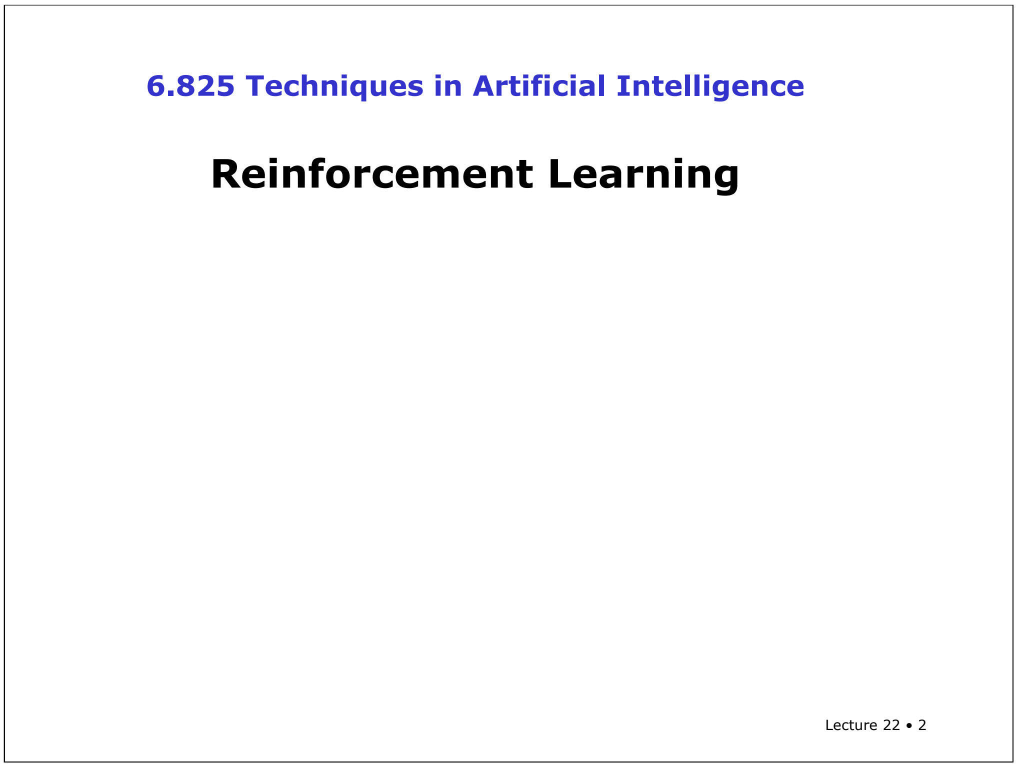
6.825 Techniques in Artificial Intelligence

Reinforcement Learning

When we talked about MDPs, we assumed that we knew the agent’s reward function, R, and a model of how the world works, expressed as the transition probability distribution. In reinforcement learning, we would like an agent to learn to behave well in an MDP world, but without knowing anything about R or P when it starts out.

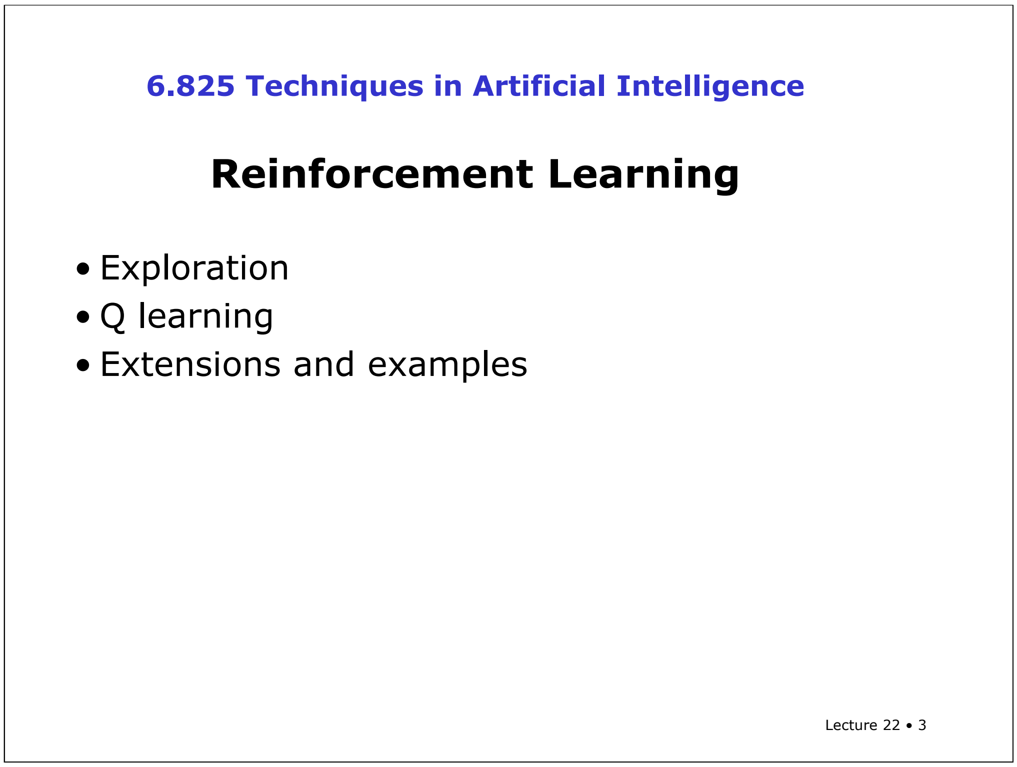


6.825 Techniques in Artificial Intelligence

Reinforcement Learning

Lecture 22 • 2

It’s called reinforcement learning because it’s related to early mathematical psychology models of conditioning, or behavior learning, in animals.

