MachineLearning-Lecture20

Instructor (Andrew Ng):Okay. Good morning. Just one quick announcement before I start. Poster session, next Wednesday, 8:30 as you already know, and poster boards will be made available soon, so the poster boards we have are 20 inches by 30 inches in case you want to start designing your posters. That's 20 inches by 30 inches. And they will be available this Friday, and you can pick them up from Nicki Salgudo who's in Gates 187, so starting this Friday. I'll send out this information by e-mail as well, in case you don't want to write it down.

For those you that are SCPD students, if you want to show up here only on Wednesday for the poster session itself, we'll also have blank posters there, or you're also welcome to buy your own poster boards. If you do take poster boards from us then please treat them well. For the sake of the environment, we do ask you to give them back at the end of the poster session. We'll recycle them from year to year. So if you do take one from us, please don't cut holes in it or anything. So welcome to the last lecture of this course. What I want to do today is tell you about one final class of reinforcement learning algorithms. I just want to say a little bit about POMDPs, the partially observable MDPs, and then the main technical topic for today will be policy search algorithms. I'll talk about two specific algorithms, essentially called reinforced and called Pegasus, and then we'll wrap up the class. So if you recall from the last lecture, I actually started to talk about one specific example of a POMDP, which was this sort of linear dynamical system. This is sort of LQR, linear quadratic revelation problem, but I changed it and said what if we only have observations YT. And what if we couldn't observe the state of the system directly, but had to choose an action only based on some noisy observations that maybe some function of the state?

So our strategy last time was that we said that in the fully observable case, we could choose actions – AT equals two, that matrix LT times ST. So LT was this matrix of parameters that [inaudible] describe the dynamic programming algorithm for finite horizon MDPs in the LQR problem. And so we said if only we knew what the state was, we choose actions according to some matrix LT times the state. And then I said in the partially observable case, we would compute these estimates. I wrote them as S of T given T, which were our best estimate for what the state is given all the observations. And in particular, I'm gonna talk about a Kalman filter which we worked out that our posterior distribution of what the state is given all the observations up to a certain time that was this.

So this is from last time. So that given the observations Y one through YT, our posterior distribution of the current state ST was Gaussian would mean ST given T sigma T given T. So I said we use a Kalman filter to compute this thing, this ST given T, which is going to be our best guess for what the state is currently. And then we choose actions using our estimate for what the state is, rather than using the true state because we don't know the true state anymore in this POMDP. So it turns out that this specific strategy actually allows you to choose optimal actions, allows you to choose actions as well as you possibly can given that this is a POMDP, and given there are these noisy observations. It turns out that in general finding optimal policies with POMDPs – finding optimal policies for these sorts of partially observable MDPs is an NP-hard problem. Just to be concrete about the formalism of the POMDP – I should just write it here – a POMDP formally is a tuple like that where the changes are the set Y is the set of possible observations, and this O subscript S are the observation distributions. And so at each step, we observe – at each step in the POMDP, if we're in some state ST, we observe some observation YT drawn from the observation distribution O subscript ST, that there's an index by what the current state is. And it turns out that computing the optimal policy in a POMDP is an NP-hard problem. For the specific case of linear dynamical systems with the Kalman filter model, we have this strategy of computing the optimal policy assuming full observability and then estimating the states from the observations, and then plugging the two together.

That turns out to be optimal essentially for only that special case of a POMDP. In the more general case, that strategy of designing a controller assuming full observability and then just estimating the state and plugging the two together, for general POMDPs that same strategy is often a very reasonable strategy but is not always guaranteed to be optimal. Solving these problems in general, NP-hard. So what I want to do today is actually talk about a different class of reinforcement learning algorithms. These are called policy search algorithms. In particular, policy search algorithms can be applied equally well to MDPs, to fully observed Markov decision processes, or to these POMDPs, or to these partially observable MPDs. What I want to do now, I'll actually just describe policy search algorithms applied to MDPs, applied to the fully observable case. And in the end, I just briefly describe how you can take policy search algorithms and apply them to POMDPs. In the latter case, when you apply a policy search algorithm to a POMDP, it's going to be hard to guarantee that you get the globally optimal policy because solving POMDPs in general is NP-hard, but nonetheless policy search algorithms – it turns out to be I think one of the most effective classes of reinforcement learning algorithms, as well both for MDPs and for POMDPs.

So here's what we're going to do. In policy search, we're going to define of some set which I denote capital pi of policies, and our strategy is to search for a good policy lower pi into set capital pi. Just by analogy, I want to say – in the same way, back when we were talking about supervised learning, the way we defined the set capital pi of policies in the search for policy in this set capital pi is analogous to supervised learning where we defined a set script H of hypotheses and search – and would search for a good hypothesis in this policy script H. Policy search is sometimes also called direct policy search. To contrast this with the source of algorithms we've been talking about so far, in all the algorithms we've been talking about so far, we would try to find V star. We would try to find the optimal value function. And then we'd use V star – we'd use the optimal value function to then try to compute or try to approximate pi star. So all the approaches we talked about previously are strategy for finding a good policy. Once we compute the value function, then we go from that to policy. In contrast, in policy search algorithms and something that's called direct policy search algorithms, the idea is that we're going to quote "directly" try to approximate a good policy without going through the intermediate stage of trying to find the value function. Let's see. And also as I develop policy search – just one step that's sometimes slightly confusing. Making an analogy to supervised learning again, when we talked about logistic regression, I said we have input features X and some labels Y, and I sort of said let's approximate Y using the logistic function of the inputs X. And at least initially, the logistic function was sort of pulled out of the air.

