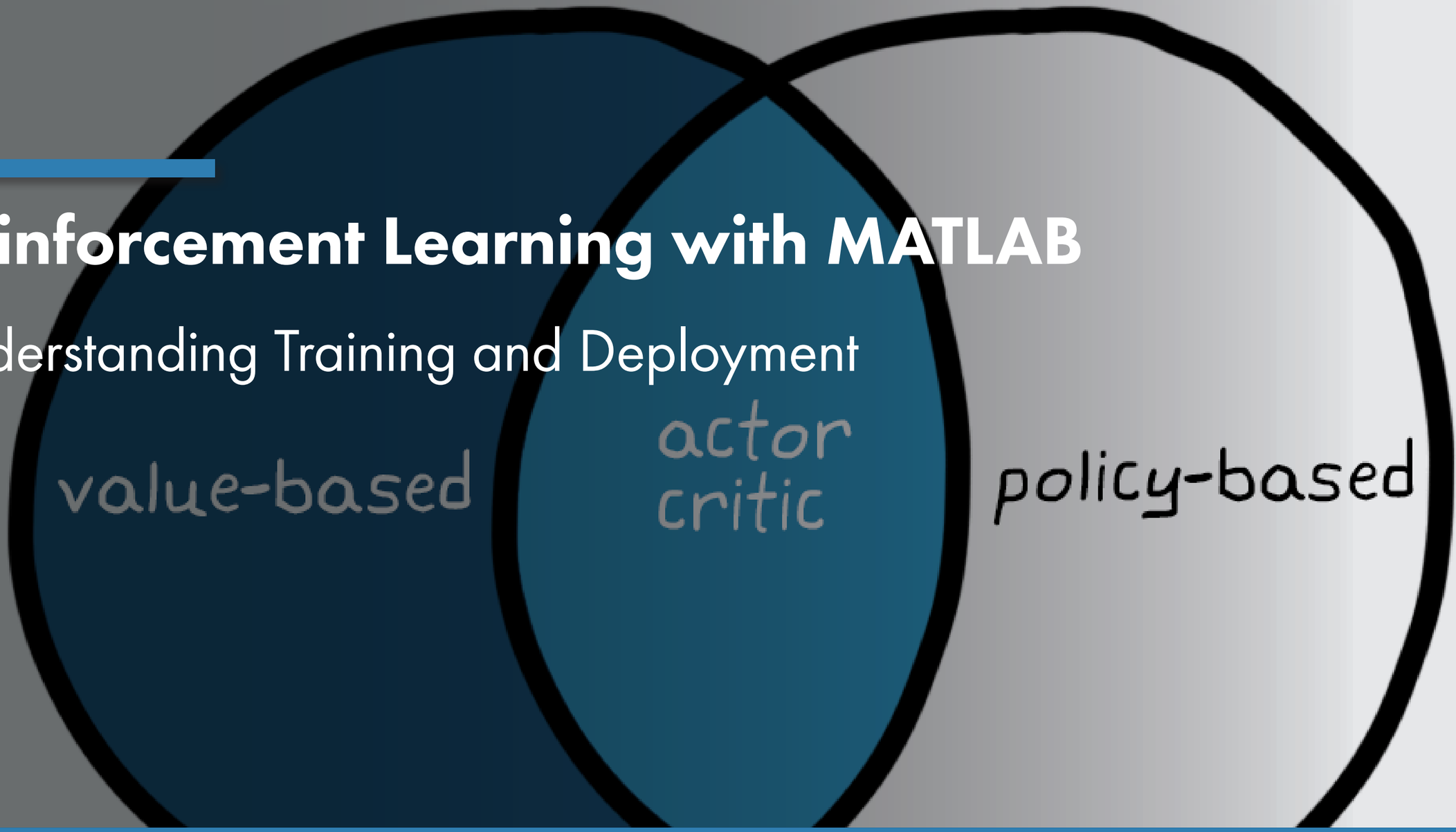
Reinforcement Learning with MATLAB

Understanding Training and Deployment



MathWorks®

Reinforcement Learning Workflow Overview

This ebook series addresses the five areas of reinforcement learning. It covers concepts and then shows how you can do it in MATLAB® and Simulink®.

The first ebook focuses on *setting up the environment*. The second explores *rewards and policy structures*. This ebook covers ***training and deployment***.

1 You need an environment where your agent can learn. You need to choose what should exist within the environment and whether it's a simulation or a physical setup.

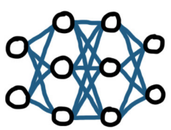
environment

2 You need to think about what you ultimately want your agent to do and craft a reward function that will incentivize the agent to do just that.

reward

3 You need to choose a way to represent the policy. Consider how you want to structure the parameters and logic that make up the decision-making part of the agent.

policy

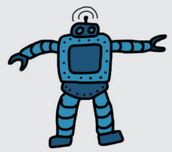


4 You need to choose an algorithm to train the agent that works to find the optimal policy parameters.

training

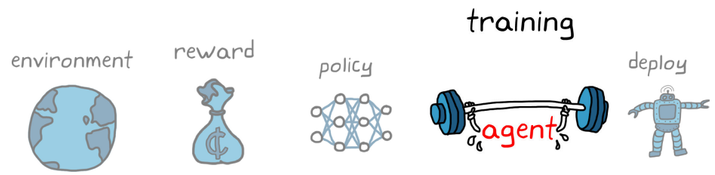
5 Finally, you need to exploit the policy by deploying it in the field and verifying the results.

