In the last video, we focused mainly on setting up the environment: what the environment includes and how rewards from the environment are used to guide the agent's behavior. In this video, we're going to focus on the setting up the agent—specifically, the last three steps in the RL workflow.

Now, to fully cover these steps is easily several semesters' worth of material. Luckily, that's not the goal of this 17-minute video. Instead, I want to introduce a few topics at a high level that will give you a general understanding of what's going on so that it makes more sense when you dive in with other, more complete sources. So with that being said, in this video, I will address these two main questions:

One, why use neural networks to represent functions as opposed to something else, like tables or transfer functions? And two, why you may have to set up two neural networks and how they complement each other in a powerful family of methods called actor-critic. So let's get to it. I'm Brian, and welcome to a MATLAB Tech Talk.

Last time, we ended by explaining how a policy is a function that takes in state observations and outputs actions. And then I briefly introduced the idea of why tables and specifically defined functions aren't a great solution in many cases. This is because tables are impractical when the state and action space get really large and also because it's difficult to craft the right function structure for complex environments. Using neural networks, however, can address both of these problems.

A neural network is a group of nodes, or artificial neurons, that are connected in a way that allows them to be a universal function approximator. This means that given the right combination of nodes and connections, we can set up the network to mimic any input and output relationship. This is good for us because we can use, say, the hundreds of pixel values in a robotic vision system as the input into this function, and the output could be the actuator commands that drive the robot's arms and legs. Even though the function might be extremely complex, we know that there is a neural network of some kind that can achieve it.

If you're not familiar with the mathematics of a neural network, I highly recommend the four-part series by 3 Blue 1 Brown on the topic. He provides a fantastic visualization of what's going on within the network, so I'll skip most of that discussion here. I do, however, want to highlight just a few things.

On the left are the input nodes, one for each input to the function, and on the

right are the output nodes. In between are columns of nodes called hidden layers. This network has 2 inputs, 2 outputs, and 2 hidden layers of 3 nodes each. With a fully connected network, there is an arrow, or a weighted connection from each input node to each node in the next layer, and then from those nodes to the layer after that and again until the output nodes. The value of a node is equal to the sum of every input node times its respective weighting factor plus a bias. We can perform this calculation for every node in a layer and write it out in a compact matrix form as a system of linear equations.

Now if we calculated the values of the nodes simply like this and then fed them as inputs into the next layer to perform the same type of linear operations and again to the output layer, you'll probably have a concern. How in the world could a bunch of linear equations act as a universal function approximator? Specifically, how could it represent a nonlinear function? Well, that's because I left out a step, which is possibly one of the most important aspects of an artificial neural network. After the value of a node has been calculated, an activation function is applied that changes the value of the node in some way. Two popular activation functions are the sigmoid, which squishes the node value down to between 0 and 1, and the ReLU function, which basically zeroes out any negative node values. There are a number of different activation functions but what they all have in common is that they are nonlinear, which is critical to making a network that can approximate any function.

Now, as to why this is the case. I really like the explanations from Brendon Fortuner and Michael Neilson, who show how it can be demonstrated with ReLU and sigmoid activations, respectively. I've linked to their blogs in the description.

Okay, so now let's remind us where we are. We want to find a function that can take in a large number of observations and transform them into a set of actions that will control some nonlinear environment. And since the structure of this function is often too complex for us to solve for directly, we want to approximate it with a neural network that learns the function over time. And it's tempting to think that we can just plunk any network in and let loose a reinforcement learning algorithm to find the right combination of weights and biases and we're done. Unfortunately, as usual, that's not the case.

We have to make a few choices about our neural network ahead of time in order to make sure it's complex enough to approximate the function we're looking for, but not too complex as to make training impossible or impossibly slow. For example, as we've already seen, we need to choose an activation function, and the number of hidden layers, and the number of neurons in each layer. But beyond

that, we also have control over the internal structure of the network. Should it be fully connected like the network I've drawn, or should the connections skip layers like in a residual neural network? Should they loop back on themselves to create internal memory with recurrent neural networks? Should groups of neurons work together like with a convolutional neural network? And so on.

We have a lot of choices, but as with other control techniques there isn't one right approach. A lot of times, it comes down to starting with a network structure that has already worked for the type of problem you're trying to solve and tweak it from there.