In the same way, as I define policy search algorithms, there'll sort of be a step where I say, "Well, let's try to compute the actions. Let's try to approximate what a good action is using a logistic function of the state." So again, I'll sort of pull a function out of the air. I'll say, "Let's just choose a function, and that'll be our choice of the policy cost," and I'll say, "Let's take this input the state, and then we'll map it through logistic function, and then hopefully, we'll approximate what is a good function – excuse me, we'll approximate what is a good action using a logistic function of the state." So there's that sort of – the function of the choice of policy cost that's again a little bit arbitrary, but it's arbitrary as it was when we were talking about supervised learning. So to develop our first policy search algorithm, I'm actually gonna need the new definition. So our first policy search algorithm, we'll actually need to work with stochastic policies. What I mean by stochastic policy is there's going to be a function that maps from the space of states across actions. They're real numbers where pi of S comma A will be interpreted as the probability of taking this action A in sum state S. And so we have to add sum over A – In other words, for every state a stochastic policy specifies a probability distribution over the actions. So concretely, suppose you are executing some policy pi. Say I have some stochastic policy pi. I wanna execute the policy pi. What that means is that – in this example let's say I have three actions.

What that means is that suppose I'm in some state S. I would then compute pi of S comma A1, pi of S comma A2, pi of S comma A3, if I have a three action MDP. These will be three numbers that sum up to one, and then my chance of taking action A1 will be equal to this. My chance of taking action A2 will be equal to pi of S comma A2. My chance of taking action A3 will be equal to this number. So that's what it means to execute a stochastic policy. So as a concrete example, just let me make this – the concept of why you wanna use stochastic policy is maybe a little bit hard to understand. So let me just go ahead and give one specific example of what a stochastic policy may look like. For this example, I'm gonna use the inverted pendulum as my motivating example. It's that problem of balancing a pole. We have an inverted pendulum that swings freely, and you want to move the cart left and right to keep the pole vertical. Let's say my actions – for today's example, I'm gonna use that angle to denote the angle of the pole phi. I have two actions where A1 is to accelerate left and A2 is to accelerate right. Actually, let me just write that the other way around. A1 is to accelerate right. A2 is to accelerate left. So let's see. Choose a reward function that penalizes the pole falling over whatever. And now let's come up with a stochastic policy for this problem. To come up with a class of stochastic policies really means coming up with some class of functions to approximate what action you want to take as a function of the state.

So here's my somewhat arbitrary choice. I'm gonna say that the probability of action A1, so pi of S comma A1, I'm gonna write as – okay? And I just chose the logistic function because it's a convenient function we've used a lot. So I'm gonna say that my policy is parameterized by a set of parameters theta, and for any given set of parameters theta, that gives me a stochastic policy. And if I’m executing that policy with parameters theta, that means that the chance of my choosing to a set of [inaudible] is given by this number. Because my chances of executing actions A1 or A2 must sum to one, this gives me pi of S A2. So just [inaudible], this means that when I’m in sum state S, I’m going to compute this number, compute one over one plus E to the minus state of transpose S. And then with this probability, I will execute the accelerate right action, and with one minus this probability, I’ll execute the accelerate left action. And again, just to give you a sense of why this might be a reasonable thing to do, let’s say my state vector is – this is [inaudible] state, and I added an extra one as an interceptor, just to give my logistic function an extra feature. If I choose my parameters and my policy to be say this, then that means that at any state, the probability of my taking action A1 – the probability of my taking the accelerate right action is this one over one plus E to the minus state of transpose S, which taking the inner product of theta and S, this just gives you phi, equals one over one plus E to the minus phi.

And so if I choose my parameters theta as follows, what that means is that just depending on the angle phi of my inverted pendulum, the chance of my accelerating to the right is just this function of the angle of my inverted pendulum. And so this means for example that if my inverted pendulum is leaning far over to the right, then I’m very likely to accelerate to the right to try to catch it. I hope the physics of this inverted pendulum thing make sense. If my pole’s leaning over to the right, then I wanna accelerate to the right to catch it. And conversely if phi is negative, it’s leaning over to the left, and I’ll accelerate to the left to try to catch it. So this is one example for one specific choice of parameters theta. Obviously, this isn’t a great policy because it ignores the rest of the features. Maybe if the cart is further to the right, you want it to be less likely to accelerate to the right, and you can capture that by changing one of these coefficients to take into account the actual position of the cart. And then depending on the velocity of the cart and the angle of velocity, you might want to change theta to take into account these other effects as well. Maybe if the pole’s leaning far to the right, but is actually on its way to swinging back, it’s specified to the angle of velocity, then you might be less worried about having to accelerate hard to the right. And so these are the sorts of behavior you can get by varying the parameters theta.

