Now that we have an understanding of the reinforcement learning workflow, in this video I want to show how that workflow is put to use in getting a bipedal robot to walk using an RL-equipped agent. We're going to use the walking robot example from the MATLAB and Simulink Robotics Arena that you can find on GitHub. I've left a link to it in the description. This example comes with an environment model where you can adjust training parameters, train an agent, and visualize the results. In this video, we'll also look at how we can modify this example to make it look more like how we would set up a traditional control problem and then show some of the limitations of this design. So I hope you stick around for this because I think it'll help you understand how to use reinforcement learning for typical control applications. I'm Brian, and welcome to a MATLAB Tech Talk.

Let's start with a quick overview of the problem. The high-level goal is to get a two-legged robot to walk, somewhat like how a human would. Our job as designers is to determine the actions to take that correctly move the legs and body of the robot. The actions we can take are the motor torque commands for each joint; there's the left and right ankle, the left and right knee, and left and right hip joints. So there are six different torque commands that we need to send at any given time.

The robot body and legs, along with the world in which it operates, make up the environment. The observations from the environment are based on the type and locations of sensors as well as any other data that is generated by the software. For this example, we're using 31 different observations. These are the Y and Z body position, the X, Y, and Z body velocity, and the body orientation and angular rate. There are also the angles and angular rates of the six joints, and the contact forces between the feet and the ground. Those are the sensed observations. We're also feeding back the six actions that we commanded in the previous time step, which are stored in a buffer in software. So, in all, our control system takes in these 31 observations and has to calculate the values of six motor torques continuously. So you can start to see how complex the logic has to be for even this really simple system.

As I mentioned in the previous videos, rather than try to design the logic, the loops, the controllers, the parameters, and all of that stuff using the traditional tools of control theory, we can replace this whole massive function, end to end, with a reinforcement learning agent; one that uses an actor network to map these 31 observations to the six actions, and a critic network to make training the actor more efficient.

As we know, the training process requires a reward function—something that tells

the agent how it's doing so that it can learn from its actions. And I want to reason through what should exist in the reward function by thinking about the conditions that are important for a walking robot. This might be how you'd approach building up a reward function if you didn't know where to start. Now, I'll show you the results of training with this function as we create it, so you can see how the changes impact the solution; however, I'm not going to cover how to run the model because there is already a great video by Sebastian Castro that does that. So if you're interested in trying all of this out on your own, I'd recommend checking out the link in the description below. All right, so on to the reward.

Where to start? We obviously want the body of the robot to move forward; otherwise it will just stand there. But instead of distance, we can reward it for its forward velocity. That way there is a desire for the robot to walk faster rather than slower. After training with this reward, we can see that the robot dives forward to get that quick burst of speed at the beginning and then falls over and doesn't really make it anywhere. It may eventually figure out how to make it further with this reward, but it was taking a long time to converge and not making a lot of progress, so let's think about what we can add to help with training. We could penalize the robot for falling down so that diving forward isn't as attractive. So if it stays on its feet for a longer time, or if more sample times elapse before the simulation ends, then the agent should get more reward.

Let's see how this does. It's got a bit of hop at the beginning before ultimately falling down again. Perhaps if I had let this agent train longer, I could have a robot that jumps across the world like a frog, which is cool but that's not what I want. It's not enough that the robot moves forward and doesn't fall; we want some semblance of walking instead of hopping or crouch walking. So to fix this, we should also reward the agent for keeping the body as close to a standing height as possible.

Let's check out this reward function. Okay, this is looking better, but the solution isn't really natural looking. It stops occasionally to sort of jitter its legs back and forth, and most of the time it's dragging its right leg like a zombie and putting all of the actuation in the left leg. And this isn't ideal if we're concerned with actuator wear and tear or the amount of energy it takes to run this robot. We'd like both legs to do equal work and not to overuse the actuators with a lot of jittering. So to fix this, we can reward the agent for minimizing actuator effort. This should reduce extra jittering and balance the effort so that each leg has a share of the load.

Let's check out our trained agent. Okay, we are getting close here. This is looking

pretty good. Except now we have one final problem. We want to keep the robot moving in a straight line and not veering off to the right or left like it's doing here, so we should reward it for staying close to the x-axis.

This is our final reward and training with it needs about 3500 simulations. So if we set this up in our model and we unleash the simulation on a computer with multiple cores or a GPU or on a cluster of computers, then after a few hours of training, we'll have a solution. We'll have a robot that walks in a straight line in a human-like way.

With the reward function set, let's move onto the policy. I've already stated that the policy is an actor neural network and along with it is a critic neural network. And each of these networks have several hidden layers of hundreds of neurons each, so there are a lot of calculations that go into them. If we don't have enough neurons then the network will never be able to mimic the high dimensional function that is required to map the 31 observations to the six actions for this nonlinear environment. On the other hand, too many neurons and we spend more time training the excessive logic. In addition, the architecture of the network is really important in functional complexity. These are things like the number of layers, how they're connected, and the number of neurons in each layer. So there is some experience and knowledge needed to find the sweet spot that makes training possible and efficient.