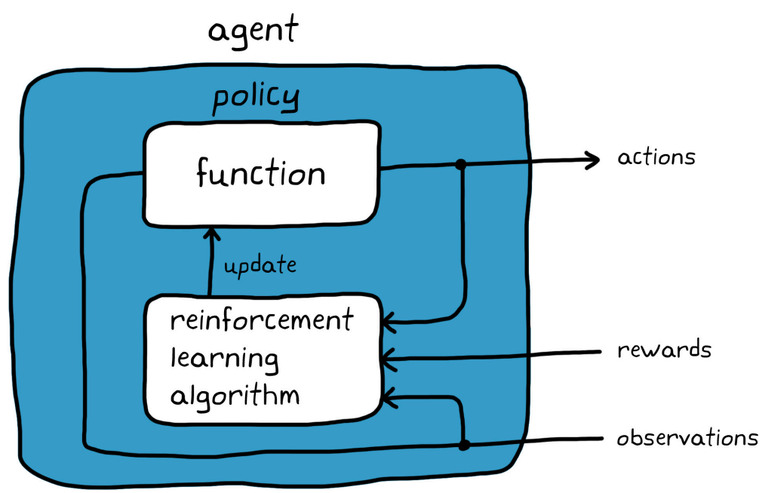
deploy



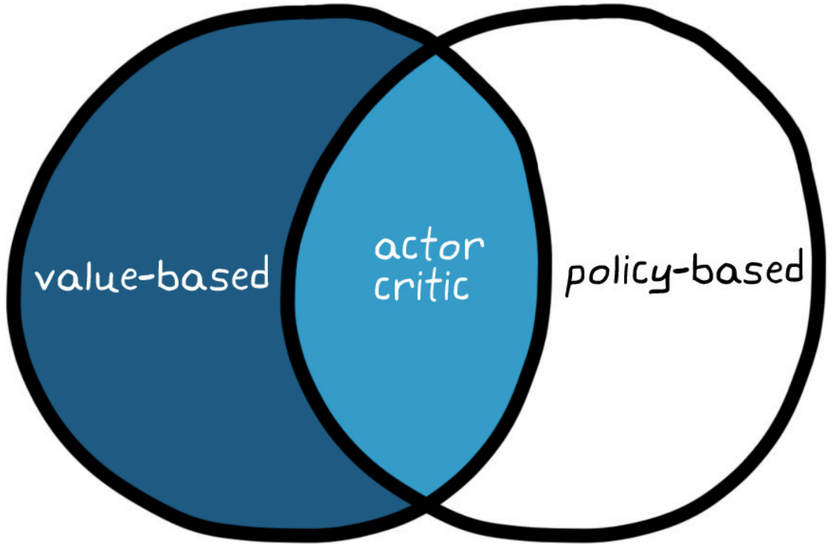
How the Policy Is Structured

In a reinforcement learning (RL) algorithm, neural networks represent the policy in the agent. The policy structure and the reinforcement learning algorithm are intimately intertwined; you can't structure the policy without also choosing the RL algorithm.



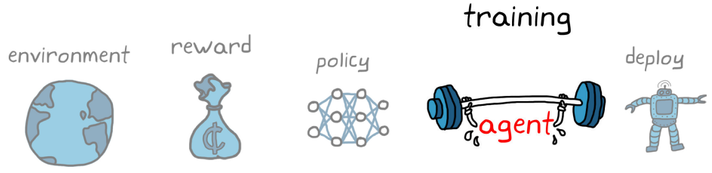


the policy structure and the RL algorithm are intimately intertwined



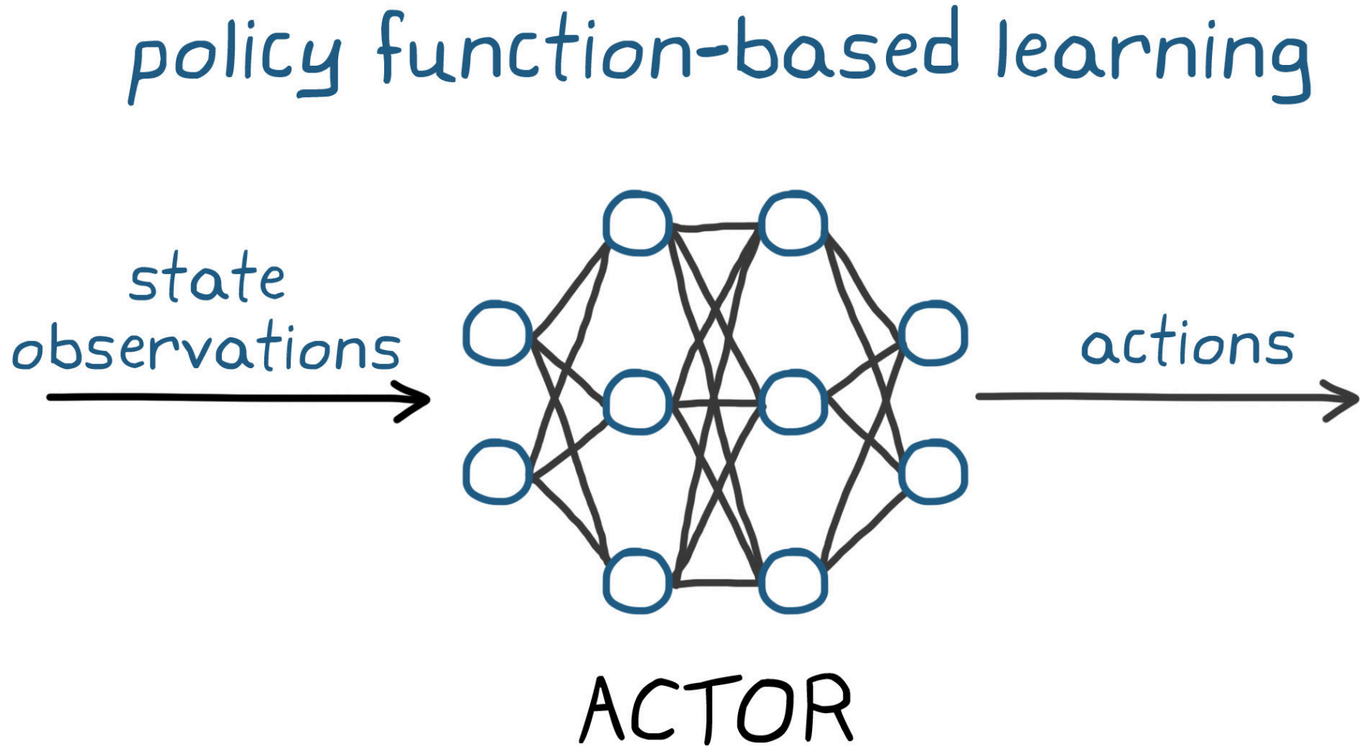
The next few pages will describe policy function–based, value function–based, and actor-critic approaches to reinforcement learning to highlight the differences in the policy structures. This will definitely be an oversimplification, but if you want a basic understanding of the ways policies can be structured, it should help you get started.

Policy Function–Based Learning

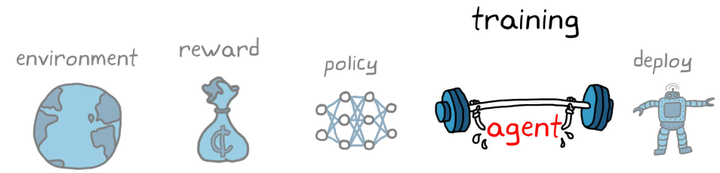


Policy function–based learning algorithms train a neural network that takes in the state observations and outputs actions. This neural network is the entire policy—hence the name *policy function–based algorithms*. The neural network is called the *actor* because it directly tells the agent which actions to take.