And so our goal is to tune the parameters theta – our goal in policy search is to tune the parameters theta so that when we execute the policy pi subscript theta, the pole stays up as long as possible. In other words, our goal is to maximize as a function of theta – our goal is to maximize the expected value of the payoff for when we execute the policy pi theta. We want to choose parameters theta to maximize that. Are there questions about the problem set up, and policy search and policy classes or anything? Yeah.

Student: In a case where we have more than two actions, would we use a different theta for each of the distributions, or still have the same parameters?

Instructor (Andrew Ng):Oh, yeah. Right. So what if we have more than two actions. It turns out you can choose almost anything you want for the policy class, but you have say a fixed number of discrete actions, I would sometimes use like a softmax parameterization. Similar to softmax regression that we saw earlier in the class, you may say that – [inaudible] out of space. You may have a set of parameters theta 1 through theta D if you have D actions and – pi equals E to the theta I transpose S over – so that would be an example of a softmax parameterization for multiple actions. It turns out that if you have continuous actions, you can actually make this be a density over the actions A and parameterized by other things as well.

But the choice of policy class is somewhat up to you, in the same way that the choice of whether we chose to use a linear function or linear function with quadratic features or whatever in supervised learning that was sort of up to us. Anything else? Yeah.

Student:[Inaudible] stochastic?

Instructor (Andrew Ng):Yes.

Student: So is it possible to [inaudible] a stochastic policy using numbers [inaudible]?

Instructor (Andrew Ng):I see. Given that MDP has stochastic transition probabilities, is it possible to use [inaudible] policies and [inaudible] the stochasticity of the state transition probabilities. The answer is yes, but for the purposes of what I want to show later, that won't be useful. But formally, it is possible. If you already have a fixed – if you have a fixed policy, then you'd be able to do that. Anything else? Yeah. No, I guess even a [inaudible] class of policy can do that, but for the derivation later, I actually need to keep it separate. Actually, could you just – I know the concept of policy search is sometimes a little confusing. Could you just raise your hand if this makes sense? Okay. Thanks. So let's talk about an algorithm. What I'm gonna talk about – the first algorithm I'm going to present is sometimes called the reinforce algorithm. What I'm going to present it turns out isn't exactly the reinforce algorithm as it was originally presented by the author Ron Williams, but it sort of captures its essence.

Here's the idea. In the sequel – in what I'm about to do, I'm going to assume that S0 is some fixed initial state. Or it turns out if S0 is drawn from some fixed initial state distribution then everything else [inaudible], but let's just say S0 is some fixed initial state. So my goal is to maximize this expected sum [inaudible]. Given the policy and whatever else, drop that. So the random variables in this expectation is a sequence of states and actions: S0, A0, S1, A1, and so on, up to ST, AT are the random variables. So let me write out this expectation explicitly as a sum over all possible state and action sequences of that – so that's what an expectation is. It's the probability of the random variables times that. Let me just expand out this probability. So the probability of seeing this exact sequence of states and actions is the probability of the MDP starting in that state. If this is a deterministic initial state, then all the probability mass would be on one state. Otherwise, there's some distribution over initial states. Then times the probability that you chose action A0 from that state as zero, and then times the probability that the MDP's transition probabilities happen to transition you to state S1 where you chose action A0 to state S0, times the probability that you chose that and so on. The last term here is that, and then times that.

So what I did was just take this probability of seeing this sequence of states and actions, and then just [inaudible] explicitly or expanded explicitly like this. It turns out later on I'm going to need to write this sum of rewards a lot, so I'm just gonna call this the payoff from now. So whenever later in this lecture I write the word payoff, I just mean this sum. So our goal is to maximize the expected payoff, so our goal is to maximize this sum. Let me actually just skip ahead. I'm going to write down what the final answer is, and then I'll come back and justify the algorithm. So here's the algorithm. This is how we're going to update the parameters of the algorithm. We're going to sample a state action sequence. The way you do this is you just take your current stochastic policy, and you execute it in the MDP. So just go ahead and start from some initial state, take a stochastic action according to your current stochastic policy, see where the state transition probably takes you, and so you just do that for T times steps, and that's how you sample the state sequence. Then you compute the payoff, and then you perform this update.

So let's go back and figure out what this algorithm is doing. Notice that this algorithm performs stochastic updates because on every step it updates data according to this thing on the right hand side. This thing on the right hand side depends very much on your payoff and on the state action sequence you saw. Your state action sequence is random, so what I want to do is figure out – so on every step, I'll sort of take a step that's chosen randomly because it depends on this random state action sequence. So what I want to do is figure out on average how does it change the parameters theta. In particular, I want to know what is the expected value of the change to the parameters. So I want to know what is the expected value of this change to my parameters theta. Our goal is to maximize the sum [inaudible] – our goal is to maximize the value of the payoff. So long as the updates on expectation are on average taking us uphill on the expected payoff, then we're happy. It turns out that this algorithm is a form of stochastic gradient ascent in which – remember when I talked about stochastic gradient descent for least squares regression, I said that you have some parameters – maybe you're trying to minimize a quadratic function. Then you may have parameters that will wander around randomly until it gets close to the optimum of the [inaudible] quadratic surface. It turns out that the reinforce algorithm will be very much like that. It will be a stochastic gradient ascent algorithm in which on every step – the step we take is a little bit random. It's determined by the random state action sequence, but on expectation this turns out to be essentially gradient ascent algorithm. And so we'll do something like this. It'll wander around randomly, but on average take you towards the optimum.