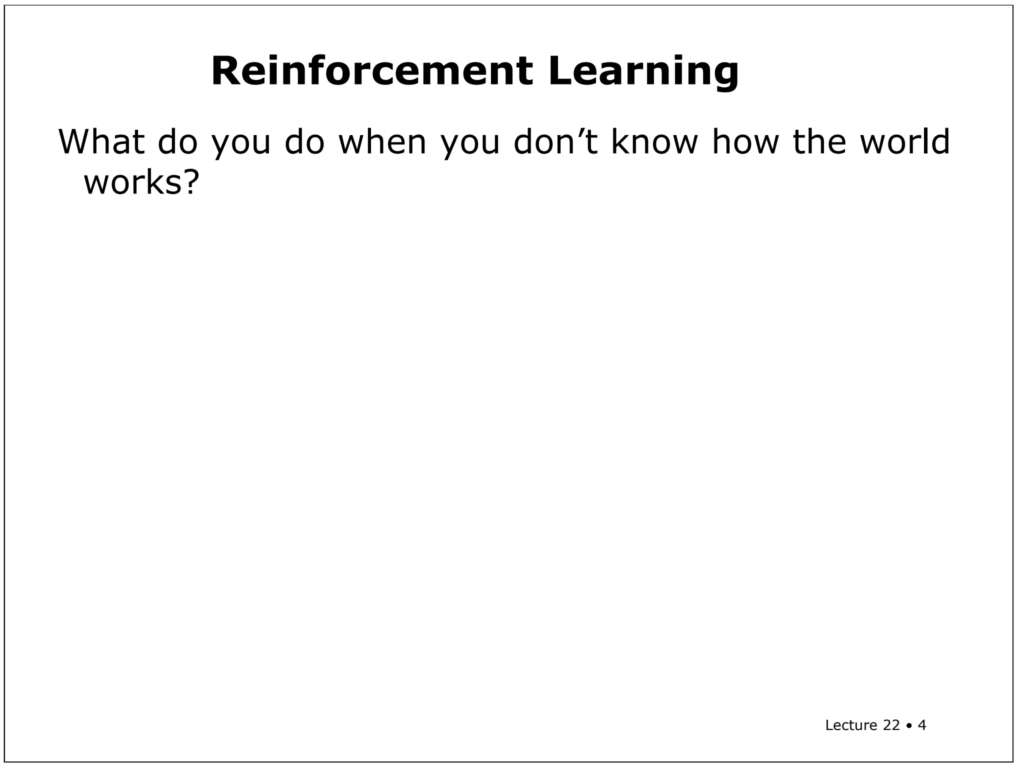


6.825 Techniques in Artificial Intelligence

Reinforcement Learning

- Exploration
- Q learning
- Extensions and examples

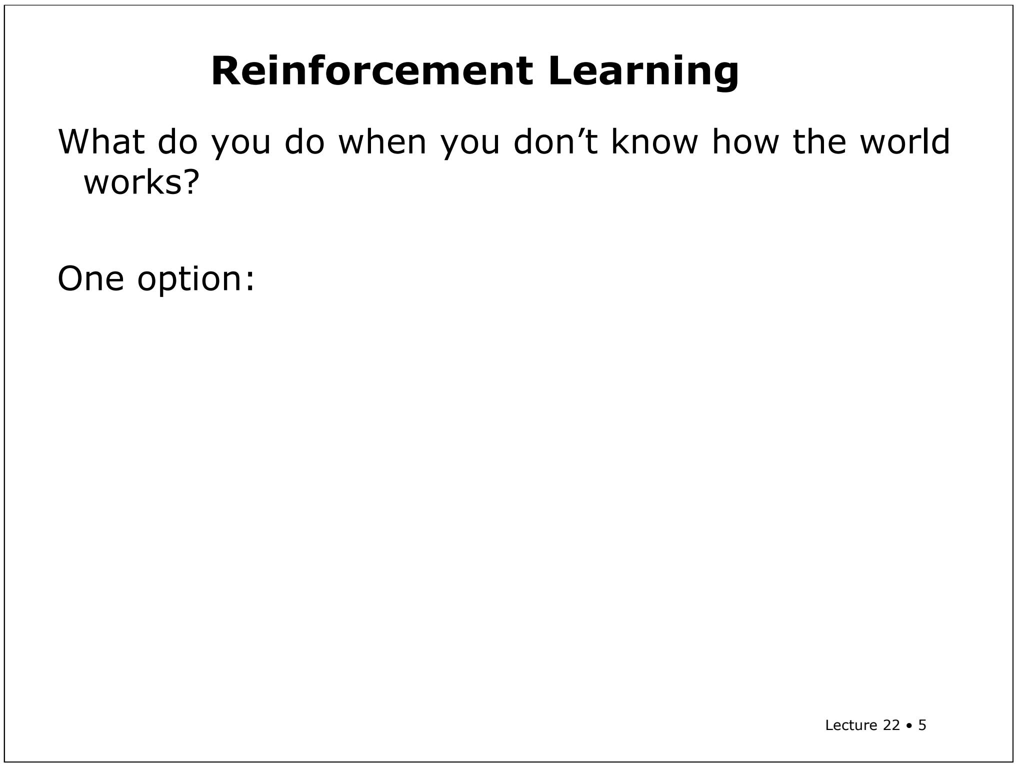
We'll look at the issue of exploration, talk about Q-learning, which is one of the most successful approaches to reinforcement learning, and then consider some extensions and examples.



Reinforcement Learning

What do you do when you don’t know how the world works?

So, how should you behave when you don’t even know how the world works?

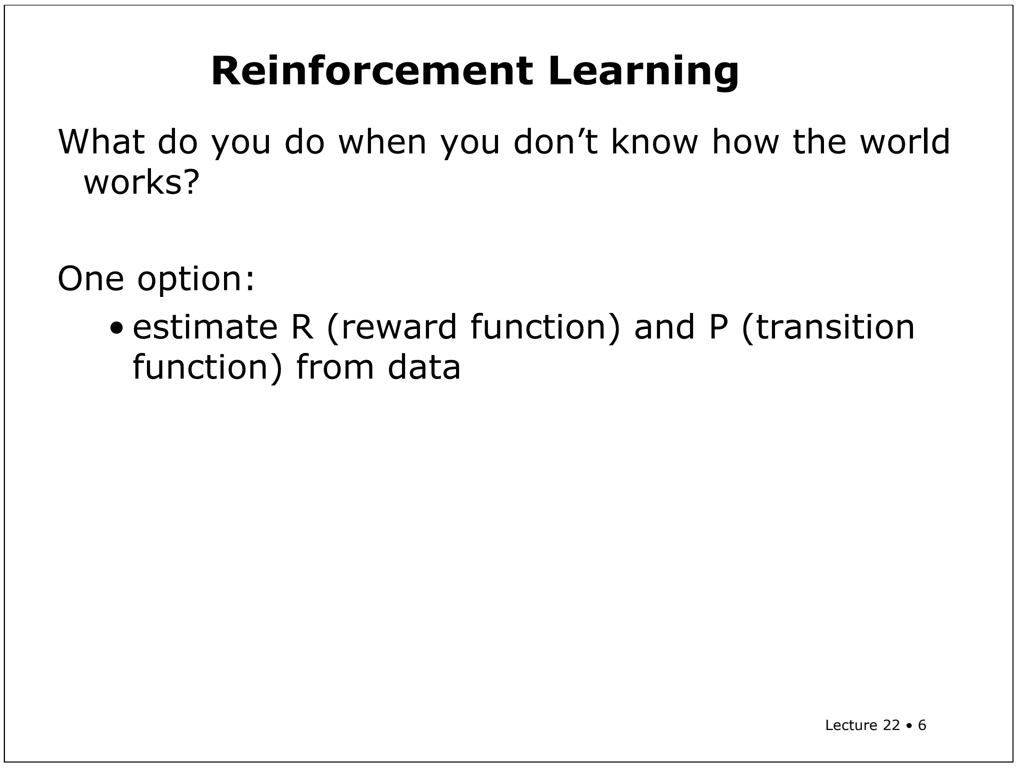


Reinforcement Learning

What do you do when you don't know how the world works?

One option:

One obvious strategy for dealing with an unknown world is to learn a model of the world, and then solve it using known techniques.



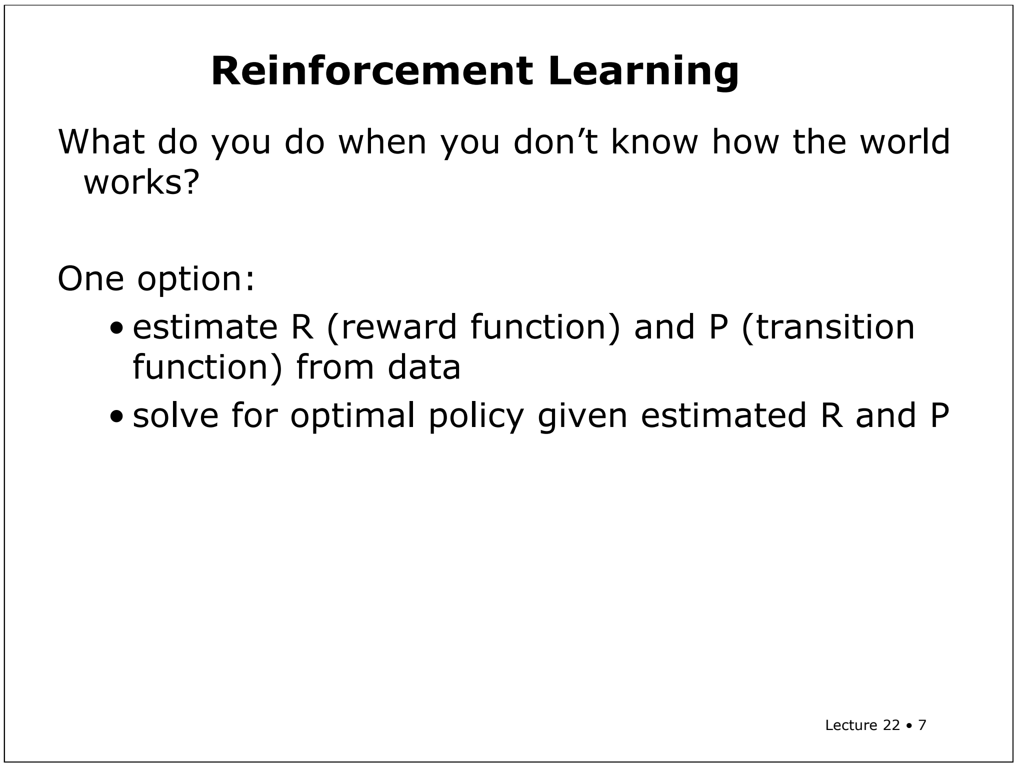
Reinforcement Learning

What do you do when you don’t know how the world works?

One option:

- estimate R (reward function) and P (transition function) from data

You all know how to do parameter estimation now, by counting the number of times various events occur and taking ratios. So, you can estimate the next-state distribution P(s’|s,a) by counting the number of times the agent has taken action a in state s and looking at the proportion of the time that s’ has been the next state. Similarly, you can estimate R(s) just by averaging all the rewards you’ve received when you were in state s.



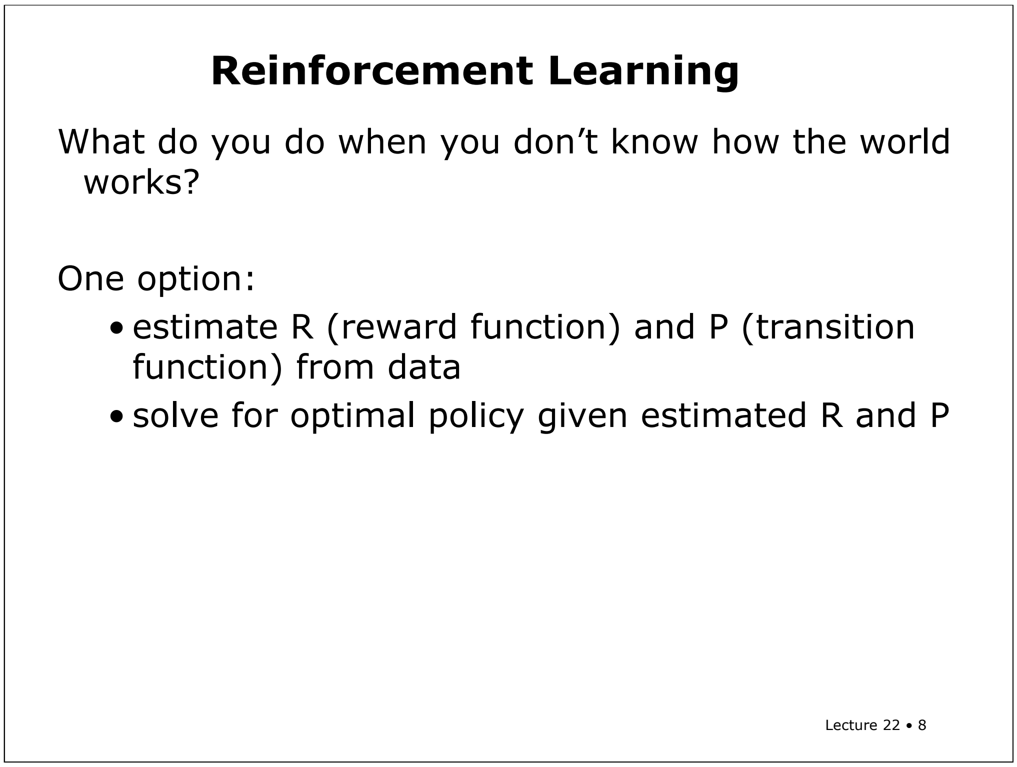
Reinforcement Learning

What do you do when you don't know how the world works?