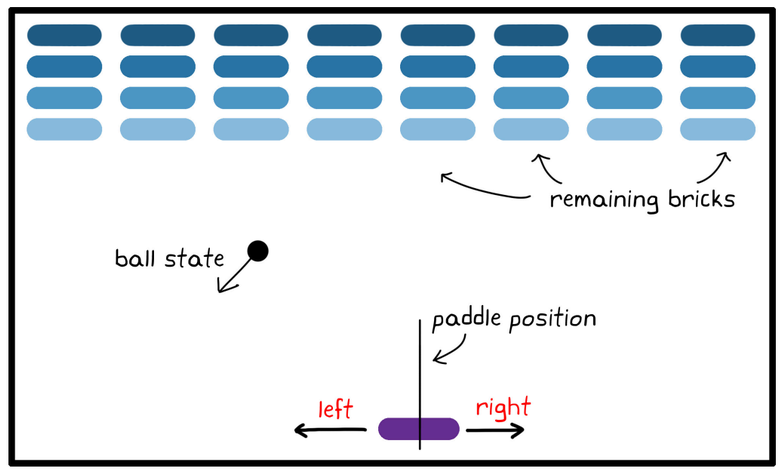
The question now is, how do we approach training this neural network? To get a general feel for how this is done, take a look at the Atari game Breakout.

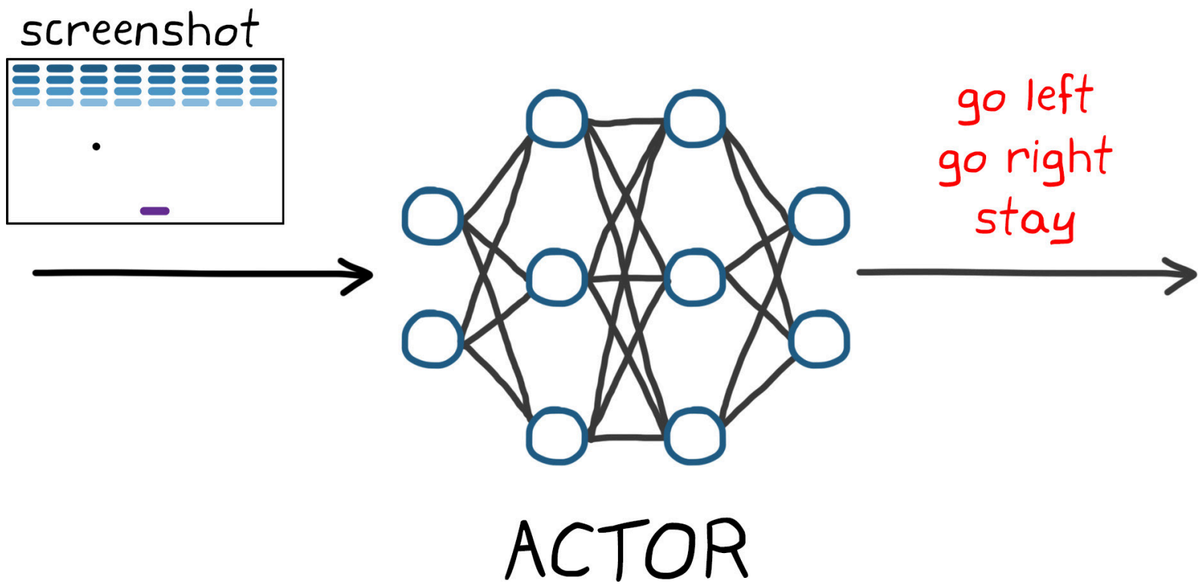


The Policy Approach to Learning Breakout



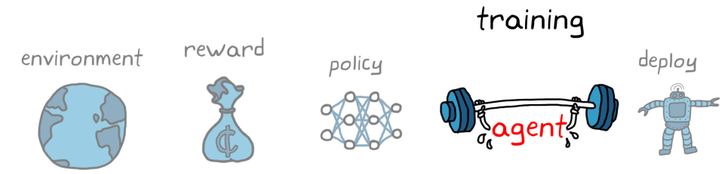
Breakout is a game in which you try to eliminate bricks using a paddle to direct a bouncing ball. The game has three actions, move the paddle left, right, or not at all, and a near-continuous state space that includes the position of the paddle, the position and velocity of the ball, and the location of the remaining bricks.



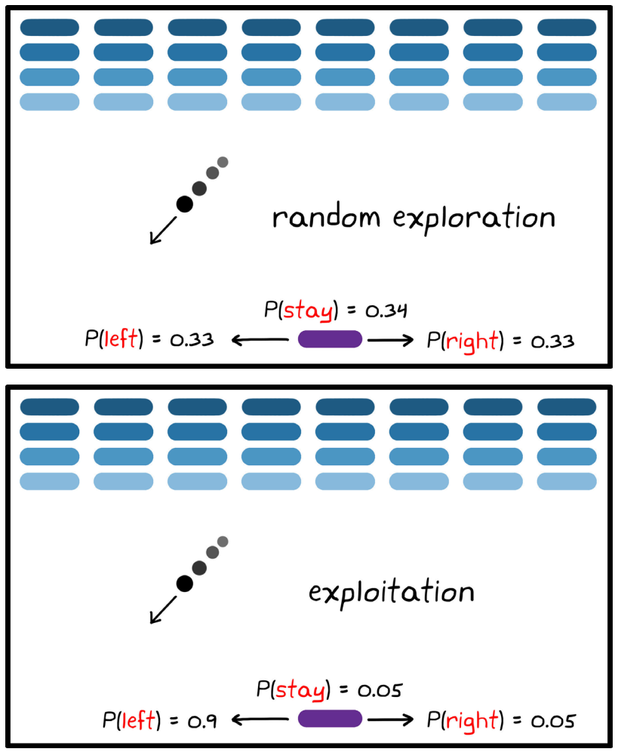


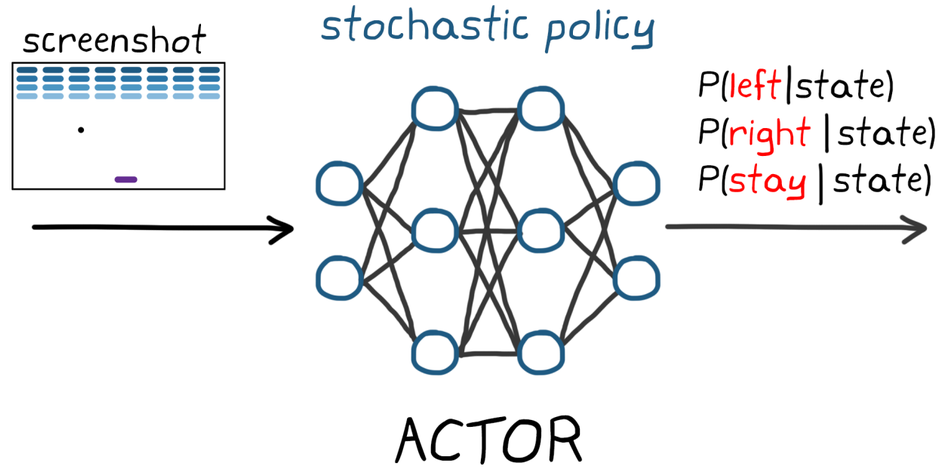
In this example, the inputs into the actor network are the states of the paddle, ball, and blocks. The outputs are nodes representing the actions: left, right, and stay. Rather than calculating the states manually and feeding them into the network, you can input a screen shot of the game and let the network learn which features in the image are the most important to base its output on. The actor would map the intensity of thousands of pixels to the three outputs.

