The first four videos in this series covered how great reinforcement learning is and how you can use it to solve some really hard control problems. So you may have this idea that you can essentially set up an environment, place an RL agent in it, and then let the computer solve your problem while you go off and drink a coffee or something. Unfortunately, even if you set up a perfect agent and a perfect environment and then the learning algorithm converges on a solution, there are still drawbacks to this method that we need to talk about. So in this video, I'm going to address a few possibly non-obvious problems with RL and try to provide some ways to mitigate them. Even if there aren't straightforward ways to address some of the challenges that you'll face, at the very least it'll get you thinking about them. So let's get to it. I'm Brian, and welcome to a MATLAB Tech Talk.

The problems that we'll address in this video come down to two main questions: The first is once you have a learned policy, is there a way to manually adjust it if it's not quite perfect? And the second is how do you know the solution is going to work in the first place?

Let's start with the first question. To answer this, let's assume that we've learned a policy that can get a robot to walk on two legs and we're about to deploy this static policy onto the target hardware. Think about what this policy is mathematically. It's made up of a neural network with possibly hundreds of thousands of weights and biases and nonlinear activation functions. The combination of these values and the structure of the network creates a complex function that maps high-level observations to low-level actions.

This function is essentially a black box to the designer. We may have an intuitive sense of how this function operates. You know the mathematics that convert the observations to actions. We may even understand some of the hidden features that this network has identified. However, we don't know the reason behind the value of any given weight or bias. So if the policy doesn't meet a specification or if the plant or the rest of the operating environment changes, how do you adjust the policy to address that? Which weight or bias do you change? The problem is that the very thing that has made solving the problem easier—namely, condensing all the difficult logic down to a single black box function—has made our final solution incomprehensible. Since a human didn't craft this function and doesn't know every bit of it, it's difficult to manually target problem areas and fix them.

Now, there is active research that is trying to push the concept of explainable artificial intelligence. This is the idea that you can set up your network so that it can be easily understood and audited by humans. However, at the moment, the

majority of RL-generated policies still would be categorized as a black box, where the designer can't explain why the output is the way it is. Therefore, at least for the time being, we need to learn how to deal with this situation.

Contrast this with a traditionally designed control system where there is typically a hierarchy with loops and cascaded controllers, each designed to control a very specific dynamic quality of the system. Think about how gains might be derived from physical properties like appendage lengths or motor constants. And how simple it is to change those gains if the physical system changes.

In addition, if the system doesn't behave the way you expect, with a traditional design you can often pinpoint the problem to a specific controller or loop and focus your analysis there. You have the ability to isolate a controller and test and modify it standalone to ensure it's performing under the specified conditions, and then bring that controller back into the larger system.

This is really difficult to do when the solution is a monolithic collection of neurons and weights and biases. So, if we end up with a policy that isn't quite right, rather than being able to fix the offending part of the policy, we have to redesign the agent or the environment model and then train it again. This cycle of redesigning, training, and testing, and redesigning, training, and testing can be time consuming. But there is a larger issue looming here that goes beyond the length of time it takes to train an agent. It comes down to the needed accuracy of the environment model.

The issue is that it's really hard to develop a sufficiently realistic model—one that takes into account all the important dynamics in the system as well as disturbances and noise. At some point, it's not going to perfectly reflect reality. It's why we still have to do physical tests rather than just verify everything with the model.

Now, if we use this imperfect model to design a traditional control system, then there is a chance that our design won't work perfectly on the real hardware and we'll have to make a change. Since we can understand the functions that we created, we're able to tune our controllers and make the necessary adjustments to the system.

However, with a neural network policy, we don't have that luxury. And since we can't really build a perfect model, any training we do with it will be not quite correct. Therefore, the only option is to finish training the agent on the physical hardware. Which, as we've discussed in previous videos, can be challenging in its

own right.