One option:

- estimate R (reward function) and P (transition function) from data
- solve for optimal policy given estimated R and P

Once you have estimated values for R and P, then you can use value iteration to compute a policy that's optimal, if not for the real world, for the world described by the estimated model.



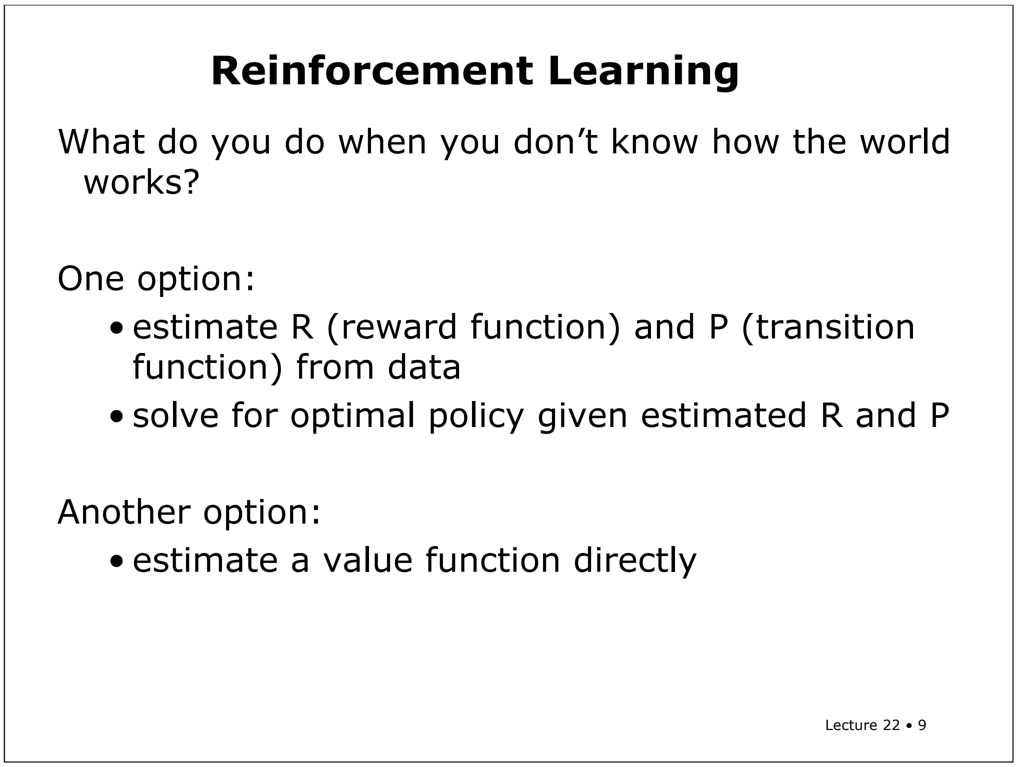
Reinforcement Learning

What do you do when you don’t know how the world works?

One option:

- estimate R (reward function) and P (transition function) from data
- solve for optimal policy given estimated R and P

Of course, there’s a question of how long you should gather data to estimate the model before it’s good enough to use to find a policy. One way to get around this problem is to re-estimate the model on every step. Each time you get a piece of experience, you update your model estimates. Then, you run value iteration on the updated model. This might be too computationally expensive; if so, you can run value iteration over the continually updated model as a kind of background job, at whatever rate you can afford computationally.



Reinforcement Learning

What do you do when you don't know how the world works?

One option:

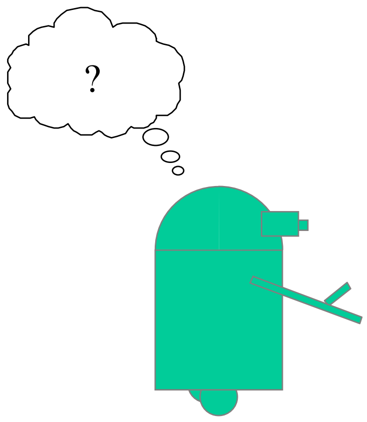
- estimate R (reward function) and P (transition function) from data
- solve for optimal policy given estimated R and P

Another option:

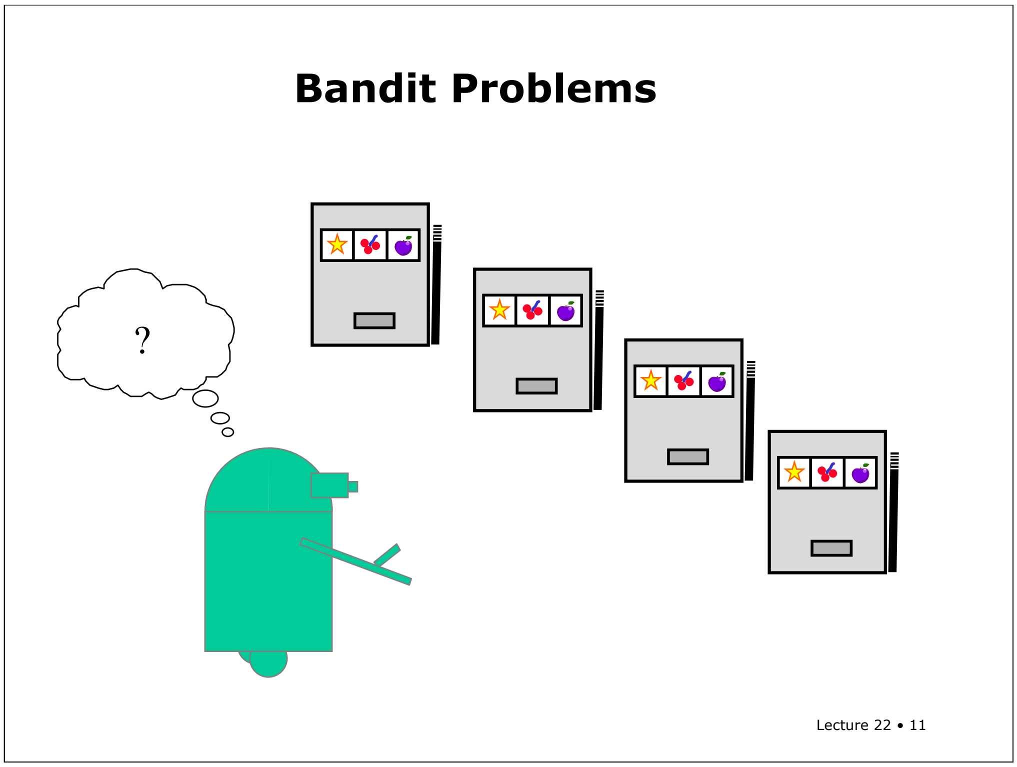
- estimate a value function directly

This approach is sound, and in a lot of cases, it's probably the right thing to do. But, interestingly enough, it's possible to find the optimal value function without ever estimating the state transition probabilities directly. We'll investigate an algorithm for doing that.