A Stochastic Policy



Once the network is set, it's time to look at approaches to training it. One broad approach that itself has a lot of variations is policy gradient methods. Policy gradient methods can work with a stochastic policy, so rather than producing a deterministic "Take a left," the policy would output the probability of taking a left. The probabilities are directly related to the value of the three output nodes.



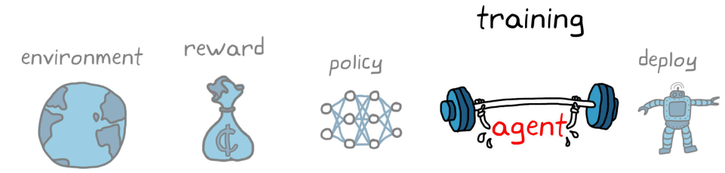


Should the agent exploit the environment by choosing the actions that collect the most rewards that it already knows about, or should it choose actions that explore parts of the environment that are still unknown?

A stochastic policy addresses this tradeoff by building exploration into the probabilities. Now, when the agent learns, it just needs to update the probabilities. Is taking a left a better option than taking a right? If so, push the probability that you take a left in this state higher.

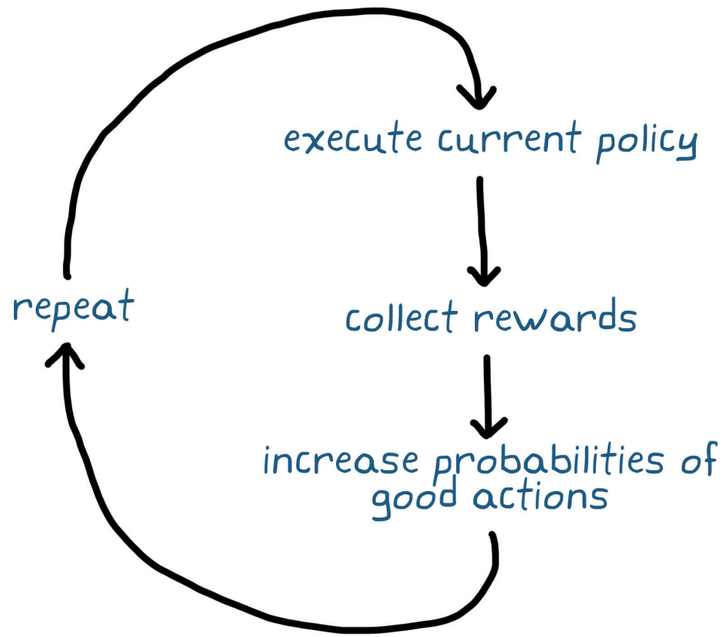
Over time, the agent will nudge these probabilities in the direction that produces the most reward. Eventually, the most advantageous action for every state will have such a high probability that the agent always takes that action.

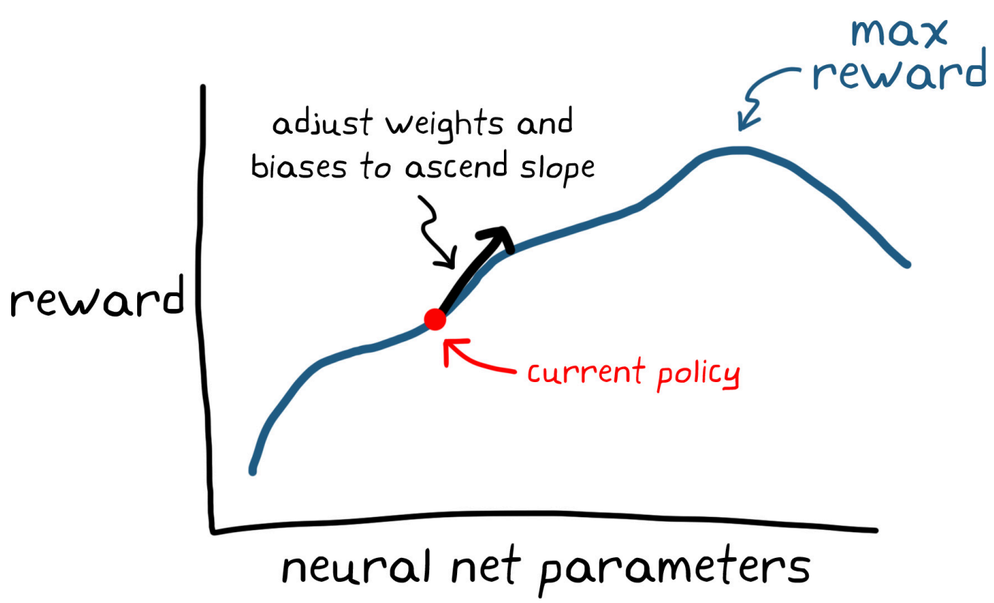
Policy Gradient Methods



So how does the agent know whether the actions were good or not? The idea is this: execute the current policy, collect reward along the way, and then update the network to increase the probabilities of actions that led to higher rewards.

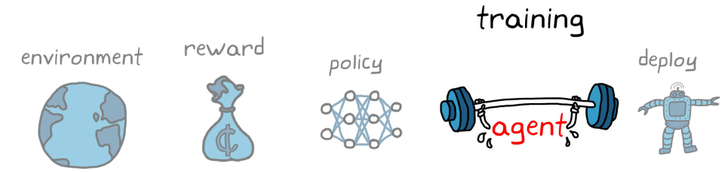
If the paddle went left, missing the ball and causing a negative reward, then change the neural network to increase the probability of moving the paddle right next time the agent is in that state.



