In the last video, we covered a high-level introduction to reinforcement learning and I ended by telling you we can't just jump into RL by unleashing a learning algorithm on a problem while being completely ignorant of what it means to set up the problem correctly. In order to get started, you should have an understanding of the RL workflow and how each part of the process contributes to solving the problem, and what some of the decisions are that you'll have to make along the way. So in this video, we're going to build on our basic understanding of reinforcement learning and explore what it means to set up the problem. I'm Brian, and welcome to a MATLAB Tech Talk.

Now I know that I said that understanding the system you're trying to control is the first step because you don't want to choose reinforcement learning if a traditional controls approach is better. However, I think it's easier to make that choice if you have a good understanding of the RL workflow. Therefore, for now, assume we've already decided we want to use reinforcement learning and we'll cover when not to use it in a future video once we've built up this good foundation.

Okay, so let's jump into the RL workflow. We need an environment where our agent can learn, and therefore, we need to choose what should exist within the environment and whether it's a simulation or a real physical setup. Then we need to think about what we ultimately want our agent to do and craft a reward function that will incentivize the agent to do just that. We need to choose a way to represent a policy—how we want to structure the parameters and logic that make up the decision-making part of the agent. Once we have this set up, we choose a training algorithm and get to work finding the optimal policy. Finally, we need to exploit the policy by deploying it onto an agent in the field and verifying the results. To put this workflow into perspective, let's think about each of these steps in the context of two examples: balancing an inverted pendulum and getting a robot to walk. So let's get to it.

The environment is everything that exists outside the agent. Practically speaking, it's where the agent sends actions and it's what generates rewards and observations.

I think this concept is a bit confusing at first, especially coming from a controls background, because we tend to think of the environment as everything outside of the controller and the plant; things like road imperfections, wind gusts, and other disturbances that impact the system you're trying to control. But in reinforcement learning, the environment is everything outside the controller. So this would include the plant dynamics as well. For the walking robot example, most of the

robot is part of the environment. The agent is just the bit of software that is generating the actions and updating the policy through learning. It's the brain of the robot, so to speak.

The reason this distinction is important is because with reinforcement learning, the agent doesn't need to know anything about the environment at all. This is called model-free RL, and it's powerful because you can basically just plop an RL-equipped agent into any system and, assuming you've given the policy access to the observations, actions, and enough internal states, the agent will learn how to collect the most reward on its own. This means the agent doesn't need to initially know anything about our walking robot. It'll still figure out how to collect rewards without knowing for example how the joints move or how strong the actuators are or the lengths of the appendages.

But as engineers, we do typically know some things about the environment, so why are we throwing out all our knowledge of physics and not helping the agent at all? This seems crazy! Well, this is where model-based RL can help.

Without any understanding of the environment, an agent needs to explore all areas of the state space to fill out its value function, which means that it'll spend some time exploring low-reward areas during the learning process. However, as designers, we often know some parts of the state space that are not worth exploring and so by providing a model of the environment or part of the environment, we provide the agent with this knowledge. For example, take agent that is trying to determine the fastest route to a destination. Should it go right or left at this point? Without a model, the agent would have to explore the entire map to know what the best action is. With a model, the agent could explore going right, sort of mentally without having to take that action physically. It could then figure out that going right results in a dead end and our agent would then go left. In this way, a model can complement the learning process by avoiding areas that are known to be bad, and exploring and learning the rest.

Model-based RL is pretty powerful, but the reason model-free RL is so popular right now is because people hope to use it to solve problems where developing a model is difficult, such as controlling a car or a robot from pixel observations. And also, since model-free RL is the more general case, we're going to focus on it for the rest of this series.

Okay, we know the agent learns by interacting with the environment, so it makes sense that we have a way for the agent to actually interact with it. This might be a physical environment or a simulation. For example, for the inverted pendulum,

we may let the agent learn how to balance by running it with a physical pendulum setup. This might be a good solution since it's probably hard for the hardware to damage itself or others. With the walking robot, however, this might not be such a good idea. As you could imagine, if the agent treats the robot and world like a black box it knows nothing about, then it's going to do a lot of falling and flailing before it even learns how to move its legs, let alone how to walk. Not only could this damage the hardware, but it would be extremely time consuming to have to pick the robot up each time. Not ideal.

So, an attractive alternative is to train your agent within a high-fidelity model of the environment and simulate the learning experience. And there are a lot of benefits to doing this.

The first comes from the idea of sample inefficiency. Learning is a process and that requires lots of samples: lots of trials, errors, and corrections, often in the millions or tens of millions. And so with a simulation, you have the ability to run the learning process at faster than real time and you can also spin up lots of simulations and run them all in parallel.

The other beneficial thing you can do with a model of the environment is simulate conditions that are hard to test for in the real world. For example, with the walking robot, you could simulate walking on a low-friction surface like ice, which would help the robot stay upright on all surfaces.