So let me go ahead and prove that now. Let's see. What I'm going to do is I'm going to derive a gradient ascent update rule for maximizing the expected payoff. Then I'll hopefully show that by deriving a gradient ascent update rule, I'll end up with this thing on expectation. So before I do the derivation, let me just remind you of the chain rule – the product rule for differentiation in which if I have a product of functions, then the derivative of the product is given by taking of the derivatives of these things one at a time. So first I differentiate with respect to F prime, leaving the other two fixed. Then I differentiate with respect to G, leaving the other two fixed. Then I differentiate with respect to H, so I get H prime leaving the other two fixed. So that's the product rule for derivatives. If you refer back to this equation where earlier we wrote out that the expected payoff by this equation, this sum over all the states of the probability times the payoff. So what I'm going to do is take the derivative of this expression with respect to the parameters theta because I want to do gradient ascent on this function. So I'm going to take the derivative of this function with respect to theta, and then try to go uphill on this function.

So using the product rule, when I take the derivative of this function with respect to theta what I get is – we'll end up with the sum of terms right there. There are a lot of terms here that depend on theta, and so what I'll end up with is I'll end up having a sum – having one term that corresponds to the derivative of this keeping everything else fixed, to have one term from the derivative of this keeping everything else fixed, and I'll have one term from the derivative of that last thing keeping everything else fixed. So just apply the product rule to this.

Let's write that down. So I have that – the derivative with respect to theta of the expected value of the payoff is – it turns out I can actually do this entire derivation in exactly four steps, but each of the steps requires a huge amount of writing, so I'll just start writing and see how that goes, but this is a four step derivation. So there's the sum over the state action sequences as we saw before. Close the bracket, and then times the payoff. So that huge amount of writing, that was just taking my previous formula and differentiating these terms that depend on theta one at a time. This was the term with the derivative of the first pi of theta S0 A0. So there's the first derivative term. There's the second one. Then you have plus dot, dot, dot, like in terms of [inaudible]. That's my last term. So that was step one of four. And so by algebra – let me just write this down and convince us all that it's true. This is the second of four steps in which it just convinced itself that if I expand out – take the sum and multiply it by that big product in front, then I get back that sum of terms I get. It's essentially – for example, when I multiply out, this product on top of this ratio, of this first fraction, then pi subscript theta S0 A0, that would cancel out this pi subscript theta S0 A0 and replace it with the derivative with respect to theta of pi theta S0 A0. So [inaudible] algebra was the second.

But that term on top is just what I worked out previously – was the joint probability of the state action sequence, and now I have that times that times the payoff. And so by the definition of expectation, this is just equal to that thing times the payoff. So this thing inside the expectation, this is exactly the step that we were taking in the inner group of our reinforce algorithm, roughly the reinforce algorithm. This proves that the expected value of our change to theta is exactly in the direction of the gradient of our expected payoff. That's how I started this whole derivation. I said let's look at our expected payoff and take the derivative of that with respect to theta. What we've proved is that on expectation, the step direction I'll take reinforce is exactly the gradient of the thing I'm trying to optimize. This shows that this algorithm is a stochastic gradient ascent algorithm.

I wrote a lot. Why don't you take a minute to look at the equations and [inaudible] check if everything makes sense. I'll erase a couple of boards and then check if you have questions after that. Questions? Could you raise your hand if this makes sense? Great.

Some of the comments – we talked about those value function approximation approaches where you approximate V star, then you go from V star to pi star. Then here was also policy search approaches, where you try to approximate the policy directly. So let’s talk briefly about when either one may be preferable.

It turns out that policy search algorithms are especially effective when you can choose a simple policy class pi. So the question really is for your problem does there exist a simple function like a linear function or a logistic function that maps from features of the state to the action that works pretty well. So the problem with the inverted pendulum – this is quite likely to be true. Going through all the different choices of parameters, you can say things like if the pole’s leaning towards the right, then accelerate towards the right to try to catch it. Thanks to the inverted pendulum, this is probably true. For lots of what’s called low level control tasks, things like driving a car, the low level reflexes of do you steer your car left to avoid another car, do you steer your car left to follow the car road, flying a helicopter, again very short time scale types of decisions – I like to think of these as decisions like trained operator for like a trained driver or a trained pilot. It would almost be a reflex, these sorts of very quick instinctive things where you map very directly from the inputs, data, and action. These are problems for which you can probably choose a reasonable policy class like a logistic function or something, and it will often work well. In contrast, if you have problems that require long multistep reasoning, so things like a game of chess where you have to reason carefully about if I do this, then they’ll do that, then they’ll do this, then they’ll do that. Those I think of as less instinctual, very high level decision making. For problems like that, I would sometimes use a value function approximation approaches instead.