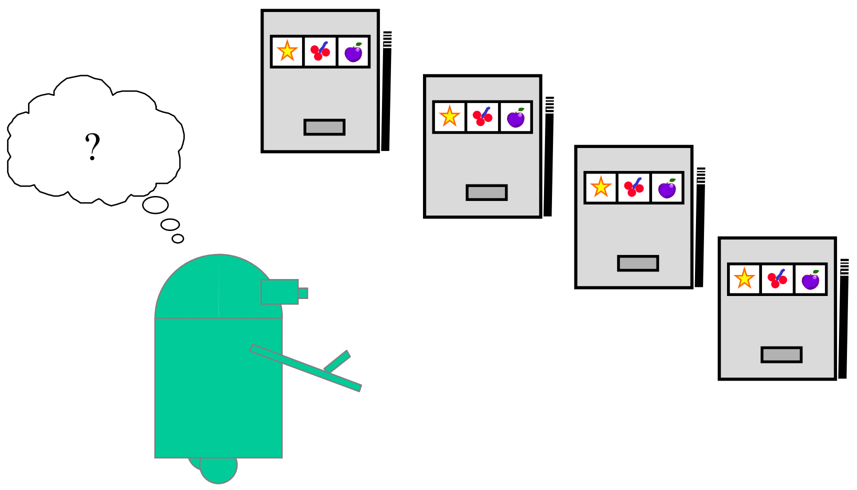
Bandit Problems



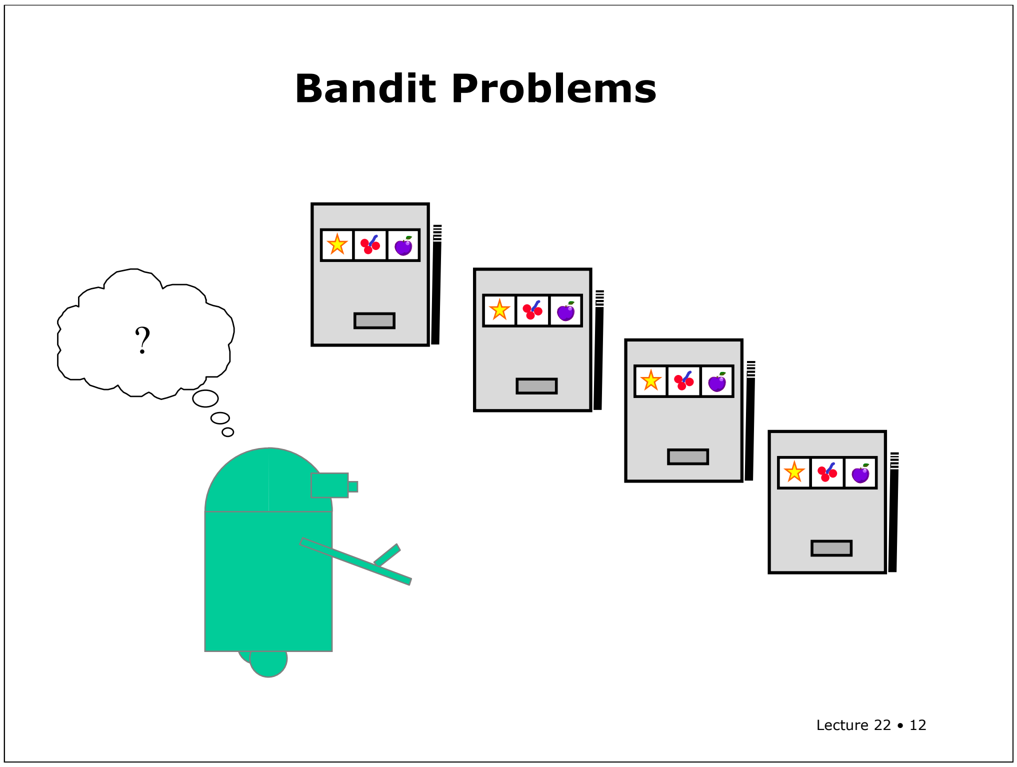
But first, let’s think about how the agent should choose its actions. In most supervised learning problems, the agent is simply given a big collection of data and asked to learn something from it. In reinforcement learning, there is an interesting added dimension: the agent gets to choose its own actions and, therefore, it has very direct influence on the data it will receive. Now there are two, possibly opposing reasons for the agent to choose an action: because it thinks the action will have a good result in the world, or because it thinks the action will give it more information about how the world works. Both of these things are valuable, and it’s interesting to think about how to trade them off.



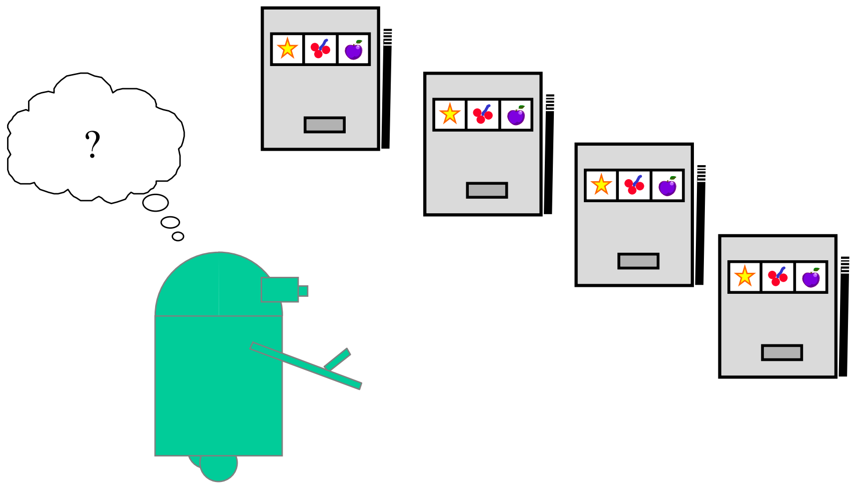
Bandit Problems



One view of the very simplest reinforcement learning problem, where there's a single state, is as a **bandit problem**. Imagine you enter a casino with k slot machines. Every time you pull an arm on a machine, it either pays off a dollar or nothing. Assume that each machine has a hidden probability of paying off, and that whenever you pull an arm, the outcome is independent of previous outcomes and is determined by the hidden payoff probability. This is called a **k-armed-bandit** problem, because, in English slang, slot machines are sometimes called one-armed bandits (presumably because they take all your money).



Bandit Problems

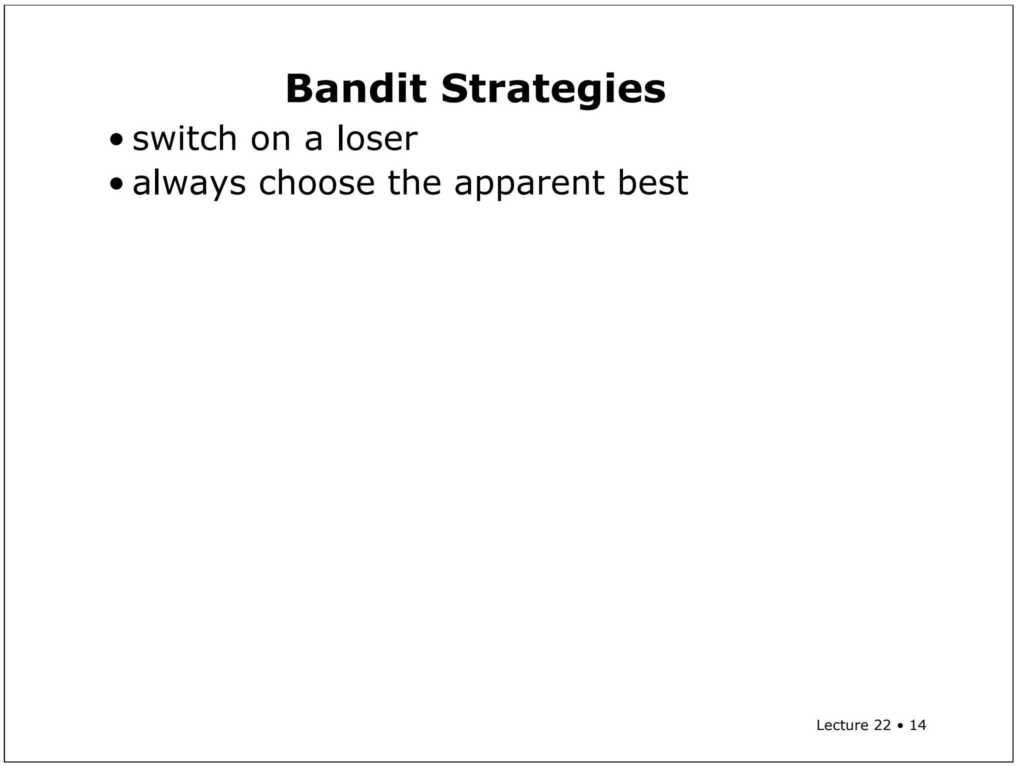


Now, assume that I let you into the casino and tell you that you have 1000 chances to pull the arm of a machine, and that you should try to make as much money as possible during that time. What should you do?