A good way to reduce the scale of this problem is simply to narrow the scope of the RL agent. Like I showed in the last video, rather than learn a policy that takes in the highest-level observations and commands the lowest-level actions, we can wrap traditional controllers around an RL agent so it only solves a very specialized problem. By targeting a smaller problem with an RL agent, we shrink the unexplainable black box to just the parts of the system that are too difficult to solve with traditional methods.

Shrinking the agent obviously doesn't solve our problem, it just decreases its complexity. The policy is more focused so it's easier to understand what it's doing, the time to train is reduced, and the environment doesn't need to include as much of the dynamics. However, even with this, we're still left with the second question: How do you know the RL agent will work regardless of its size? For example, is it robust to uncertainties? Are there stability guarantees? And can you verify that the system will meet the specifications?

To answer this, let's again start with how would we go about verifying a traditional control system. One of the most common ways is just through test. Like we talked about, this is testing using a simulation and a model as well as with the physical hardware, and we verify that the system meets the specifications—that is, it does the right thing—across the whole state space and in the presence of disturbances and hardware failures.

And we can do this same level of verification testing with an RL-generated policy. Again, if we find a problem we have to retrain the policy to fix it, but testing the policy appears to be similar. However, there are some pretty important differences that make testing a neural network policy difficult.

For one, with a learned policy, it's hard to predict how the system will behave in one state based on its behavior in another. For example, if we train an agent to control the speed of an electric motor by having it learn to follow a step input from 0 to 100 RPM, we can't be certain, without testing, that that same policy will follow a similar step input from 0 to 150 RPM. This is true even if the motor behaves linearly. This slight change may cause a completely different set of neurons to activate and produce an undesired result. We won't know that unless we test it.

With traditional methods, if the motor behaves linearly, then we have some guarantees that all step inputs within the linear range will behave similarly. We

don't need to run each of those tests independently. So the number of tests we have to run with an RL policy is typically larger than with traditional methods.

A second reason verification through test is difficult with neural network policies is because of the potential for extremely large and hard-to-define input spaces. Remember, one of the benefits of deep neural networks is that they can handle rich sensors like images from a camera. For example, if you are trying to verify that your system can sense an obstacle using an image, think about how many different ways an obstacle can appear. If the image has thousands of pixels, and each pixel can range from 0 to 255, think about how many permutations of numbers that works out to and how impossible it would be to test them all. And just like with the step input example, just because your network has learned to recognize an obstacle in one portion of the image at some lighting condition and orientation and at some scale, it doesn't give you any guarantees that it works in any other way in the image.

The last difficulty I want to bring up is with formal verification. These methods involve guaranteeing that some condition will be met by providing a formal proof rather than using test.

In fact, with formal verification, we can make a claim that a specification will be met even if we can't make that claim through testing. For example, with the motor speed controller, we tested a step input from 0 to 100 RPM and from 0 to 150. But what about all the others, like 50 to 300, or 75-80? Even if we sampled 10,000 speed combinations, it won't guarantee every combination will work because we can't test all of them. It just reduces risk. But if we had a way to inspect the code or perform some mathematical verification that covered the whole range, then it wouldn't matter that we couldn't test every combination. We'd still have confidence that they would work.

For example, we don't have to test to make sure a signal will always be positive if the absolute value of that signal is performed in the software. We can verify it simply by inspecting the code and showing that the condition will always be met. Other types of formal verification include calculating robustness and stability factors like gain and phase margins.

But for neural networks, this type of formal verification is more difficult, or even impossible in some cases. For the reasons I explained earlier, it's hard to inspect the code and make any guarantees about how it will behave. We also don't have methods to determine its robustness or its stability, and we can't reason through what will happen if a sensor or actuator fails. It all comes back to the fact that

we can't explain what the function is doing internally.

Okay, so these are some of the ways that learned neural networks make design difficult. It's hard to verify their performance over a range of specifications. If the system does fail, it's hard to pinpoint the source of the failure, and then, even if you can pinpoint the source, it's hard to manually adjust the parameters or the structure, leaving you with the only option of redesigning and starting the training process over again.

