reward function and we'll apply it to the sequence of states and add up the sum of rewards that your robot obtained on the sequence of states it visited. State s0 is your action, you get to s1, take an action, you get to s2 and so on. So you keep the reward function in the pile to every state in the sequence and this is the sum of rewards you obtain. Let me show you just one more bit. You can multiply this by gamma, gamma squared, and the next term will be multiplied by gamma cubed and so on. And this is called – I'm going to call this the Total Payoff for the sequence of states s0, s1, s2, and so on that your robot visited. And so let me also say what gamma is. See the quality gamma is a number that's usually less than one. It usually you think of gamma as a number like open 99. So the effect of gamma is that the reward you obtain at time 1 is given a slightly smaller weight than the reward you get at time zero. And then the reward you get at time 2 is even a little bit smaller than the reward you get at a previous time set and so on. Let's see. And so if this is an economic application, if you're in like stock market trading with Gaussian algorithm or whatever, this is an economic application, then your rewards are dollars earned and lost. Then to this kind of factor, gamma has a very natural interpretation as the time value of money because like a dollar today is worth slightly less than – excuse me, the dollar today is worth slightly more than the dollar tomorrow because the dollar in the bank can earn a little bit of interest. And conversely, having to pay out a dollar tomorrow is better than having to pay out a dollar today. So in other words, the effect of this compacted gamma tends to weight wins or losses in the future less than wins or losses in the immediate

future – tends to weight wins and losses in the distant future less than wins and losses in the immediate. And so the girth of the reinforcement learning algorithm is to choose actions over time, to choose actions a0, a1 and so on to try to maximize the expected value of this total payoff. And more concretely, what we will try to do is have our reinforcement learning algorithms compute a policy, which I denote by the lower case p, which – and all a policy is, a definition of a policy is a function mapping from the states of the actions and so it goes to kind of a policy that tells us so for every state, what action it recommends we take in that state. So concretely, here is an example of a policy. And this actually turns out to be the optimal policy for the MDP and I'll tell you later how I computed this. And so this is an example of a policy. A policy is just a mapping from the states to the actions, and so our policy tells me when you're in this state, you should take the left action and so on. And this particular policy I drew out happens to be after a policy in the sense that when you execute this policy, this will maximize your expected value of the total payoff. This will maximize your expected total sum of the counter rewards.

Student:[Inaudible]

Instructor (Andrew Ng):Yeah, so can policy be over multiple states, can it be over – so can it be a function of not only current state, but the state I was in previously as well. So the answer is yes. Sometimes people call them strategies instead of policies, but usually you're going to use policies. It actually turns out that for an MDP, you're allowing policies that depend on my previous states, will not allow you to do any better. At least in the limited context we're talking about. So in other words, there exists a policy that only ever lets the current state ever maximize my expected total payoff. And this statement won't be true for some of the richer models we talk about later, but for now, all we need to do is – this suffices to just look at the current states and actions.

And sometimes they use the term executable policy to mean that I'm going to take actions according to the policies, so I'm going to execute the policy p. That means I'm going to – whenever some state s, I'm going to take the action that the policy p outputs when given the current state.

All right. So it turns out that one of the things MDPs are very good at is – all right, let's look at our states. Say the optimal policy in this state is to go left. There's actually – this probably wasn't very obvious. Why is it that you have actions that go left take a longer path there? The alternative would be to go north and try to find a much shorter path to the +1 state, but when you're in this state over here, this, I guess, free comma 2 state, when in that state over there, when you go north, there's a .1 chance you accidentally veer off to the right to the -1 state. And so there will be subtle tradeoffs. Is it better to take the longer, safer route, but the discount factor tends to discourage that and the .02 charge per step will tend to discourage that. Or is it better to take a shorter, riskier route.

And so it wasn't obvious to me until I computed it, but just to see also action and this is one of those things that MDP machinery is very good at making, to make subtle tradeoffs to make these optimal.

So what I want to do next is make a few more definitions and that will lead us to our first algorithm for computing optimal policies and MDPs, so finding optimal ways to act on MDPs. Before I move on, let's check for any questions about the MDP formalism. Okay, cool.

So let's now talk about how we actually go about computing optimal policy like that and to get there, I need to define a few things. So just as a preview of the next moves I'm gonna take, I'm going to define something called Vp and then I'm going to define V* and then I'm going to define p*. And it will be a consequence of my definitions that p* is the optimal policy.