Bandit Strategies

- switch on a loser

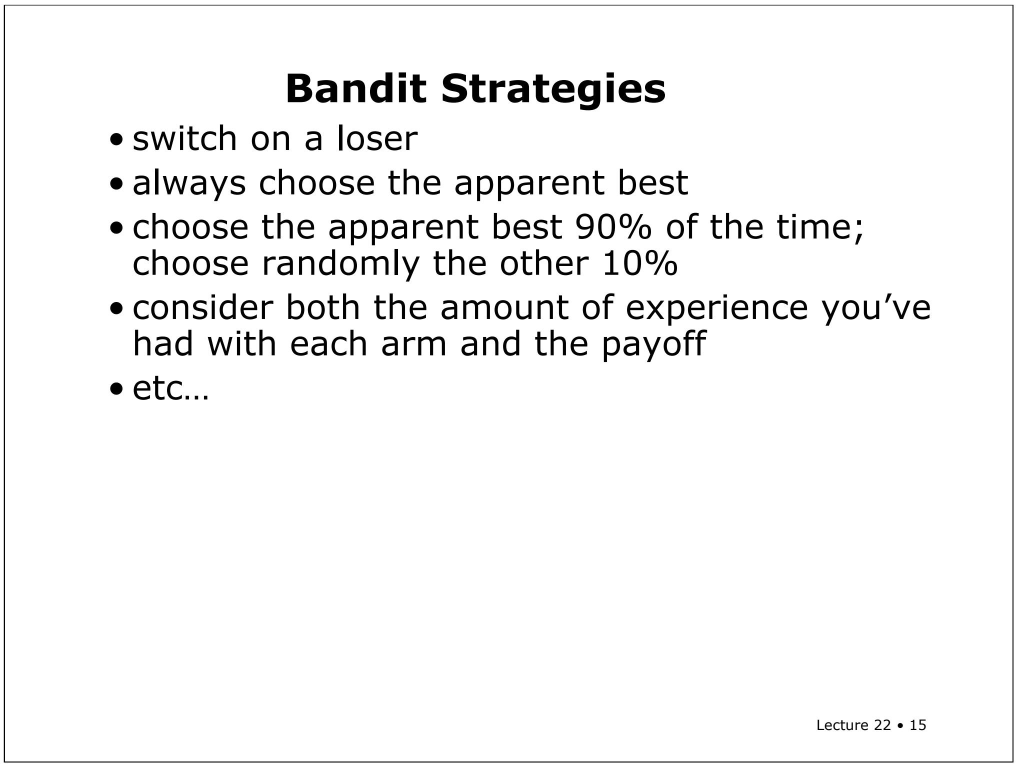
There are a lot of strategies, even for this simple problem. One of the first ones that people studied was “switch on a loser”. It says that you should pick one arm. As long as it keeps paying off, you should keep pulling it. As soon as it loses, go to the next arm, and so on. It can be shown that this strategy is better than behaving at random, but it’s not optimal!



Bandit Strategies

- switch on a loser
- always choose the apparent best

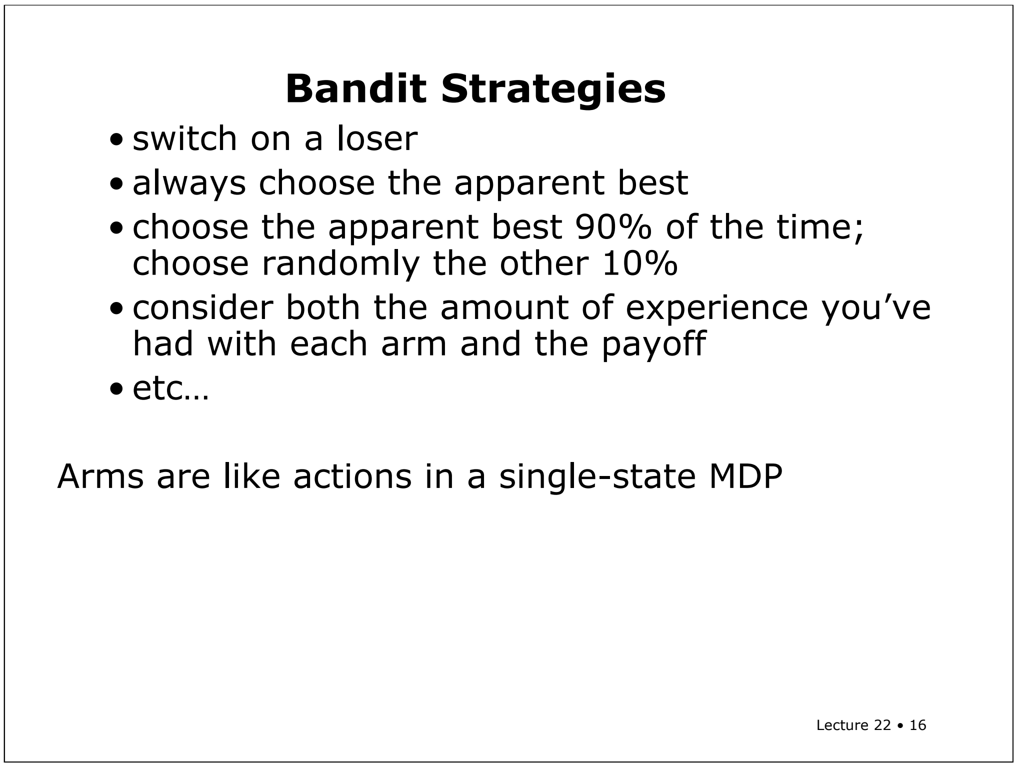
Another strategy would be to keep estimates of the payoff probabilities of each arm (by counting). Then, always choose the arm with the highest estimated probability of paying off. The problem with this strategy, which initially seems quite appealing, is that you might have initial bad luck with an arm that is actually pretty good, and arrive at a very low estimate of its probability of paying off. If this happens, you might never choose it again, thus preventing yourself from discovering that it's actually better than it seems to be now.



Bandit Strategies

- switch on a loser
- always choose the apparent best
- choose the apparent best 90% of the time; choose randomly the other 10%
- consider both the amount of experience you've had with each arm and the payoff
- etc…

Ultimately, the best strategies spend some time "exploring": trying all the arms to see what their probabilities are like, and some time exploiting: doing the apparently best action to try to get reward. In general, the longer you expect to live (the longer your horizon, in the finite horizon case, or the closer your gamma is to 1, in the discounted case), the more time you should devote to exploration, because the more you stand to lose by converging too early to the wrong action.



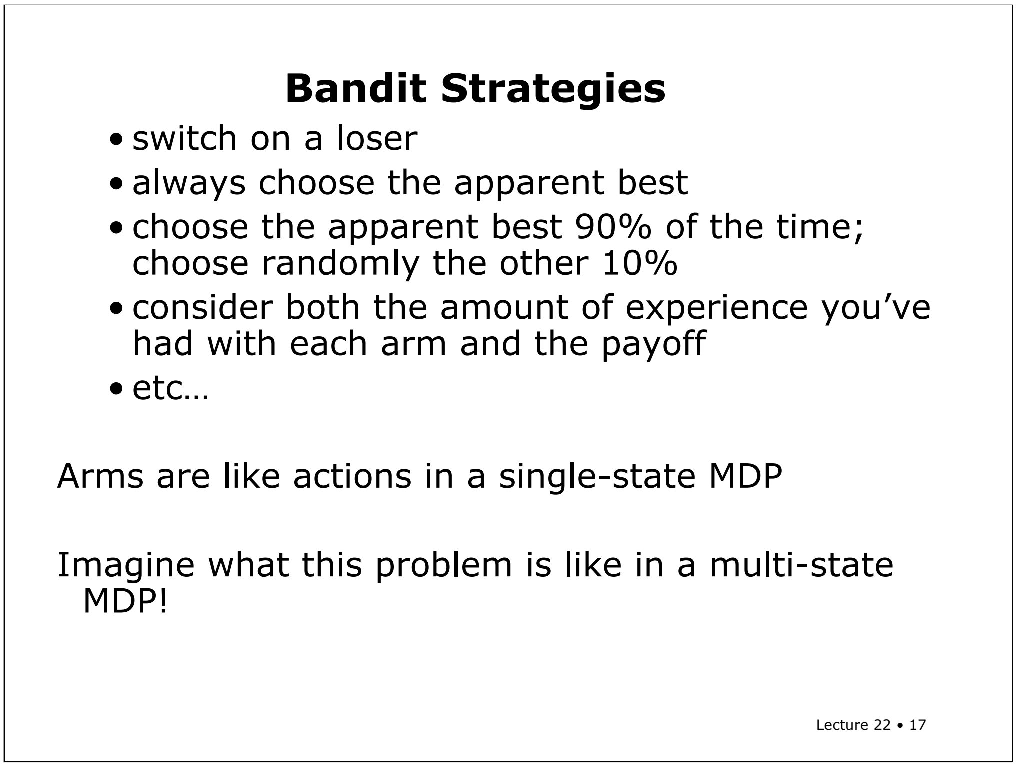
Bandit Strategies

- switch on a loser
- always choose the apparent best
- choose the apparent best 90% of the time; choose randomly the other 10%
- consider both the amount of experience you've had with each arm and the payoff
- etc…

Arms are like actions in a single-state MDP

Lecture 22 • 16

So, how does this relate to reinforcement learning? Well, you can think of the problem of choosing which action to take in a single-state MDP as being equivalent to the problem of choosing which arm to pull (though the rewards may have different distributions).