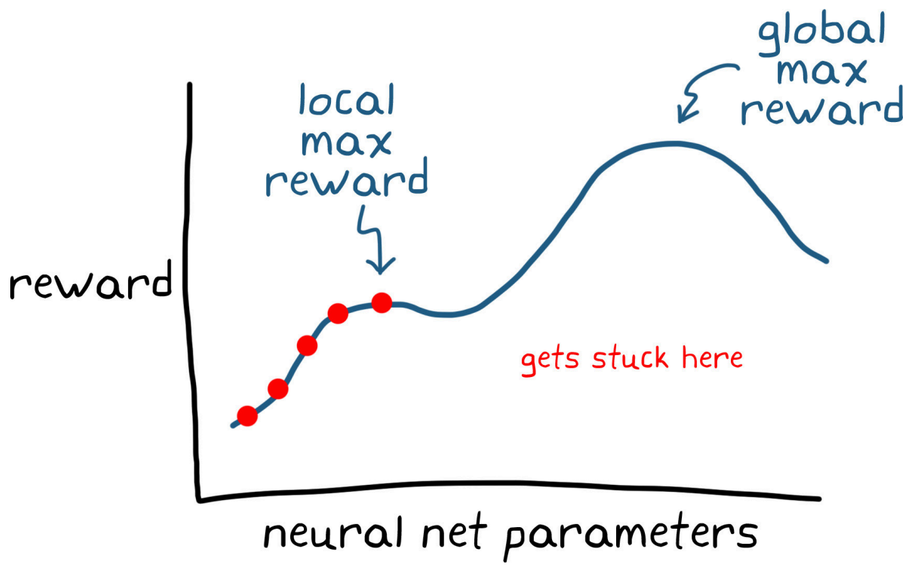


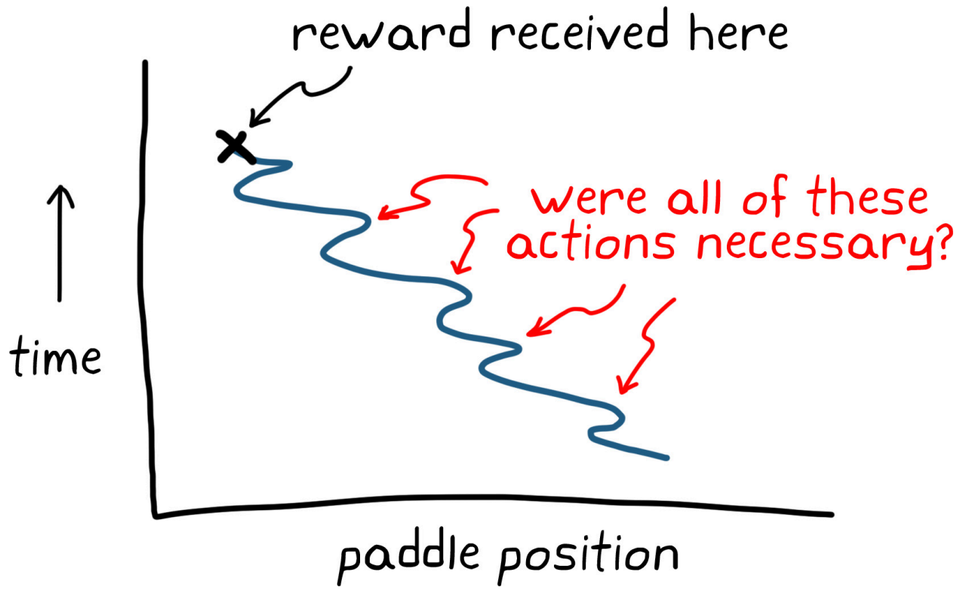
You take the derivative of each weight and bias in the network with respect to reward, and adjust them in the direction of a positive reward increase. In this way, the learning algorithm is moving the weights and biases of the network to ascend up the reward slope. This is why the term *gradient* is used in the name.

The Downside of Policy Gradient Methods



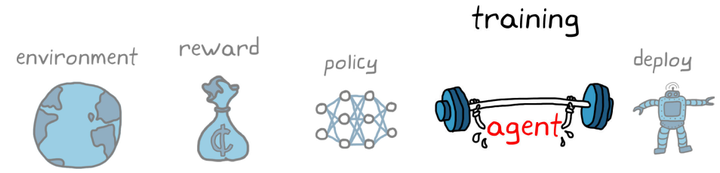
One of the downsides of policy gradient methods is that the naive approach of just following the direction of steepest ascent can converge on a local maxima rather than global. Policy gradient methods can also converge slowly due to their sensitivity to noisy measurements, which happens, for example, when it takes a lot of sequential actions to receive a reward and the resulting cumulative reward has high variance between episodes.



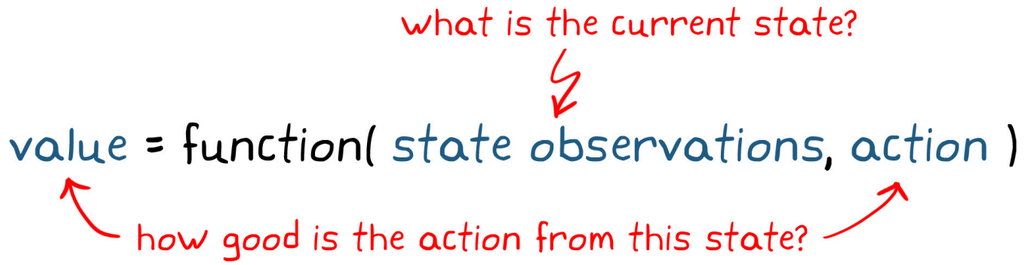


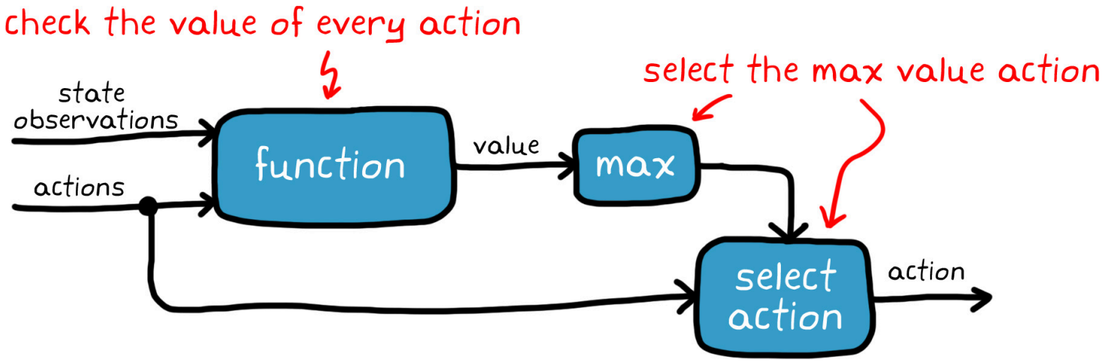
For example, in Breakout the agent might make a lot of quick left and right paddle movements as the paddle ultimately works its way across the field to strike the ball and receive a reward. The agent wouldn't know if every single one of those actions was actually required to get that reward, so the policy gradient algorithm would have to treat each action as though it was necessary and adjust the probabilities accordingly.