Now I keep saying that we use these neural nets to represent a policy in the agent, but as for exactly what that means, we need to look at a high-level description of a few different classes of reinforcement learning algorithms: policy function based, value function based, and actor-critic. The following will definitely be an oversimplification, but if you're just trying to get a basic understanding of the ways RL can be approached, I think this will help get you started.

At a high level, I think policy function-based leaning algorithms make a lot of sense because we're trying to train a neural network that takes in the state observations and outputs an action. This is closely related to what a controller is doing in a control system. We'll call this neural network the actor because it is directly telling the agent how to act.

The structure looks pretty straightforward, so now the question is, how do we approach training this type of network? To get a general feel for how this is done, let's look at the Atari game Breakout.

If you're not familiar, Breakout is this game where you are trying to eliminate bricks using a paddle to direct a bouncing ball. This game only has three actions: move the paddle left, right, or not at all, and a near-continuous state space that includes the position of the paddle, the position and velocity of the ball, and the location of the remaining bricks.

For this example, we'll say the observation is a screen capture of the game, feeding in one frame at a time, and therefore there is one input into our neural net for every pixel. Now, with the network set there are many approaches to training it, but I'm going to highlight one broad approach that a lot of variations, and those are policy gradient methods. Policy gradient methods can work with a stochastic policy, which means that rather than a deterministic—take a left or right —the policy would output the probability of taking a left or the probability of

taking a right. A stochastic policy takes care of the exploration/exploitation problem because exploring is built into the probabilities. Now, when we learn, the agent just needs to update the probabilities. Is taking a left a better option than right? Push the probability that you take a left in this state a little bit higher. Then over time, the agent will nudge these probabilities around in the direction that produces the most reward.

So how does it know whether the actions were good or not? The idea is this: Execute the current policy, collect reward along the way, and update the network to increase the probabilities of actions that increased reward. If the paddle went left, missing the ball, causing a negative reward? Then change the neural network to increase the probability of moving the paddle right next time the agent is in that state. Essentially, it's taking the derivative of each weight and bias with respect to reward and adjusting them in the direction of positive reward increase. In this way, the learning algorithm is moving the weights and biases of the network to ascend up the reward slope. This is why the term gradient is used in the name.

The math behind this is more than I want to go into in this video, but I'd encourage you to read up on the policy gradient theorem to see how it's possible to find the gradient without actually taking a derivative. I've left a link in the description.

One of the downsides of policy gradient methods is that the naive approach of just following the direction of steepest ascent can converge on a local maxima rather than global. They also can converge slowly due to their sensitivity to noisy measurements, which happens, for example, when it takes a lot of sequential actions to receive a reward and the resulting cumulative reward has high variance between episodes. Imagine having to string together hundreds or thousands of sequential actions before the agent ever receives a single reward. You could see how time consuming it could be to train an agent using that extremely sparse reward system. Keep that in mind, and we'll come back to that problem by the end of this video.

Let's move on to value function-based learning algorithms. For these, let's start with an example using the popular grid world as our environment. In this environment there are two discrete state variables: the X grid location and the Y grid location. There is only one state in grid world that has a positive reward, and the rest have negative rewards. The idea is that we want our agent to collect the most reward, which means getting to that positive reward in the fewest moves possible. The agent can only move one square at a time either up, down, left, or

right and to us powerful humans, it's easy to see exactly which way to go to get to the reward. However, we have to keep in mind that the agent knows nothing about the environment. It just knows that it can take one of four actions and it gets its location and reward back from the environment after it takes an action.

With a value function-based agent, a function would take in the state and one of the possible actions from that state, and output the value of taking that action. The value is the sum of the total discounted rewards from that state and on like we talked about in the first video. In this way, the policy would simply be to check the value of every possible action and choose the one with the highest value. We can think of this function as a critic since it's looking at the possible actions and it's criticizing the agent's choices. Since there are a finite number of states and actions in grid world, we could use a lookup table to represent this function. This is called the Q-table, where there is a value for every state and action pairing.

So how does the agent learn these values? Well, at first we can initialize it to all zeroes, so all actions look the same to the agent. This is where the exploration rate epsilon comes in and it allows the agent to just take a random action. After it takes that action, it gets to a new state and collects the reward from the environment. The agent uses that reward, that new information, to update the value of the action that it just took. And it does that using the famous Bellman equation.