And so I'm going to say as I define these things, keep in mind what is a definition and what is a consequence of a definition. In particular, I won't be defining p* to be the optimal policy, but I'll define p* by a different equation and it will be a consequence of my definition that p* is the optimal policy.

The first one I want to define is Vp, so for any given policy p, for any policy p, I'm going to define the value function Vp and sometimes I call this the value function for p. So I want to find the value function for Vp, the function mapping from the state's known numbers, such that Vp(s) is the expected payoff – is the expected total payoff if you started in the state s and execute p. So in other words, Vp(s) is equal to the expected value of this here, sum of this counted rewards, the total payoff, given that you execute the policy p and the first state in the sequence is zero, is that state s.

I say this is slightly sloppy probabilistic notation, so p isn't really in around the variable, so maybe I shouldn't actually be conditioning on p. This is sort of moderately standard notation horror, so we use the steady sloppy policy notation.

So as a concrete example, here's a policy and this is not a great policy. This is just some policy p. It's actually a pretty bad policy that for many states, seems to be heading to the -1 rather than the +1. And so the value function is the function mapping from the states of known numbers, so it associates each state with a number and in this case, this is Vp. So that's the value function for this policy.

And so you notice, for instance, that for all the states in the bottom two rows, I guess, this is a really bad policy that has a high chance to take you to the -1 state and so all the values for those states in the bottom two rows are negative. So in this expectation, your total payoff would be negative. And if you execute this rather bad policy, you start in any of the states in the bottom row, and if you start in the top row, the total payoff would be positive. This is not a terribly bad policy if it stays in the topmost row.

And so given any policy, you can write down a value function for that policy. If some of you are still writing, I'll leave that up for a second while I clean another couple of boards.

Okay. So the circumstances of the following, Vp(s) is equal to – well, the expected value of R if s is zero, which is the reward you get right away for just being in the initial state s,

plus – let me write this like this – I'm going to write gamma and then R if s1 plus gamma, R of s2 plus dot, dot, dot, what condition of p. Okay? So just de-parenthesize these.

This first term here, this is sometimes called the immediate reward. This is the reward you get right away just for starting in the state at zero. And then the second term here, these are sometimes called the future reward, which is the rewards you get sort of one time step into the future and what I want you to note is what this term is. That term there is really just the value function for the state s1 because this term here in parentheses, this is really – suppose I was to start in the state s1 or this is the sum of the counter rewards I would get if I were to start in the state s1. So my immediate reward for starting in s1 would be R (s1), then plus gamma times additional future rewards in the future.

And so it turns out you can write VRp recursively in terms of itself. And presume that VRp is equal to the immediate reward plus gamma times – actually, let me write – it would just be mapped as notation – s0 gets mapped to s and s1 gets mapped to s prime and it just makes it all right.

So value function for p to stay zero is the immediate reward plus this current factor gamma times – and now you have Vp of s1. Right here is Vp of s prime. But s prime is a random variable because the next state you get to after one time set is random and so in particular, taking expectations, this is the sum of all states s prime of your probability of getting to that state times that. And just to be clear on this notation, right, this is P subscript (s, a) of s prime is the chance of you getting to the state s prime when you take the action a in state s.

And in this case, we're executing the policy p and so this is P(s) p(s) because the action we're going to take in state s is the action p reverse. So this is – in other words, this Ps p(s), this distribution overstates s prime that you would transitioned to, the one time step, when you take the action p(s) in the state s.

So just to give this a name, this equation is called Bellman's equations and is one of the central equations that we'll use over and over when we solve MDPs. Raise your hand if this equation makes sense. Some of you didn't raise your hands. Do you have questions?

So let's try to say this again. Actually, which of the symbols don't make sense for those of you that didn't raise your hands? You're regretting not raising your hand now, aren't you? Let's try saying this one more time and maybe it will come clear later. So what is it? So this equation is sort of like my value at the current state is equal to R(s) plus gamma times – and depending on what state I get to next, my expected total payoff from the state s prime is Vp of s prime, whereas prime is the state I get to after one time step. So incurring one state as I'm going to take some action and I get to some state that's prime and this equation is sort of my expected total payoff for executing the policy p from the state s.

But s prime is random because the next state I get to is random and well, we'll use the next board. The chance I get to some specific state as prime is given by we have a P subscript (s, a), s prime, where – because these are just [inaudible] and probabilities – where the action a I chose is given by p(s) because I'm executing the action a in the current state s. And so when you plug this back in, you get P subscript s R(s) as prime, just gives me the distribution over the states in making the transition to it in one step, and hence, that just needs Bellman's equations.