Let me say more about this later. The last thing I want to do is actually tell you about – I guess just as a side comment, it turns out also that if you have POMDP, if you have a partially observable MDP – I don’t want to say too much about this – it turns out that if you only have an approximation – let’s call it S hat of the true state, and so this could be S hat equals S of T given T from Kalman filter – then you can still use these sorts of policy search algorithms where you can say pi theta of S hat comma A – There are various other ways you can use policy search algorithms for POMDPs, but this is one of them where if you only have estimates of the state, you can then choose a policy class that only looks at your estimate of the state to choose the action. By using the same way of estimating the states in both training and testing, this will usually do some – so these sorts of policy search algorithms can be applied often reasonably effectively to POMDPs as well. There’s one more algorithm I wanna talk about, but some final words on the reinforce algorithm. It turns out the reinforce algorithm often works well but is often extremely slow. So it [inaudible] works, but one thing to watch out for is that because you’re taking these gradient ascent steps that are very noisy, you’re sampling a state action sequence, and then you’re sort of taking a gradient ascent step in essentially a sort of random direction that only on expectation is correct.

The gradient ascent direction for reinforce can sometimes be a bit noisy, and so it’s not that uncommon to need like a million iterations of gradient ascent, or ten million, or 100 million iterations of gradient ascent for reinforce [inaudible], so that’s just something to watch out for. One consequence of that is in the reinforce algorithm – I shouldn't really call it reinforce. In what's essentially the reinforce algorithm, there's this step where you need to sample a state action sequence. So in principle you could do this on your own robot. If there were a robot you were trying to control, you can actually physically initialize in some state, pick an action and so on, and go from there to sample a state action sequence. But if you need to do this ten million times, you probably don't want to [inaudible] your robot ten million times. I personally have seen many more applications of reinforce in simulation. You can easily run ten thousand simulations or ten million simulations of your robot in simulation maybe, but you might not want to do that – have your robot physically repeat some action ten million times. So I personally have seen many more applications of reinforce to learn using a simulator than to actually do this on a physical device.

The last thing I wanted to do is tell you about one other algorithm, one final policy search algorithm. [Inaudible] the laptop display please. It's a policy search algorithm called Pegasus that's actually what we use on our autonomous helicopter flight things for many years. There are some other things we do now. So here's the idea. There's a reminder slide on RL formalism. There's nothing here that you don't know, but I just want to pictorially describe the RL formalism because I'll use that later. I'm gonna draw the reinforcement learning picture as follows. The initialized [inaudible] system, say a helicopter or whatever in sum state S0, you choose an action A0, and then you'll say helicopter dynamics takes you to some new state S1, you choose some other action A1, and so on. And then you have some reward function, which you reply to the sequence of states you summed out, and that's your total payoff.

So this is just a picture I wanna use to summarize the RL problem. Our goal is to maximize the expected payoff, which is this, the expected sum of the rewards. And our goal is to learn the policy, which I denote by a green box. So our policy – and I'll switch back to deterministic policies for now. So my deterministic policy will be some function mapping from the states to the actions.

As a concrete example, you imagine that in the policy search setting, you may have a linear class of policies. So you may imagine that the action A will be say a linear function of the states, and your goal is to learn the parameters of the linear function. So imagine trying to do linear progression on policies, except you're trying to optimize the reinforcement learning objective. So just [inaudible] imagine that the action A is state of transpose S, and you go and policy search this to come up with good parameters theta so as to maximize the expected payoff. That would be one setting in which this picture applies. There's the idea. Quite often we come up with a model or a simulator for the MDP, and as before a model or a simulator is just a box that takes this input some state ST, takes this input some action AT, and then outputs some [inaudible] state ST plus one that you might want to take in the MDP. This ST plus one will be a random state. It will be drawn from the random state transition probabilities of MDP. This is important. Very important, ST plus one will be a random function ST and AT. In the simulator, this is [inaudible].

So for example, for autonomous helicopter flight, you [inaudible] build a simulator using supervised learning, an algorithm like linear regression [inaudible] to linear regression, so we can get a nonlinear model of the dynamics of what ST plus one is as a random function of ST and AT. Now once you have a simulator, given any fixed policy you can quite straightforwardly evaluate any policy in a simulator. Concretely, our goal is to find the policy pi mapping from states to actions, so the goal is to find the green box like that. It works well. So if you have any one fixed policy pi, you can evaluate the policy pi just using the simulator via the picture shown at the bottom of the slide. So concretely, you can take your initial state S0, feed it into the policy pi, your policy pi will output some action A0, you plug it in the simulator, the simulator outputs a random state S1, you feed S1 into the policy and so on, and you get a sequence of states S0 through ST that your helicopter flies through in simulation. Then sum up the rewards, and this gives you an estimate of the expected payoff of the policy.