The nice part about needing a simulation is that for control problems we usually already have a good model of the system and environment since we typically need it from traditional control design. This is where if you already have a model built in MATLAB or Simulink, you can replace your existing controller with an RL agent, add a reward function to the environment, and start the learning process.

One of the difficulties here is figuring out how much of the environment to model —what to include and what to leave out. However, this is the same question you have when modeling a plant for controller design, and so you can use the same intuition about your system to build an RL environment model.

One approach is to start training on a simple model, find the right combination of hyper parameters that will let training succeed, and then add more complexity to the model later on. Hyper-parameters are the knobs we can turn on the training algorithms that set things like the learning rate and the sample times, and I'll cover this in more detail in a future video.

Okay, with the environment set, the next step is to think about what you want your agent to do and how you'll reward it for doing what you want. This is similar to the cost function in LQR, in which we think about performance versus effort.

However, unlike LQR where the cost function is quadratic, in RL there's really no restriction on creating a reward function. We can have sparse rewards, or rewards every time step, or rewards that only come at the very end of an episode after long periods of time. They can be calculated from a nonlinear function or calculated using thousands of parameters. Really, it completely depends on what it takes to effectively train your agent.

Want to get an inverted pendulum to stand upright? Then maybe give more rewards to your agent as the angle from vertical gets smaller. Want to take controller effort into account? Then subtract rewards as actuator use increases. Want to encourage a robot to walk across the floor? Then give the agent a reward when it reaches some state that's far away.

With that being said, making a reward function is easy. It can be pretty much be any function you can think of. Making a good reward function, on the other hand, is really, really hard. And unfortunately, there isn't a straightforward way to craft your reward to guarantee your agent will converge on the solution you actually want. And I think it boils down to two main reasons.

One, often the goal you want to incentivize comes after a long sequence of actions; this is the sparse reward system. Therefore, your agent will stumble around for long periods of time, not receiving any rewards in the process. This would be the case for the walking robot by only giving a reward after the robot successfully walked 10 meters. The chance your agent will randomly stumble on the action sequence that produces the sparse reward is very unlikely. Imagine the luck needed to generate all the correct motor commands to keep a robot upright and walking, rather than just flopping around on the ground!

It's possible, but relying on that random exploration is extremely slow to where it's impractical.

This sparse reward problem can be improved by shaping the reward—providing smaller intermediate rewards that coax the agent along the right path. But reward shaping comes with its own set of problems, and this is the second reason crafting a reward function is difficult. If you give an optimization algorithm a shortcut, it'll take it! And shortcuts are hidden within reward functions, and more so when you start shaping them. This causes your agent to converge on a solution that is

optimal given the reward function, but not ideal.

An easy example to think about is give an intermediate reward if the body of the robot traveled 1 meter from its current spot. The optimal solution might not be to walk that 1 meter, but rather fall ungracefully toward the reward. To the learning algorithm, walking and falling both provide the same reward, but obviously, to the designer, one result is preferred over the other.

I don't want to make crafting a reward function sound easy, because getting it right is possibly one of the more difficult tasks in reinforcement learning. However, hopefully with this general overview you'll be in a better position to at least understand some of the things you need to watch out for and that might make crafting the reward function a little less painful.

Okay, so now that we have the environment which provides the rewards, we're ready to start work on the agent itself. The agent is comprised of the policy and the learning algorithm and these two things are intimately intertwined. Many learning algorithms require a specific policy structure and choosing an algorithm depends on the nature of the environment. And we'll cover this in the next video, but before I end this video, I want to introduce the topic and get us thinking about how the parameters and the logic within the policy can be represented.

Remember, the policy is a function that takes in state observations and outputs actions, so really, any function with that input and output relationship can work. In that way of thinking, we could use a simple table to represent policies.

Tables are exactly what you'd expect. They're an array of numbers where you use an input as a lookup address and output the corresponding value. For example, a Q-function is a table that maps states and actions to value. So given a state, S, the policy would be to look up the value of every possible action from that state and choose the action with the highest value. And training an agent with a Q-function would consist of developing over time all the actions and their values for each state.

This type of representation falls apart when the number of action value pairs gets really large or becomes infinite. This is the so-called curse of dimensionality. Imagine our inverted pendulum. The state of the pendulum can be any angle from -pi to pi, and the action you can take can be any motor torque from the negative limit to the positive limit. Trying to capture all of that in a table is not feasible. Now, we could represent the continuous nature of the state-action space with a continuous function. But setting up this function where we could learn the right

parameters would require us to know the structure of the function ahead of time, which might be difficult for high degree of freedom systems or nonlinear systems.

So instead, it makes sense to try to represent the policy with a general-purpose function approximator. Something that can handle continuous state action spaces without us having to set the structure ahead of time. And we get that with deep neural networks.