So since Bellman's equations gives you a way to solve for the value function for policy in closed form. So again, the problem is suppose I'm given a fixed policy. How do I solve for Vp? How do I solve for the – so given fixed policy, how do I solve for this equation?

It turns out, Bellman's equation gives you a way of doing this, so coming back to this board, it turns out to be – sort of coming back to the previous boards, around – could you move the camera to point to this board? Okay, cool. So going back to this board, we work the problem's equation, this equation, right, let's say I have a fixed policy p and I want to solve for the value function for the policy p. Then what this equation is just imposes a set of linear constraints on the value function. So in particular, this says that the value for a given state is equal to some constant, and then some linear function of other values.

And so you can write down one such equation for every state in your MDP and this imposes a set of linear constraints on what the value function could be. And then it turns out that by solving the resulting linear system equations, you can then solve for the value function Vp(s). There's a high level description. Let me now make this concrete.

So specifically, let me take the free one state, that state we're using as an example. So Bellman's equation tells me that the value for p for the free one state – oh, and let's say I have a specific policy so that p are free one – let's say it takes a North action, which is not the ultimate action. For this policy, Bellman's equation tells me that Vp of free one is equal to R of the state free one, and then plus gamma times our trans 0.8 I get to the free two state, which translates .1 and gets to the four one state and which times 0.1, I will get to the two one state.

And so what I've done is I've written down Bellman's equations for the free one state. I hope you know what it means. It's in my low MDP; I'm indexing the states 1, 2, 3, 4, so this state over there where I drew the circle is the free one state.

So for every one of my 11 states in the MDP, I can write down an equation like this. This stands just for one state. And you notice that if I'm trying to solve for the values, so these are the unknowns, then I will have 11 variables because I'm trying to solve for the value function for each of my 11 states, and I will have 11 constraints because I can write down 11 equations of this form, one such equation for each one of my states.

So if you do this, if you write down this sort of equation for every one of your states and then do these, you have your set of linear equations with 11 unknowns and 11 variables – excuse me, 11 constraints or 11 equations with 11 unknowns, and so you can solve that

linear system of equations to get an explicit solution for Vp. So if you have n states, you end up with n equations and n unknowns and solve that to get the values for all of your states.

Okay, cool. So actually, could you just raise your hand if this made sense? Cool.

All right, so that was the value function for specific policy and how to solve for it. Let me define one more thing. So the optimal value function when defined as V*(s) equals max over all policies p of Vp(s). So in other words, for any given state s, the optimal value function says suppose I take a max over all possible policies p, what is the best possible expected – some of the counter rewards that I can expect to get? Or what is my optimal expected total payoff for starting at state s, so taking a max over all possible control policies p.

So it turns out that there's a version of Bellman's equations for V* as well and so this is also called Bellman's equations for V* rather than for Vp and I'll just write that down. So this says that the optimal payoff you can get from the state s is equal to – so our [inaudible] are multi here, so let's see. Just for starting off in the state s, you're going to get your immediate R(s) and then depending on what action a you take your expected total payoff will be given by this. So if I take an action a in some state s, then with probability given by P subscript (s; a) of s prime, by this probability our transition of state s prime, and when we get to the state s prime, I'll expect my total payoff from there to be given by V*(s) prime because I'm now starting to use the s prime.

So the only thing in this equation I need to fill in is where is the action a, so in order to actually obtain the optimal expected payoff, and to actually obtain the maximum or the optimal expected total payoff, what you should choose here is the max over our actions a, choose your action a that maximizes the expected value of your total payoffs as well.

So it just makes sense. There's a version of Bellman's equations for V* rather than Vp and I'll just say it again. It says that my optimal expected total payoff is my immediate reward plus, and then the best action it can choose, the max over all actions a of my expected future payoff.

And these also lead to my definition of p*, which is let's say I'm in some state s and I want to know what action to choose. Well, if I'm in some state s, I'm gonna get here an immediate R(s) anyway, so what's the best action for me to choose is whatever action will enable me to maximize the second term, as well as if my robot is in some state s and it wants to know what action to choose, I want to choose the action that will maximize my expected total payoff and so p*(s) is going to define as R(max) over actions a of this same thing.

I could also put the gamma there, but gamma is just a positive. Gamma is almost always positive, so I just drop that because it's just a constant scale you go through and doesn't affect the R(max).

And so, the consequence of this definition is that p* is actually the optimal policy because p* will maximize my expected total payoffs.