This picture is just a fancy way of saying fly your helicopter in simulation and see how well it does, and measure the sum of rewards you get on average in the simulator. The picture I've drawn here assumes that you run your policy through the simulator just once. In general, you would run the policy through the simulator some number of times and then average to get an average over M simulations to get a better estimate of the policy's expected payoff. Now that I have a way – given any one fixed policy, this gives me a way to evaluate the expected payoff of that policy. So one reasonably obvious thing you might try to do is then just search for a policy, in other words search for parameters theta for your policy, that gives you high estimated payoff. Does that make sense? So my policy has some parameters theta, so my policy is my actions A are equal to theta transpose S say if there's a linear policy. For any fixed value of the parameters theta, I can evaluate – I can get an estimate for how good the policy is using the simulator. One thing I might try to do is search for parameters theta to try to maximize my estimated payoff. It turns out you can sort of do that, but that idea as I've just stated is hard to get to work. Here's the reason. The simulator allows us to evaluate policy, so let's search for policy of high value.

The difficulty is that the simulator is random, and so every time we evaluate a policy, we get back a very slightly different answer. So in the cartoon below, I want you to imagine that the horizontal axis is the space of policies. In other words, as I vary the parameters in my policy, I get different points on the horizontal axis here. As I vary the parameters theta, I get different policies, and so I'm moving along the X axis, and my total payoff I'm gonna plot on the vertical axis, and the red line in this cartoon is the expected payoff of the different policies. My goal is to find the policy with the highest expected payoff. You could search for a policy with high expected payoff, but every time you evaluate a policy – say I evaluate some policy, then I might get a point that just by chance looked a little bit better than it should have. If I evaluate a second policy and just by chance it looked a little bit worse. I evaluate a third policy, fourth, sometimes you look here – sometimes I might actually evaluate exactly the same policy twice and get back slightly different answers just because my simulator is random, so when I apply the same policy twice in simulation, I might get back slightly different answers.

So as I evaluate more and more policies, these are the pictures I get. My goal is to try to optimize the red line. I hope you appreciate this is a hard problem, especially when all [inaudible] optimization algorithm gets to see are these black dots, and they don't have direct access to the red line. So when my input space is some fairly high dimensional space, if I have ten parameters, the horizontal axis would actually be a 10-D space, all I get are these noisy estimates of what the red line is. This is a very hard stochastic optimization problem. So it turns out there's one way to make this optimization problem much easier. Here's the idea. And the method is called Pegasus, which is an acronym for something. I'll tell you later. So the simulator repeatedly makes calls to a random number generator to generate random numbers RT, which are used to simulate the stochastic dynamics. What I mean by that is that the simulator takes this input of state and action, and it outputs the mixed state randomly, and if you peer into the simulator, if you open the box of the simulator and ask how is my simulator generating these random mixed states ST plus one, pretty much the only way to do so – pretty much the only way to write a simulator with random outputs is we're gonna make calls to a random number generator, and get random numbers, these RTs, and then you have some function that takes this input S0, A0, and the results of your random number generator, and it computes some mixed state as a function of the inputs and of the random number it got from the random number generator.

This is pretty much the only way anyone implements any random code, any code that generates random outputs. You make a call to a random number generator, and you compute some function of the random number and of your other inputs. The reason that when you evaluate different policies you get different answers is because every time you rerun the simulator, you get a different sequence of random numbers from the random number generator, and so you get a different answer every time, even if you evaluate the same policy twice. So here's the idea. Let's say we fix in advance the sequence of random numbers, choose R1, R2, up to RT minus one. Fix the sequence of random numbers in advance, and we'll always use the same sequence of random numbers to evaluate different policies. Go into your code and fix R1, R2, through RT minus one. Choose them randomly once and then fix them forever.

If you always use the same sequence of random numbers, then the system is no longer random, and if you evaluate the same policy twice, you get back exactly the same answer. And so these sequences of random numbers, [inaudible] call them scenarios, and Pegasus is an acronym for policy evaluation of gradient and search using scenarios. So when you do that, this is the picture you get. As before, the red line is your expected payoff, and by fixing the random numbers, you've defined some estimate of the expected payoff. And as you evaluate different policies, they're still only approximations to their true expected payoff, but at least now you have a deterministic function to optimize, and you can now apply your favorite algorithms, be it gradient ascent or some sort of local [inaudible] search to try to optimize the black curve. This gives you a much easier optimization problem, and you can optimize this to find hopefully a good policy. So this is the Pegasus policy search method.

So when I started to talk about reinforcement learning, I showed that video of a helicopter flying upside down. That was actually done using exactly method, using exactly this policy search algorithm. This seems to scale well even to fairly large problems, even to fairly high dimensional state spaces. Typically Pegasus policy search algorithms have been using – the optimization problem is still – is much easier than the stochastic version, but sometimes it's not entirely trivial, and so you have to apply this sort of method with maybe on the order of ten parameters or tens of parameters, so 30, 40 parameters, but not thousands of parameters, at least in these sorts of things with them.

Student: So is that method different than just assuming that you know your simulator exactly, just throwing away all the random numbers entirely?