Value Function–Based Learning



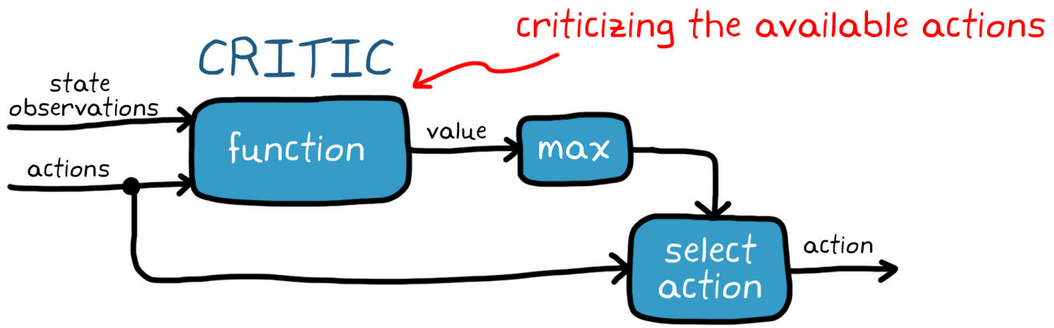
With a value function–based agent, a function would take in the state and one of the possible actions from that state, and output the value of taking that action.



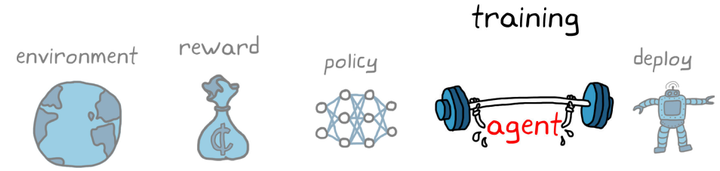


This function alone is not enough to represent the policy since it outputs a value and the policy needs to output an action. Therefore, the policy would be to use this function to check the **value** of every possible **action** from a given state and choose the action with the highest value.

You can think of this function as a critic since it's looking at the possible actions and criticizing the agent's choices.



Value Functions and Grid World



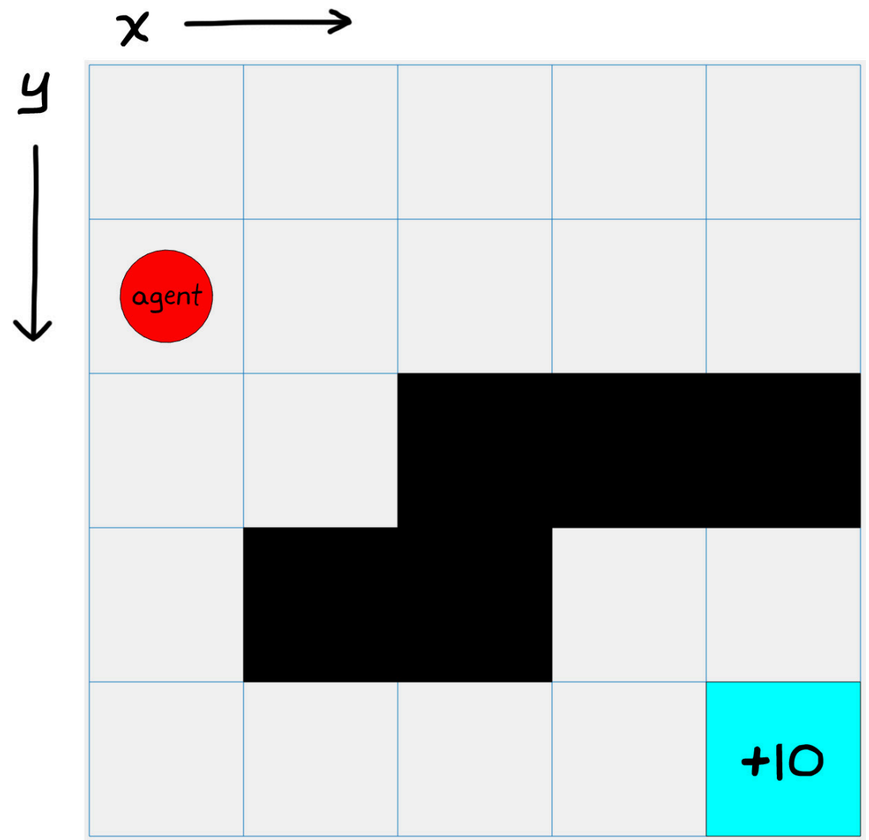
To see how this would work in practice, here's an example using the Grid World environment.

In this environment, there are two discrete state variables: the X grid location and the Y grid location. The agent can only move one square at a time either up, down, left, or right, and each action it takes results in a reward of -1.

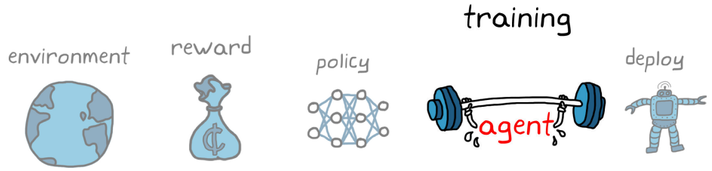
If the agent tries to move off the grid or into one of the black obstacles, then the agent doesn't move into the new state but the -1 reward is still received. In this way, the agent is penalized for essentially running into a wall and it doesn't make any physical progress for that effort.

There is one state that produces a +10 reward; the idea is that in order to collect the most reward, the agent needs to learn the policy that will get it to the +10 in the fewest moves possible.