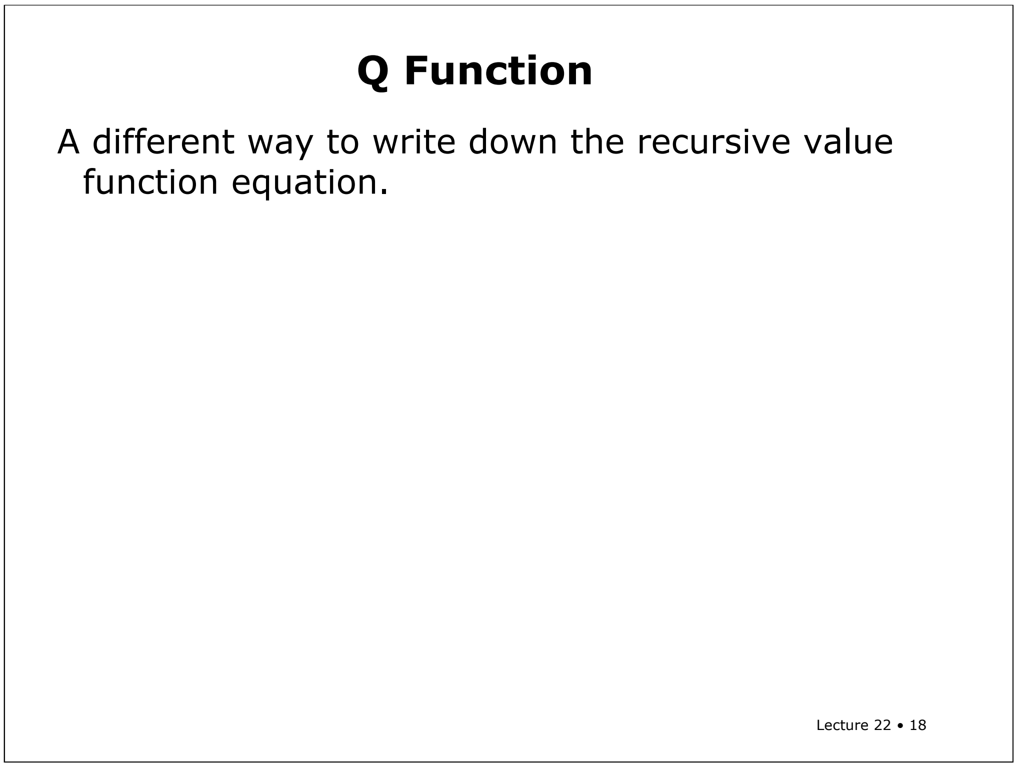
Bandit Strategies

- switch on a loser
- always choose the apparent best
- choose the apparent best 90% of the time; choose randomly the other 10%
- consider both the amount of experience you've had with each arm and the payoff
- etc…

Arms are like actions in a single-state MDP

Imagine what this problem is like in a multi-state MDP!

It gets even more complicated when you consider exploring a whole MDP. Now, it's as if you have a casino with many rooms, each room corresponding to a state in the MDP. And each room has many arms, corresponding to the actions available to the agent. Now, each time the agent pulls an arm, it gets some immediate payoff. And then it gets teleported to another room! according to a probability distribution that's a function of the room and the arm. It's really hard to think of the best exploration strategy for problems like this, and it's really still open from a theoretical perspective how to do it.



Q Function

A different way to write down the recursive value function equation.

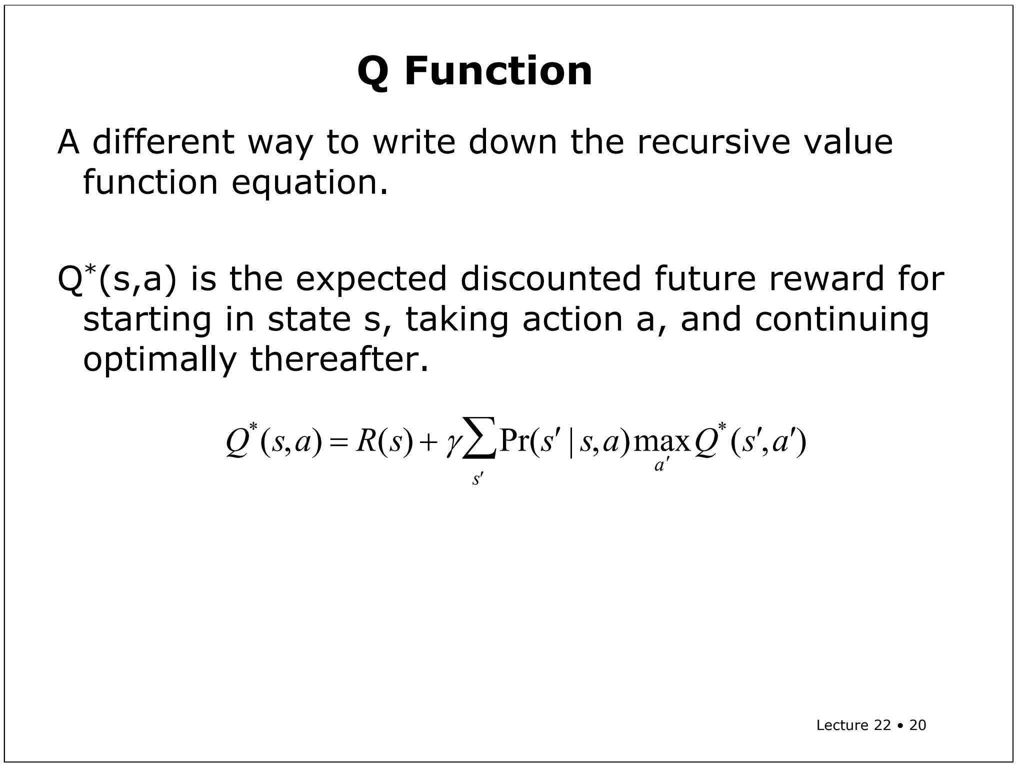
Assuming we have some way of choosing actions, now we're going to focus on finding a way to estimate the value function directly. For a long time, people worked on methods of learning V* directly, but they ran into a variety of problems. Then someone had the insight that if we just wrote the equations for the optimal value function down in a slightly different way, it would make learning much easier.

Q Function

A different way to write down the recursive value function equation.

$Q^*(s,a)$ is the expected discounted future reward for starting in state s, taking action a, and continuing optimally thereafter.

So, instead of trying to estimate V*, we'll focus on a slightly different function, called Q*. Q*(s,a) is the expected discounted future reward for starting in state s, taking a as our first action, and then continuing optimally. It's like V*, except that it specifies the first action, and that action could potentially be sub-optimal (we are going to compute Q* for each possible action in each state).



Q Function

A different way to write down the recursive value function equation.

$Q^*(s,a)$ is the expected discounted future reward for starting in state s, taking action a, and continuing optimally thereafter.

$$Q^*(s,a) = R(s) + \gamma \sum_{s'} \Pr(s' \mid s,a) \max_{a'} Q^*(s',a')$$

There's a nice set of recursive equations for the Q values, just as there was for V. The Q value of being in state s and taking action a is the immediate reward, R(s), plus the discounted expected value of the future. We get the expected value of the future by taking an expectation over all possible next states, s'. In each state s', we need to know the value of behaving optimally. We can get that by choosing, in each s', the action a' that maximizes Q*(s',a').

Q Function

A different way to write down the recursive value function equation.

Q*(s,a) is the expected discounted future reward for starting in state s, taking action a, and continuing optimally thereafter.

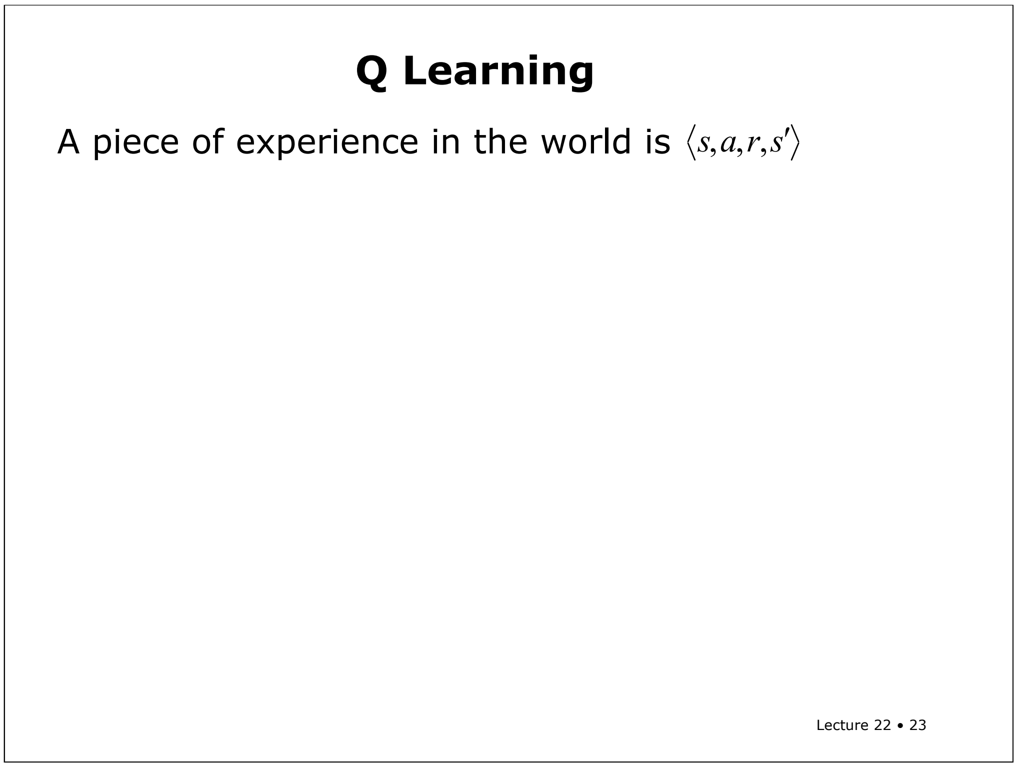
$$Q^*(s,a) = R(s) + \gamma \sum_{s'} \Pr(s' \mid s,a) \max_{a'} Q^*(s',a')$$

$$\pi^*(s) = \arg\max_{a} Q^*(s,a)$$

A convenient aspect of this approach is that if you know Q*, then it's really easy to compute the optimal action in a state. All you have to do is take the action that gives the largest Q value in that state. (Back when we were using V*, it required knowing the transition probabilities to compute the optimal action, so this is considerably simpler. And it will be effective when the model is not explicitly known).