Luckily, as we know, we don't need to manually solve for the hundred of thousands of weights and biases in our networks. We let the training algorithm do that for us. In this example, we're using an actor/critic training algorithm called the Deep Deterministic Policy Gradient, or DDPG. The reason is because this algorithm can learn with environments with continuous action spaces like we have with the continuous range of torques that we can apply to the motors. Also, since it estimates a deterministic policy, it's much faster to learn than one that learns a stochastic policy.

I know this all sounds fairly complicated and abstract, but the cool thing to me about this is that most of the complexity is there for training the policy. Once we have a fully trained agent, then all we have to do is deploy the actor network to the target hardware. Remember, the actor is the function that maps observations to actions; it's the thing that is deciding what to do, it's the policy. The critic and the learning algorithm are just there to help determine the parameters in the actor.

Okay, at this point here's a question that you might have. Sure, we can use RL to

get a robot to walk in a straight line; however, won't this policy do only this one thing? For instance, if I deploy this policy and turn on my robot, it's just instantly going to start walking straight, forever. So how can I learn a policy that will let me send commands to the robot to walk where I want it to walk?

Well let's think about that. Right now, this is what our system looks like. We have the environment that generates the observations and the reward, and then we have the agent that generates the actions. There's no way for us to inject any outside commands into this system and there's no way for our agent to respond to them even if we had them. So we need to write some additional logic, outside of the agent, that receives a reference signal and calculates an error term. The error is the difference between the current X position, which we can get from the environment, and the reference value. This is the same error calculation we would have in a normal feedback control system.

Now, rather than reward the agent for a higher velocity in the X direction, we can reward it for low error. This should incentivize the robot to walk toward and stay at the commanded x reference value.

For the observations, we need to give the agent a way to view the error term so that it can develop a policy accordingly. Since it might help our agent to have access to rate of change of error and maybe other higher derivatives, I'll feed in the error from the last five sample times. This will allow the policy to create derivatives if it needs to. Eventually the policy will be to walk forward at some specified rate if the error is positive, and backwards if it's negative.

Since we now have 36 observations into our agent, we need to adjust our actor network to handle the additional inputs. Again, check out Sebastian's video in the description if you need guidance on how to make these changes.

I've updated the default model in Simulink with the new error term and fed it into the observation block and the reward block. And I've trained this agent over thousands of episodes using this particular profile, so it should be really good at following this. But the hope is that the trained policy will be robust enough to follow other profiles as well that have similar rates and accelerations. So let's give it a shot. I'll have it walk forward, pause for a bit, and then walk backwards.

It's kind of funny looking when it walks backwards, but overall a pretty good effort. With some reward tweaking and maybe a little more time spent training, I might be on to something pretty good here.

So, in this way, you can start to see how we can use an RL agent to replace part of the control system. Instead a function that learns a single behavior, we can extract the high-level reference signal and have the agent work off error so that we can retain the ability to send commands.

We also can remove low-level functionality from the agent. For example, instead of the actions being the low-level torques for each of the six joints, the agent could just learn where to place its feet on the ground. So the action would be to place left foot at some location in the body coordinate frame. This action could be the reference command for a lower-level traditional control system that drives the joint motors. You know, something that might feedforward a torque command based on your knowledge of the dynamics of the system and feedback some signal to guarantee performance and stability.

This is beneficial because we can use our specific domain knowledge to solve the easy problems, and that will give us insight and control over the design, and then we can reserve reinforcement learning for the problems that are difficult.

Something to note about our final walking solution so far is that it's really only robust to its own state. You know, it can walk around without falling over, which is good, but only in a perfectly flat, featureless plain. It's not taking into account any part of the world outside the robot so it's actually quite a fragile design. For example, let's see what happens if we place a single obstacle in the way of our robot.

Well, that went as expected. The problem here is that we didn't give our agent any way of recognizing the state of the environment beyond the motions of the robot itself. There's nothing that can sense the obstacles and therefore nothing can be done to avoid them.

But here's the thing. The beauty of neural network-based agents is that they can handle what we'll call rich sensors. These are things like LiDAR and visible cameras, things that don't produce singular measurements like an angle, but rather return arrays of numbers that represent thousands of distances or pixels of varying light intensities. So we could install a visible camera and a LiDAR sensor on our robot and feed the thousand or so new values as additional observations into our agent. You can imagine how the complexity of this function needs to grow as our observations increase from 36 to thousands.

We may find that a simple, fully connected network is not ideal and so we may add additional layers that incorporate specialized logic that minimizes connections

like convolutional networks or adds memory like recurrent networks. These are network layers that are more adapted to dealing with large image data and more dynamic environments. However, we might not need to change the reward function in order to get the robot to avoid these obstacles. The agent could still learn that straying from the path, and therefore getting a lower reward here, allows the robot to continue walking without falling down, thereby earning more reward overall.

I've addressed a few of the problems with reinforcement learning in this video and showed how we can modify the problem by combining the benefits of traditional control design with reinforcement learning. We're going to expand on this some more in the next video, where we'll talk about other downsides of reinforcement learning and what we can do to mitigate them.