Instructor (Andrew Ng):So is this different from assuming that we have a deterministic simulator? The answer is no. In the way you do this, for the sake of simplicity I talked about one sequence of random numbers. What you do is – so imagine that the random numbers are simulating different wind gusts against your helicopter. So what you want to do isn't really evaluate just against one pattern of wind gusts. What you want to do is sample some set of different patterns of wind gusts, and evaluate against all of them in average. So what you do is you actually sample say 100 – some number I made up like 100 sequences of random numbers, and every time you want to evaluate a policy, you evaluate it against all 100 sequences of random numbers and then average. This is in exactly the same way that on this earlier picture you wouldn't necessarily evaluate the policy just once. You evaluate it maybe 100 times in simulation, and then average to get a better estimate of the expected reward. In the same way, you do that here but with 100 fixed sequences of random numbers. Does that make sense? Any other questions?

Student: If we use 100 scenarios and get an estimate for the policy, [inaudible] 100 times [inaudible] random numbers [inaudible] won't you get similar ideas [inaudible]?

Instructor (Andrew Ng):Yeah. I guess you're right. So the quality – for a fixed policy, the quality of the approximation is equally good for both cases. The advantage of fixing the random numbers is that you end up with an optimization problem that's much easier. I have some search problem, and on the horizontal axis there's a space of control policies, and my goal is to find a control policy that maximizes the payoff.

The problem with this earlier setting was that when I evaluate policies I get these noisy estimates, and then it's just very hard to optimize the red curve if I have these points that are all over the place. And if I evaluate the same policy twice, I don't even get back the same answer. By fixing the random numbers, the algorithm still doesn't get to see the red curve, but at least it's now optimizing a deterministic function. That makes the optimization problem much easier. Does that make sense?

Student: So every time you fix the random numbers, you get a nice curve to optimize. And then you change the random numbers to get a bunch of different curves that are easy to optimize. And then you smush them together?

Instructor (Andrew Ng):Let’s see. I have just one nice black curve that I’m trying to optimize.

Student: For each scenario.

Instructor (Andrew Ng):I see. So I’m gonna average over M scenarios, so I’m gonna average over 100 scenarios. So the black curve here is defined by averaging over a large set of scenarios. Does that make sense? So instead of only one – if the averaging thing doesn’t make sense, imagine that there’s just one sequence of random numbers. That might be easier to think about. Fix one sequence of random numbers, and every time I evaluate another policy, I evaluate against the same sequence of random numbers, and that gives me a nice deterministic function to optimize. Any other questions? The advantage is really that – one way to think about it is when I evaluate the same policy twice, at least I get back the same answer. This gives me a deterministic function mapping from parameters in my policy to my estimate of the expected payoff. That’s just a function that I can try to optimize using the search algorithm. So we use this algorithm for inverted hovering, and again policy search algorithms tend to work well when you can find a reasonably simple policy mapping from the states to the actions. This is sort of especially the low level control tasks, which I think of as sort of reflexes almost.

Completely, if you want to solve a problem like Tetris where you might plan ahead a few steps about what’s a nice configuration of the board, or something like a game of chess, or problems of long path plannings of go here, then go there, then go there, then sometimes you might apply a value function method instead. But for tasks like helicopter flight, for low level control tasks, for the reflexes of driving or controlling various robots, policy search algorithms were easier – we can sometimes more easily approximate the policy directly works very well. Some [inaudible] the state of RL today. RL algorithms are applied to a wide range of problems, and the key is really sequential decision making. The place where you think about applying reinforcement learning algorithm is when you need to make a decision, then another decision, then another decision, and some of your actions may have long-term consequences. I think that is the heart of RL’s sequential decision making, where you make multiple decisions, and some of your actions may have long-term consequences. I’ve shown you a bunch of robotics examples. RL is also applied to thinks like medical decision making, where you may have a patient and you want to choose a sequence of treatments, and you do this now for the patient, and the patient may be in some other state, and you choose to do that later, and so on.

It turns out there’s a large community of people applying these sorts of tools to queues. So imagine you have a bank where you have people lining up, and after they go to one cashier, some of them have to go to the manager to deal with something else. You have a system of multiple people standing in line in multiple queues, and so how do you route people optimally to minimize the waiting time. And not just people, but objects in an assembly line and so on. It turns out there’s a surprisingly large community working on optimizing queues. I mentioned game playing a little bit already. Things like financial decision making, if you have a large amount of stock, how do you sell off a large amount – how do you time the selling off of your stock so as to not affect market prices adversely too much? There are many operations research problems, things like factory automation. Can you design a factory to optimize throughput, or minimize cost, or whatever. These are all sorts of problems that people are applying reinforcement learning algorithms to.