But I don't want to destroy your confidence in reinforcement learning as a useful tool for production systems, because, with all that being said, there are ways to set up learning so that the resulting policy is more robust in the presence of uncertainties. There are also ways to increase safety and make reinforcement learning a viable option for production systems. And we can even use reinforcement learning to solve a slightly different problem, one that avoids a lot of these issues in the first place. So let's end this video on a positive note and talk about all three of these. We'll start with making the policy more robust.

Even though we can't quantify robustness, we can make the system more robust by actively adjusting parameters in the environment while the agent is learning.
For example, with our walking robot, let's say manufacturing tolerances cause the maximum torque for a joint motor to fall between 2 and 2.1 Nm. We are going to be building dozens of these robots and we'd like to learn a single policy for all robots that is robust to these variations. We can do this by training the agent in an environment where the motor torque value is adjusted each time the simulation is run. We could uniformly choose a different max torque value in that range at the start of each episode so that over time, the policy will converge to something that is robust to these manufacturing tolerances. By tweaking all the important parameters in this way—you know, things like appendage lengths, delays in the system, obstacles, and reference signals and so on—we will end up with an overall robust design. We may not be able to claim a specific gain or phase margin, but we will have more confidence that the result can handle a wider range within the operational state space.

This addresses the robustness but it still doesn't give us any guarantees that the policy will do the right thing on the hardware. And one thing we don't want is for the hardware to get damaged, or someone to get hurt because of an unpredictable policy. So we need to also increase the overall safety of the system. And one way we can increase safety is by determining situations that you want the system to avoid no matter what and then building software outside of the policy that monitors for that situation. If that monitor triggers, then constrain the system or

take over and place it into some kind of safe mode before it has a chance to cause damage. This doesn't prevent you from deploying a dangerous policy, but it will protect the system, allowing you to learn how it fails and adjusting the reward and training environment to address that failure.

Both the fix to increase robustness and safety are kind of workarounds to the limitations that we have with a learned neural network policy. However, there is a way to use reinforcement learning and still be able to take advantage of the result being as robust, safe, changeable, and verifiable as a traditionally architected control system. And that is by using it simply as an optimization tool for a traditionally architected control system. Let me explain it this way. Imagine designing an architecture with dozens of nested loops and controllers, each with several gains. You can end up with a situation where you have 100 or more individual gain values to tune. Rather than try to manually tune each of these gains by hand, you could set up an RL agent to learn the best values for all of them at once.

The agent would be rewarded for how well the system performs and how much effort it takes to get that performance and then the actions would be the hundred or so gains in the system. So, when you initially kick off training, the randomly initialized neural network in the agent would just generate random values, and then you'd run the simulation using those for the control gains. Now more than likely the first episode would produce some garbage result, but after each episode, the learning algorithm would tweak the neural network in a way that the gains move in the direction that increases reward—that is, it improves performance and lowers effort.

The nice part about using reinforcement learning in this way is that once you've learned an optimal set of control gains, you're done; you don't need anything else. We don't have to deploy any neural networks or verify them or worry about having to change them. We just need to code the final static gain values into our system. In this way, you still have a traditionally architected system (one that can be verified and manually adjusted on the hardware just like we're used to) but you populated it with gain values that were optimally selected using reinforcement learning. Sort of a best-of-both-worlds approach.

So hopefully you can see that reinforcement learning is really powerful for solving hard problems, and that it's definitely worth learning and figuring out how to combine it with traditional approaches in a way that you feel comfortable with the final product. There are some challenges that exist regarding understanding the solution and verifying that it will work, but as we covered a bit, we do have a few

ways right now to work around those challenges.

Here's the thought I want to leave you with. Learning algorithms, RL design tools like MATLAB, and verification methods are advancing all the time. We are nowhere near reinforcement learning's full potential. Perhaps it won't far into the future before it becomes the first design method of choice for all complex control systems. Thanks for watching this video.