» *See how to solve a Grid World environment in MATLAB*

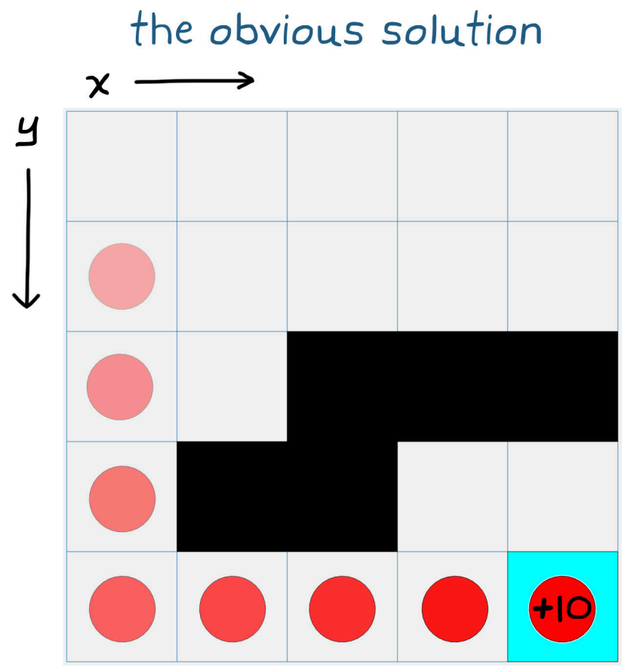


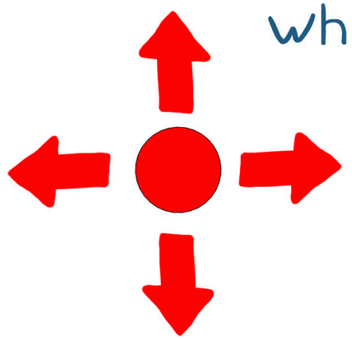
Value Functions and Grid World *Continued*



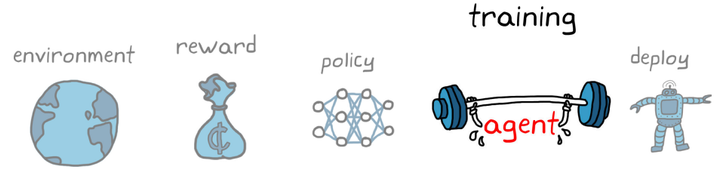
It might seem easy to determine exactly which route to take to get to the reward.

However, you have to keep in mind that in model-free RL, the agent knows nothing about the environment. It doesn't know that it's trying to get to the +10. It just knows that it can take one of four actions, and it receives its location and reward back from the environment after it takes an action.



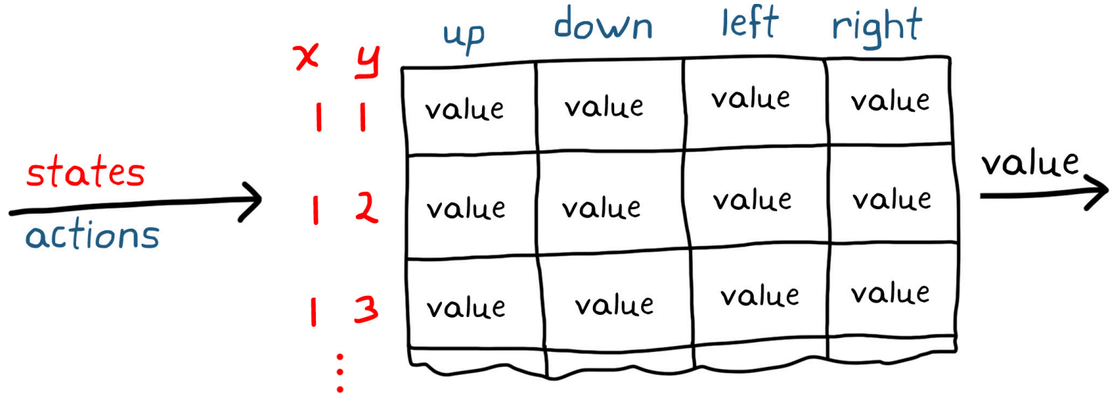


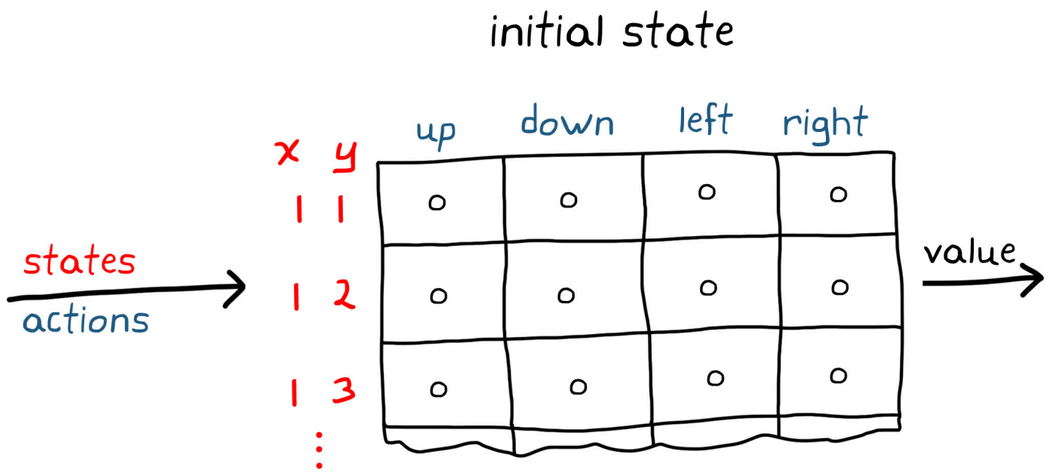
Solving Grid World with Q-Tables



The way the agent builds up knowledge of the environment is by taking actions and learning the values of that state/action pair based on the received reward. Since there are a finite number of states and actions in grid world, you can use a Q-table to map them to values.

So how does the agent learn these values? Through a process called Q-learning.

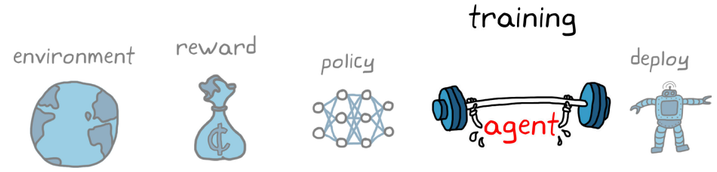




With Q-learning, you can start by initializing the table to zeros, so all actions look the same to the agent. After the agent takes a random action, it gets to a new state and collects the reward from the environment.

The agent uses that reward as new information to update the value of the previous state and the action that it just took using the famous Bellman equation.

The Bellman Equation



$$\text{new } Q(s, a) = Q(s, a) + \alpha\left[R(s, a) + \gamma \cdot \max Q'(s', a') - Q(s, a)\right]$$

The Bellman equation allows the agent to solve the Q-table over time by breaking up the whole problem into multiple simpler steps. Rather than solving the true value of a state/action pair in one step, the agent will update the value each time a state/action pair is visited through dynamic programming. The Bellman equation is important for Q-learning as well as other learning algorithms, such as DQN. Here's a closer look at the specifics of each term in the equation.