Cool. Any questions at this point? Cool. So what I'd like to do now is talk about how algorithms actually compute high start, compute the optimal policy. I should write down a little bit more before I do that, but notice that if I can compute V*, if I can compute the optimal value function, then I can plug it into this equation and then I'll be done. So if I can compute V*, then you are using this definition for p* and can compute the optimal policy.

So my strategy for computing the optimal policy will be to compute V* and then plug it into this equation and that will give me the optimal policy p*. So my goal, my next goal, will really be to compute V*.

But the definition of V* here doesn't lead to a nice algorithm for computing it because let's see – so I know how to compute Vp for any given policy p by solving that linear system equation, but there's an exponentially large number of policies, so you get 11 states and four actions and what the number of policies is froze to the par of 11. This is of a huge space of possible policies and so I can't actually exhaust the union of all policies and then take a max on [inaudible].

So I should write down some other things first, just to ground the notations, but what I'll do is eventually come up with an algorithm for computing V*, the optimal value function and then we'll plug them into this and that will give us the optimal policy p*.

And so I'll write down the algorithm in a second, but just to ground the notation, well – yeah, let's skip that. Let's just talk about the algorithm. So this is an algorithm called value iteration and it makes use of Bellman's equations for the optimal policy to compute V*. So here's the algorithm. Okay, and that's the entirety of the algorithm and oh, you repeat the step, I guess. You repeatedly do this step.

So just to be concrete, let's say in my MDP of 11 states, the first step is initialize V(s) equals zero, so what that means is I create an array in computer implementation, create an array of 11 elements and say set all of them to zero. Says I can initialize into anything. It doesn't really matter.

And now what I'm going to do is I'll take Bellman's equations and we'll keep on taking the right hand side of Bellman's equations and overwriting and start copying down the left hand side. So we'll essentially iteratively try to make Bellman's equations hold true for the numbers V(s) that are stored along the way. So V(s) here is in the array of 11 elements and I'm going to repeatedly compute the right hand side and copy that onto V(s).

And it turns out that when you do this, this will make V(s) converge to V*(s), so it may be of no surprise because we know V* [inaudible] set inside Bellman's equations.

Just to tell you, some of these ideas that they get more than the problem says, so I won't prove the conversions of this algorithm. Some implementation details, it turns out there's two ways you can do this update. One is when I say for every state s that has performed this update, one way you can do this is for every state s, you can compute the right hand side and then you can simultaneously overwrite the left hand side for every state s. And so if you do that, that's called a sequence update. Right and sequence [inaudible], so update all the states s simultaneously.

And if you do that, it's sometimes written as follows. If you do synchronous update, then it's as if you have some value function, you're at the Ith iteration or Tth iteration of the algorithm and then you're going to compute some function of your entire value function, and then you get to set your value function to your new version, so simultaneously update all 11 values in your s space value function.

So it's sometimes written like this. My B here is called the Bellman backup operator, so the synchronized valuation you sort of take the value function, you apply the Bellman backup operator to it and then the Bellman backup operator just means computing the right hand side of this for all the states and you've overwritten your entire value function.

The only way of performing these updates is asynchronous updates, which is where you update the states one at a time. So you go through the states in some fixed order, so would update V(s) for state No. 1 and then I would like to update V(s) for state No. 2, then state No. 3, and so on. And when I'm updating V(s) for state No. 5, if V(s) prime, if I end up using the values for states 1, 2, 3, and 4 on the right hand side, then I'd use my recently updated values on the right hand side. So as you update sequentially, when you're updating in the fifth state, you'd be using values, new values, for states 1, 2, 3, and 4. And that's called an asynchronous update.

Other versions will cause V(s) conversion to be *(s). In synchronized updates, it makes them just a tiny little bit faster [inaudible] and then it turns out the analysis of value iterations synchronous updates are also easier to analyze and that just matters [inaudible]. Asynchronous has been just a little bit faster.

So when you run this algorithm on the MDP – I forgot to say all these values were computed with gamma equals open 99 and actually, Roger Gross, who's a, I guess, master [inaudible] helped me with computing some of these numbers. So you compute it. That way you run value relation on this MDP. The numbers you get for V* are as follows: .86, .90 – again, the numbers sort of don't matter that much, but just take a look at it and make sure it intuitively makes sense.

And then when you plug those in to the formula for computing, that I wrote down earlier, for computing p* as a function of V*, then – well, I drew this previously, but here's the optimal policy p*.