Q Learning

Now we can look at an algorithm called Q learning, which estimates the Q* function directly, without estimating the transition probabilities. And, as we saw on the previous slide, if we know Q*, then finding the best way to behave is easy.



Q Learning

A piece of experience in the world is $\langle s, a, r, s' \rangle$

The learning algorithm deals with individual “pieces” of experience with the world. One piece of experience is a set of the current state s, the chosen action a, the reward r, and the next state s’. Each piece of experience will be folded into the Q values, and then thrown away, so there is no accumulation of old experience in memory.

Q Learning

A piece of experience in the world is $\langle s, a, r, s' \rangle$

- Initialize Q(s,a) arbitrarily

We start, as in value iteration, by initializing the Q function arbitrarily. Zero is usually a reasonable starting point.

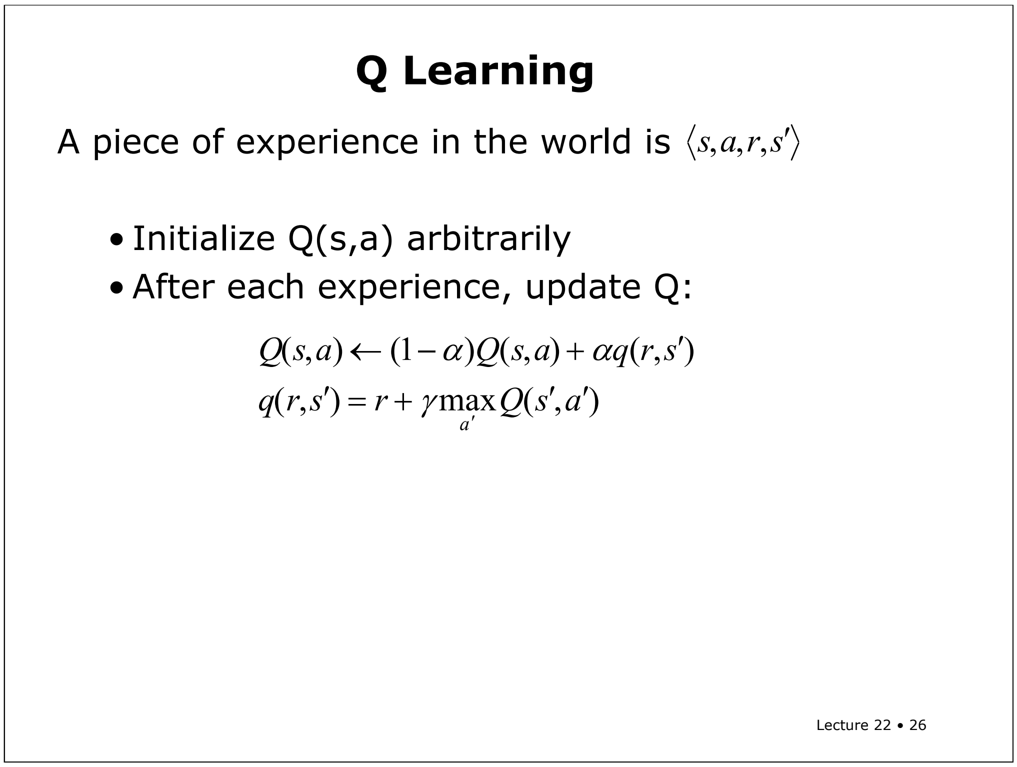
Q Learning

A piece of experience in the world is $\langle s, a, r, s' \rangle$

- Initialize Q(s,a) arbitrarily
- After each experience, update Q:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha q(r,s')$$

Now, after each experience, we update the Q function. The basic form of the update looks like this. The parameter alpha is a learning rate; usually it's something like 0.1. So, we're updating our estimate of Q(s,a) to be mostly like our old value of Q(s,a), but adding in a new term that depends on r and the new state. This kind of an update is essentially a running average of the new terms received on each step. The smaller alpha is, the longer-term the average is. With a small alpha, the system will be slow to converge, but the estimates will not fluctuate very much due to the randomness in the process. It is quite typical (and, in fact, required for convergence), to start with a large-ish alpha, and then decrease it over time.



Q Learning

A piece of experience in the world is $\langle s,a,r,s'\rangle$

- Initialize Q(s,a) arbitrarily
- After each experience, update Q:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha q(r,s')$$
$$q(r,s') = r + \gamma \max_{a'} Q(s',a')$$

Lecture 22 • 26

So, what is little q(r,s) going to be? You can think of it as an example of the value of taking action a in state s. The actual reward, r, is a sample of the expected reward R(s). And the actual next state, s', is a sample from the next state distribution. And the value of that state s' is the value of the best action we can take in it, which is the max over a' of Q(s',a').

Q Learning

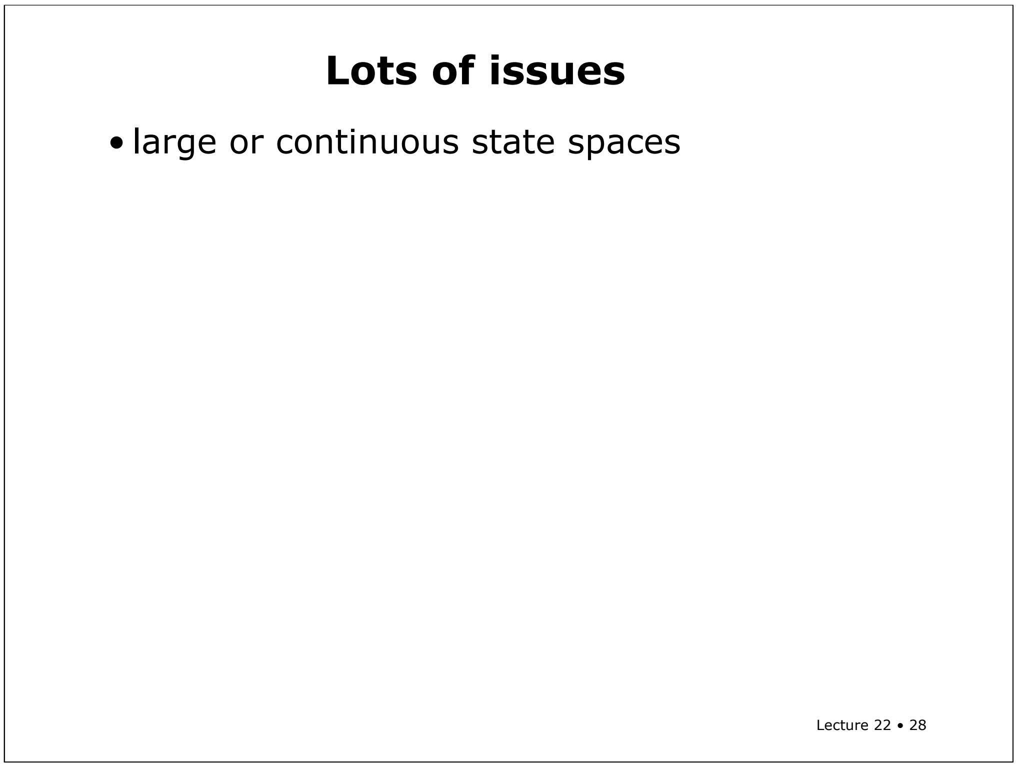
A piece of experience in the world is $\langle s, a, r, s' \rangle$

- Initialize Q(s,a) arbitrarily
- After each experience, update Q:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha q(r,s')$$
$$q(r,s') = r + \gamma \max_{a'} Q(s',a')$$

Guaranteed to converge to optimal Q if the world is really an MDP

This algorithm is guaranteed to converge to Q*, the optimal Q function, if the world is really an MDP, if we manage the learning rate correctly, and if we explore the world in such a way that we never completely ignore some actions. It's kind of tricky to show that it converges, because there are really two iterative processes going on here at once. One is the usual kind of averaging we do, when we collect a lot of samples and try to estimate their mean. The other is the dynamic programming iteration done by value iteration, updating the value of a state based on the estimated values of its successors. Both of those iterations are going on at once here, and the algorithm is still guaranteed to work. Amazing.