Let me just close with a few robotics examples because they're always fun, and we just have these videos. This video was a work of Ziko Coulter and Peter Abiel, which is a PhD student here. They were working getting a robot dog to climb over difficult rugged terrain. Using a reinforcement learning algorithm, they applied an approach that's similar to a value function approximation approach, not quite but similar. They allowed the robot dog to sort of plan ahead multiple steps, and carefully choose his footsteps and traverse rugged terrain. This is really state of the art in terms of what can you get a robotic dog to do. Here's another fun one. It turns out that wheeled robots are very fuel-efficient. Cars and trucks are the most fuel-efficient robots in the world almost. Cars and trucks are very fuel-efficient, but the bigger robots have the ability to traverse more rugged terrain. So this is a robot – this is actually a small scale mockup of a larger vehicle built by Lockheed Martin, but can you come up with a vehicle that has wheels and has the fuel efficiency of wheeled robots, but also has legs so it can traverse obstacles. Again, using a reinforcement algorithm to design a controller for this robot to make it traverse obstacles, and somewhat complex gaits that would be very hard to design by hand, but by choosing a reward function, tell the robot this is a plus one reward that's top of the goal, and a few other things, it learns these sorts of policies automatically.

Last couple fun ones – I'll show you a couple last helicopter videos. This is the work of again PhD students here, Peter Abiel and Adam Coates who have been working on autonomous flight. I'll just let you watch. I'll just show you one more.

Student:[Inaudible] do this with a real helicopter [inaudible]?

Instructor (Andrew Ng):Not a full-size helicopter. Only small radio control helicopters.

Student:[Inaudible].

Instructor (Andrew Ng):Full-size helicopters are the wrong design for this. You shouldn't do this on a full-size helicopter. Many full-size helicopters would fall apart if you tried to do this. Let's see. There's one more.

Student: Are there any human [inaudible]?

Instructor (Andrew Ng):Yes, there are very good human pilots that can. This is just one more maneuver. That was kind of fun. So this is the work of Peter Abiel and Adam Coates. So that was it. That was all the technical material I wanted to present in this class. I guess you're all experts on machine learning now. Congratulations. And as I hope you've got the sense through this class that this is one of the technologies that's really having a huge impact on science in engineering and industry. As I said in the first lecture, I think many people use machine learning algorithms dozens of times a day without even knowing about it.

Based on the projects you've done, I hope that most of you will be able to imagine yourself going out after this class and applying these things to solve a variety of problems. Hopefully, some of you will also imagine yourselves writing research papers after this class, be it on a novel way to do machine learning, or on some way of applying machine learning to a problem that you care about. In fact, looking at project milestones, I'm actually sure that a large fraction of the projects in this class will be publishable, so I think that's great. I guess many of you will also go industry, make new products, and make lots of money using learning algorithms. Remember me if that happens. One of the things I'm excited about is through research or industry, I'm actually completely sure that the people in this class in the next few months will apply machine learning algorithms to lots of problems in industrial management, and computer science, things like optimizing computer architectures, network security, robotics, computer vision, to problems in computational biology, to problems in aerospace, or understanding natural language, and many more things like that.

So right now I have no idea what all of you are going to do with the learning algorithms you learned about, but every time as I wrap up this class, I always feel very excited, and optimistic, and hopeful about the sorts of amazing things you'll be able to do. One final thing, I'll just give out this handout. One final thing is that machine learning has grown out of a larger literature on the AI where this desire to build systems that exhibit intelligent behavior and machine learning is one of the tools of AI, maybe one that's had a disproportionately large impact, but there are many other ideas in AI that I hope you go and continue to learn about. Fortunately, Stanford has one of the best and broadest sets of AI classes, and I hope that you take advantage of some of these classes, and go and learn more about AI, and more about other fields which often apply learning algorithms to problems in vision, problems in natural language processing in robotics, and so on.

So the handout I just gave out has a list of AI related courses. Just running down very quickly, I guess, CS221 is an overview that I teach. There are a lot of robotics classes also: 223A, 225A, 225B – many robotics class. There are so many applications of learning algorithms to robotics today. 222 and 227 are knowledge representation and reasoning classes. 228 – of all the classes on this list, 228, which Daphne Koller teaches, is probably closest in spirit to 229. This is one of the classes I highly recommend to all of my PhD students as well.

Many other problems also touch on machine learning. On the next page, courses on computer vision, speech recognition, natural language processing, various tools that aren't just machine learning, but often involve machine learning in many ways. Other aspects of AI, multi-agent systems taught by [inaudible]. EE364A is convex optimization. It's a class taught by Steve Boyd, and convex optimization came up many times in this class. If you want to become really good at it, EE364 is a great class. If you're interested in project courses, I also teach a project class next quarter where we spend the whole quarter working on research projects.

So I hope you go and take some more of those classes. In closing, let me just say this class has been really fun to teach, and it's very satisfying to me personally when we set these insanely difficult hallmarks, and then we'd see a solution, and I'd be like, "Oh my god. They actually figured that one out." It's actually very satisfying when I see that. Or looking at the milestones, I often go, "Wow, that's really cool. I bet this would be publishable." So I hope you take what you've learned, go forth, and do amazing things with learning algorithms. I know this is a heavy workload class, so thank you all very much for the hard work you've put into this class, and the hard work you've put into learning this material, and thank you very much for having been students in this class.

[End of Audio]

Duration: 78 minutes