And so, just to summarize, the process is run value iteration to compute V*, so this would be this table of numbers, and then I use my form of p* to compute the optimal policy, which is this policy in this case.

Now, to be just completely concrete, let's look at that free one state again. Is it better to go left or is it better to go north? So let me just illustrate why I'd rather go left than north. In the form of the p*, if I go west, then sum over s prime, P(s, a) s prime, P*(sp), this would be – well, let me just write this down. Right, if I go north, then it would be because of that. I wrote it down really quickly, so it's messy writing. The way I got these numbers is suppose I'm in this state, in this free one state. If I choose to go west and with chance .8, I get to .75 – to this table -- .75. With chance .1, I veer off and get to the .69, then at chance .1, I go south and I bounce off the wall and I stay where I am.

So that's why my expected future payoff for going west is .8 times .75, plus .1 times .69, plus .1 times .71, the last .71 being if I bounce off the wall to the south and then seeing where I am, that gives you .740.

You can then repeat the same process to estimate your expected total payoff if you go north, so if you do that, with a .8 chance, you end up going north, so you get .69. With a .1 chance, you end up here and .1 chance you end up there. This map leads mentally to that expression and compute the expectation, you get .676. And so your total payoff is higher if you go west – your expected total payoff is higher if you go west than if you go north. And that's why the optimal action in this state is to go west.

So that was value iteration. It turns out there are two sort of standard algorithms for computing optimal policies in MDPs. Value iteration is one. As soon as you finish the writing. So value iteration is one and the other sort of standard algorithm for computing optimal policies in MDPs is called policy iteration. And let me – I'm just going to write this down.

In policy iteration, we initialize the policy p randomly, so it doesn't matter. It can be the policy that always goes north or the policy that takes actions random or whatever. And then we'll repeatedly do the following. Okay, so that's the algorithm.

So the algorithm has two steps. In the first step, we solve. We take the current policy p and we solve Bellman's equations to obtain Vp. So remember, earlier I said if you have a fixed policy p, then yeah, Bellman's equation defines this system of linear equations with 11 unknowns and 11 linear constraints. And so you solve that linear system equation so you get the value function for your current policy p, and by this notation, I mean just let V be the value function for policy p.

Then the second step is you update the policy. In other words, you pretend that your current guess V from the value function is indeed the optimal value function and you let p(s) be equal to that out max formula, so as to update your policy p.

And so it turns out that if you do this, then V will converge to V* and p will converge to p*, and so this is another way to find the optimal policy for MDP.

In terms of tradeoffs, it turns out that – let's see – in policy iteration, the computationally expensive step is this one. You need to solve this linear system of equations. You have n equations and n unknowns, if you have n states. And so if you have a problem with a reasonably few number of states, if you have a problem with like 11 states, you can solve the linear system equations fairly efficiently, and so policy iteration tends to work extremely well for problems with smallish numbers of states where you can actually solve those linear systems of equations efficiently.

So if you have a thousand states, anything less than that, you can solve a system of a thousand equations very efficiently, so policy iteration will often work fine. If you have an MDP with an enormous number of states, so we'll actually often see MDPs with tens of thousands or hundreds of thousands or millions or tens of millions of states. If you have a problem with 10 million states and you try to apply policy iteration, then this step requires solving the linear system of 10 million equations and this would be computationally expensive. And so for these really, really large MDPs, I tend to use value iteration.

Let's see. Any questions about this?

Student: So this is a convex function where – that it could be good in local optimization scheme.

Instructor (Andrew Ng):Ah, yes, you're right. That's a good question: Is this a convex function? It actually turns out that there is a way to pose a problem of solving for V* as a convex optimization problem, as a linear program. For instance, I can break down the solution – you write down V* as a solution, so linear would be the only problem you can solve. Policy iteration converges as gamma T conversion. We're not just stuck with local optimal, but the proof of the conversions of policy iteration sort of uses somewhat different principles in convex optimization. At least the versions as far as I can see, yeah. You could probably relate this back to convex optimization, but not understand the principle of why this often converges.

The proof is not that difficult, but it is also sort of longer than I want to go over in this class. Yeah, that was a good point. Cool. Actually, any questions for any of these?

Okay, so we now have two algorithms for solving MDP. There's a given, the five tuple, given the set of states, the set of actions, the state transition properties, the discount factor, and the reward function, you can now apply policy iteration or value iteration to compute the optimal policy for the MDP.