Lots of issues

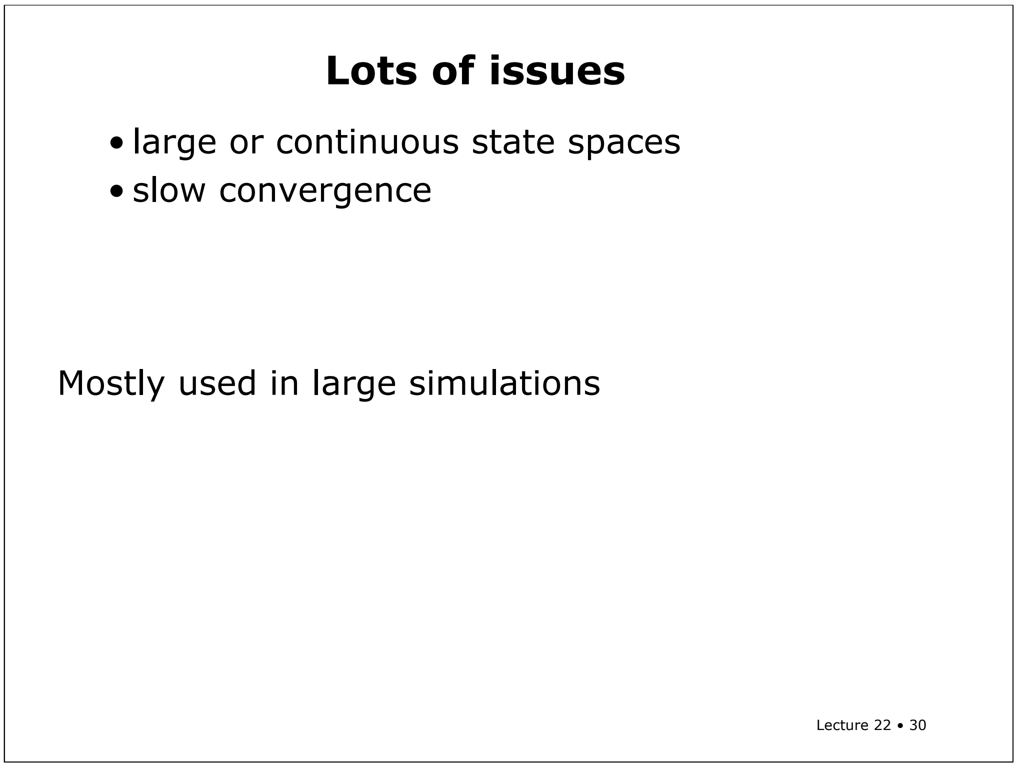
- large or continuous state spaces

Although Q learning comes with a theoretical guarantee, it doesn’t come close to solving all of our problems. Just like value iteration, it requires that the states and actions be drawn from a small enough set that we can store the Q function in a table. In many domains, we’ll have very large or even continuous state spaces, making the direct representation approach impossible. We can try to use a function approximator, such as a neural network, to store the Q function, rather than a table. Such approaches are no longer theoretically guaranteed to work, and they can be a bit tricky, but sometimes they work very well.

Lots of issues

- large or continuous state spaces
- slow convergence

Also, Q learning can sometimes be very slow to converge. There is a whole set of more advanced techniques in reinforcement learning that are aimed at addressing this problem.

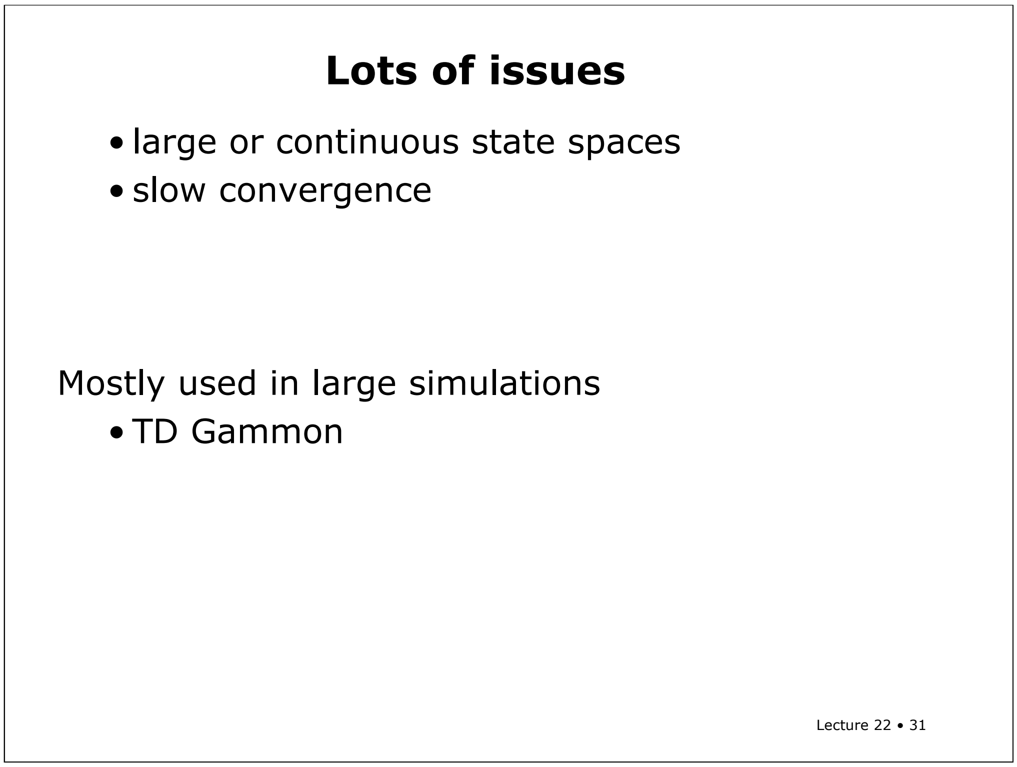


Lots of issues

- large or continuous state spaces
- slow convergence

Mostly used in large simulations

Because of the slow convergence, most of the applications of Q learning have been in very large domains for which we actually know a model. The domains are too large to solve directly using value iteration, though, so instead we use the known model to build a simulation. Then, using Q learning plus a function approximator, we learn to behave in the simulated environment, which yields a good control policy for the original problem.



Lots of issues

- large or continuous state spaces
- slow convergence

Mostly used in large simulations

- TD Gammon

The most striking success of reinforcement learning methods has been in the TD gammon system for learning to play backgammon. This system starts out not knowing anything about backgammon. It plays between 1 and 2 **million** games of backgammon against itself (that's a lot of backgammon!). It has learned to play backgammon so well that it can now draw the human world-champion backgammon player. It is recognized by humans as the expert in at least some aspects of the game, and the human experts have begun playing differently in some situations because of moves advocated by TD gammon.

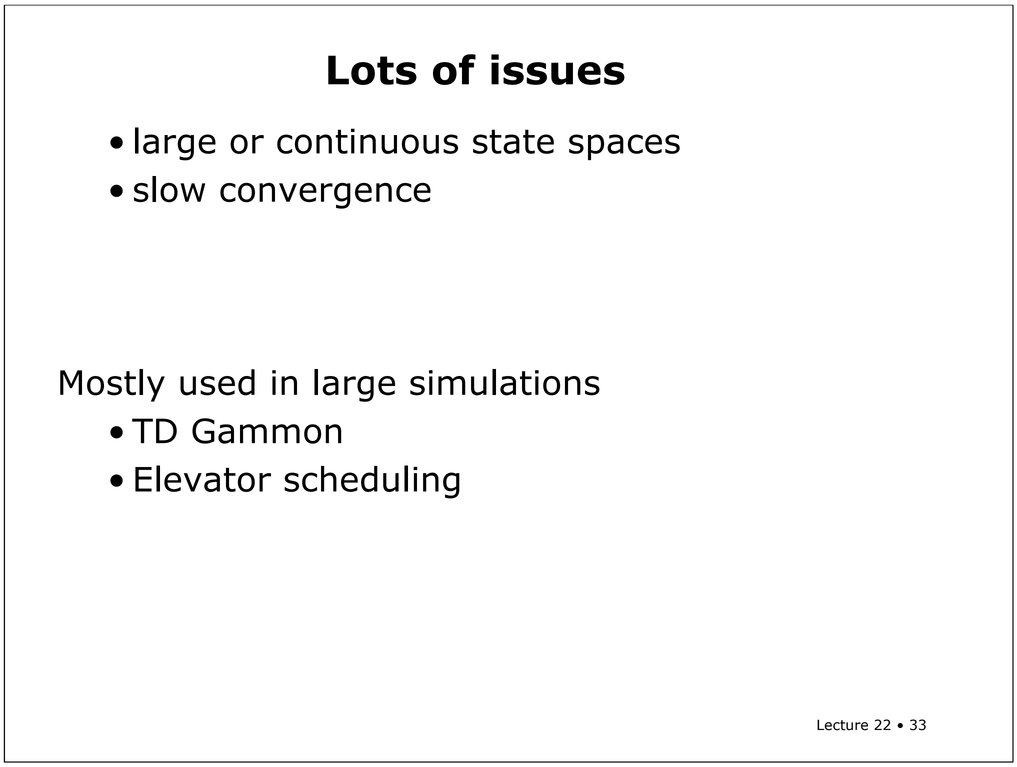
Lots of issues

- large or continuous state spaces
- slow convergence

Mostly used in large simulations

- TD Gammon
- Elevator scheduling

Another interesting example is elevator scheduling. In a skyscraper office building with many floors and many elevators, there is a serious control problem in deciding which elevators to send to which floors next. The input to the system is the locations of the elevators and the set of all buttons that have been pressed (both inside the elevators and outside, where people are waiting to be picked up). Again, through learning in a simulation of the building, the system learns to control the elevators, mapping the current inputs to a direction for each elevator, which the goal of maximizing the throughput of people through the system. The learned policies are considerably more effective that the ones that are standardly built in by the elevator companies.



Lots of issues

- large or continuous state spaces
- slow convergence

Mostly used in large simulations

- TD Gammon
- Elevator scheduling

So, reinforcement learning is a promising technology, but there are a lot of possible refinements that will have to be made before it has truly widespread application.