The Bellman equation allows the agent to solve the Q-table over time by breaking up the whole problem into multiple simpler steps. So rather than solving the true value of the state/action pair in one step, through dynamic programming, the agent will update the state/action pair each time it's visited. Let me try to describe this equation in words so that hopefully it makes some sense.

After the agent has taken an action, it receives a reward. Value is more than the instant reward from an action; it's the maximum expected return into the future. Therefore, the value of the state/action pair is the reward that the agent just received, plus how much reward the agent expects to collect going forward. And we discount the future rewards by gamma so that, as we talked about, the agent doesn't rely too much on rewards far in the future. This is now the new value of the state/action pair, s, a. And so we compare this value to what was in the Q-table to get the error, or how far off the agent's prediction was. The error is multiplied by a learning rate and the resulting delta value is added to the old estimate.

When the agent finds itself back in the same state a second time, it's going to

have this updated value and it will tweak them again when it chooses the same action. And it'll keep doing this over and over again until the true value of every state/action pair is sufficiently known to exploit the optimal path.

Let's extend this idea to an inverted pendulum where there are still two states, angle and angular rate, except now the states are continuous. Value functions can handle continuous state spaces, except not with a lookup table. So we're going to need a neural network. The idea is the same. We input the state observations and an action, and the neural network returns a value.

You can see that this setup doesn't work well for continuous action spaces, because how could you possibly try every infinite action and find the maximum value? Even for a large action space this becomes computationally expensive, but I'm getting ahead of myself. For now, let's just say that the action space is discrete and the agent can choose one of 5 possible torque values. That seems reasonable.

So here's the idea. When you feed the network the observed state and an action it'll return a value, and our policy would again be to check the value for every possible action and take the one with the highest value. Just like with grid world, our neural network would be initially set to junk values and the learning algorithm would use a version of the Bellman equation to determine what the new value should be and update the weights and biases in the network accordingly. And once the agent has explored enough of the state space, then it's going to have a good approximation of the value function, and can select the optimal action, given any state.

That's pretty neat, right? But we did encounter a drawback in that the action space needs to be relatively small. But often in control problems we have a continuous action space—like being able to apply a continuous range of torques to an inverted pendulum problem.

This brings us to a merging of the two techniques into a class of algorithms called actor/critic. The actor is a network that is trying to take what it thinks is the best action given the current state, just like we had with the policy function method, and the critic is a second network that is trying to estimate the value of the state and the action that the actor took, like we had with the value-based methods. This works for continuous action spaces because the critic only needs to look at a single action, the one that the actor took, and not try to find the best action by evaluating all of them.

Here's basically how it works. The actor chooses an action in the same way that a policy function algorithm would, and it's applied to the environment. The critic estimates what it thinks the value of that state and action pair is and then it uses the reward from the environment to determine how accurate its value prediction was. The error is difference between the new estimated value of the previous state and the old value of the previous state from the critic network. The new estimated value is based on the received reward and the discounted value of the current state. It can use the error as a sense of whether things went better or worse than the critic expected.

The critic uses this error to update itself in the same way that the value function would so that it has a better prediction the next time it's in this state. The actor also updates itself with the response from the critic and, if available, the error term so that it can adjust its probabilities of taking that action again in the future.

That is, the policy now ascends the reward slope in the direction that the critic recommends rather than using the rewards directly.
So both the actor and the critic are neural networks that are trying to learn the optimal behavior. The actor is learning the right actions using feedback from the critic to know what a good action is and what is bad, and the critic is learning the value function from the received rewards so that it can properly criticize the action that the actor takes. This is why you might have to set up two neural networks in your agent; each one plays a very specific role.

With actor-critic methods, the agent can take advantage of the best parts of policy and value function algorithms. Actor-critics can handle both continuous state and action spaces, and speed up learning when the returned reward has high variance. I'll show an example of using an actor-critic-based algorithm to get a bipedal robot to walk in the next video.

Before I close out this video, I want to briefly discuss the last step in the RL workflow: deploying the algorithm on the target hardware. So far the agent has learned offline by interacting with a simulated environment. But once the policy is sufficiently optimal, then the learning would stop and the static policy would be deployed onto the target just like you would any developed control law. However, we also have the ability to deploy the reinforcement learning algorithms along with the policy and can continue learning on the target with the actual environment. This is important for environments that are hard to model accurately or that are slowly changing over time and therefore the agent needs to continue to learn occasionally so that it can adjust to those changes.