The last thing I want to talk about is what if you don't know the state transition probabilities, and sometimes you won't know the reward function R as well, but let's leave that aside. And so for example, let's say you're trying to fly a helicopter and you

don’t really know in advance what state your helicopter will transition to and take an action in a certain state, because helicopter dynamics are kind of noisy. You sort of often don’t really know what state you end up in.

So the standard thing to do, or one standard thing to do, is then to try to estimate the state transition probabilities from data. Let me just write this out. It turns out that the MDP has its 5 tuple, right? S, A; you have the transition probabilities, gamma, and R. S and A you almost always know. The state space is sort of up to you to define. What’s the state space at the very bottom, factor you’re trying to control, whatever. Actions is, again, just one of your actions. Usually, we almost always know these. Gamma, the discount factor is something you choose depending on how much you want to trade off current versus future rewards. The reward function you usually know. There are some exceptional cases. Usually, you come up with a reward function and so you usually know what the reward function is. Sometimes you don’t, but let’s just leave that aside for now and the most common thing for you to have to learn are the state transition probabilities. So we’ll just talk about how to learn that. So when you don’t know state transition probabilities, the most common thing to do is just estimate it from data. So what I mean is imagine some robot – maybe it’s a robot roaming around the hallway, like in that grid example – you would then have the robot just take actions in the MDP and you would then estimate your state transition probabilities P subscript (s, a) s prime to be – pretty much exactly what you’d expect it to be.

This would be the number of times you took action a in state s and you got to s prime, divided by the number of times you took action a in state s. Okay? So the estimate of this is just all the times you took the action a in the state s, what’s the fraction of times you actually got to the state s prime. It’s pretty much exactly what you expect it to be. Or you can – or in case you’ve never actually tried action a in state s, so if this turns out to be 0 over 0, you can then have some default estimate for those vector uniform distribution over all states, this reasonable default.

And so, putting it all together – and by the way, it turns out in reinforcement learning, in most of the earlier parts of this class where we did supervised learning, I sort of talked about the logistic regression algorithm, so it does the algorithm and most implementations of logistic regression – like a fairly standard way to do logistic regression were SVMs or faster analysis or whatever. It turns out in reinforcement learning there’s more of a mix and match sense, I guess, so there are often different pieces of different algorithms you can choose to use. So in some of the algorithms I write down, there’s sort of more than one way to do it and I’m sort of giving specific examples, but if you’re faced with an AI problem, some of you in control of robots, you want to plug in value iteration here instead of policy iteration. You want to do something slightly different than one of the specific things I wrote down. That’s actually fairly common, so just in reinforcement learning, there’s sort of other major ways to apply different algorithms and mix and match different algorithms. And this will come up again in the weekly lectures. So just putting the things I said together, here would be a – now this would be an example of how you might estimate the state transition probabilities in a MDP and find the policy for it. So you might repeatedly do the following. Let’s see. Take

actions using some policy p to get experience in the MDP, meaning that just execute the policy p observed state transitions. Based on the data you get, you then update estimates of your state transition probabilities P subscript (s, a) based on the experience of the observations you just got. Then you might solve Bellman's equations using value iterations, which I'm abbreviating to VI, and by Bellman's equations, I mean Bellman's equations for V*, not for Vp. Solve Bellman's equations using value iteration to get an estimate for P* and then you update your policy by events equals [inaudible].

And now you have a new policy so you can then go back and execute this policy for a bit more of the MDPs to get some more observations of state transitions, get the noisy ones in MDP, use that update to estimate your state transition probabilities again; use value iteration or policy iteration to solve for [inaudible] the value function, get a new policy and so on. Okay? And it turns out when you do this, I actually wrote down value iteration for a reason. It turns out in the third step of the algorithm, if you're using value iteration rather than policy iteration, to initialize value iteration, if you use your solution from the previous used algorithm, right, then that's a very good initialization condition and this will tend to converge much more quickly because value iteration tries to solve for V(s) for every state s. It tries to estimate V*(s) and the s from the * in V(s) and so if you're looking through this and you initialize your value iteration algorithm using the values you have from the previous round through this, then that will often make this converge faster.

But again, this is again here, you can also adjust a small part in policy iteration in here as well and whatever, and this is a fairly typical example of how you would solve a policy, correct digits and then key in and try to find a good policy for a problem for which you did not know the state transition probabilities in advance.

Cool. Questions about this? Cool. So that sure was exciting. This is like our first two MDP algorithms in just one lecture. All right, let's close for today. Thanks.

[End of Audio]

Duration: 73 minutes