After the agent has taken an action **a** from state **s**, it receives a reward.

$$\text{new } Q(s, a) = Q(s, a) + \alpha\left[\mathbf{R(s, a)} + \gamma \cdot \max Q'(s', a') - Q(s, a)\right]$$

reward for taking action, a, from state, s

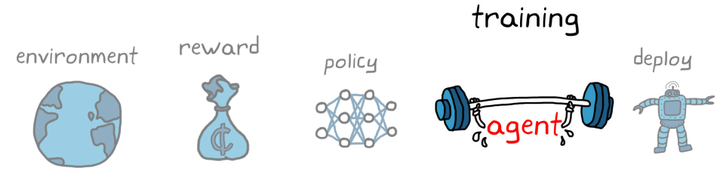
Value is more than the instant reward from an action; it's the maximum expected return into the future. Therefore, the value of the state/action pair is the reward that the agent just received, plus how much reward the agent expects to collect going forward.

$$\text{new } Q(s, a) = Q(s, a) + \alpha\left[R(s, a) + \gamma \cdot \underbrace{\max Q'(s', a')}_{\text{maximum expected value from state, s'}} - Q(s, a)\right]$$

You discount the future rewards by gamma so that the agent doesn't rely too much on rewards far in the future. Gamma is a number between 0 (looks at no future rewards to assess value) and 1 (looks at rewards infinitely far into the future).

$$\text{new } Q(s, a) = Q(s, a) + \alpha\left[R(s, a) + \underbrace{\gamma}_{\text{discount future rewards}} \cdot \max Q'(s', a') - Q(s, a)\right]$$

The Bellman Equation *Continued*



The sum is now the new value of the state and action pair (**s**, **a**), and we compare it with the previous estimate to get the error.

$$\text{new } Q(s,a) = Q(s,a) + \alpha\left[\underbrace{R(s,a) + \gamma \cdot \max Q'(s',a')}_{\text{new best estimate of value}} - \overbrace{Q(s,a)}^{\text{previous estimate of value}}\right]$$

The error is multiplied by a learning rate that gives you control over whether to replace the old value estimate with the new (alpha = 1), or nudge the old value in the direction of the new (alpha < 1).

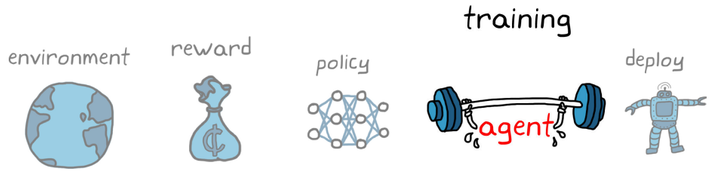
$$\text{new } Q(s,a) = Q(s,a) + \underbrace{\alpha}_{\text{error is multiplied by learning rate}}\left[R(s,a) + \gamma \cdot \max Q'(s',a') - Q(s,a)\right]$$

Finally, the resulting delta value is added to the old estimate and the Q-table has been updated.

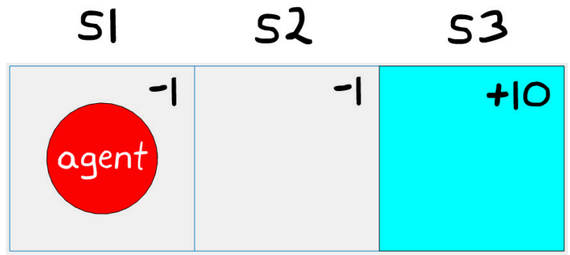
$$\text{new } Q(s,a) = \underbrace{Q(s,a)}_{\text{delta value is added to old estimate}} + \alpha\left[R(s,a) + \gamma \cdot \max Q'(s',a') - Q(s,a)\right]$$

The Bellman equation is another connection between reinforcement learning and traditional control theory. If you are familiar with optimal control theory, you may notice that this equation is the discrete version of the Hamilton-Jacobi-Bellman equation, which, when solved over the entire state space, is a necessary and sufficient condition for an optimum.

The Bellman Equation *Continued*



It might be helpful to see the Bellman equation in action by looking at the first few steps in a simple Grid World example. In this example, alpha is set to 1 and gamma is set to 0.9. If both actions have the same value, then the agent takes a random action; otherwise, the agent chooses the action with the highest value.



Episode	Step	State	current Q(s, a)	Action	R(s, a)	new Q(s, a)
1	1	S1	S1: Left = 0, Right = 0; S2: Left = 0, Right = 0; S3: Left = 0, Right = 0	right (random)	-1	S1: Left = 0, Right = -1; S2: Left = 0, Right = 0; S3: Left = 0, Right = 0
1	2	S2	S1: Left = 0, Right = 0; S2: Left = 0, Right = 0; S3: Left = 0, Right = 0	right (random)	+10	S1: Left = 0, Right = -1; S2: Left = 0, Right = 10; S3: Left = 0, Right = 0

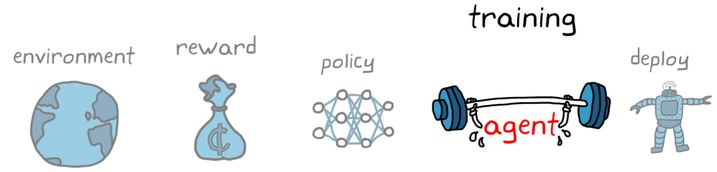
Bellman equation: $0 + 1 \cdot [-1 + 0.9 \cdot 0 - 0] = -1$

Bellman equation: $0 + 1 \cdot [10 + 0.9 \cdot 0 - 0] = +10$

End of episode

When the agent reaches the termination state, **S3**, the episode ends and the agent reinitializes at the starting state, **S1**.
The Q-table values persist and the learning continues into the next episode, which is continued on the next page.

The Bellman Equation *Continued*



Episode	Step	State	current Q(s, a)	Action	R(s, a)	new Q(s, a)
2	1	S1		left (greedy)	-1	

	S1	S2	S3
Left	**0**	0	0
Right	**-1**	10	0

	S1	S2	S3
Left	**-1**	0	0
Right	**-1**	10	0

Bellman equation: $0 + 1 \cdot [-1 + 0.9 \cdot 0 - 0] = -1$

Episode	Step	State	current Q(s, a)	Action	R(s, a)	new Q(s, a)
2	2	S1		right (random)	-1	

	S1	S2	S3
Left	**-1**	0	0
Right	**-1**	10	0

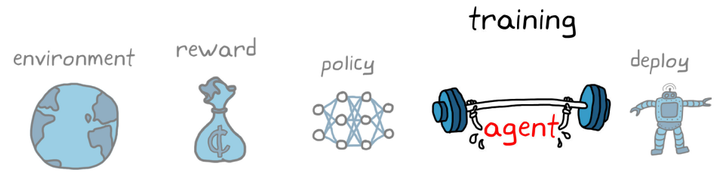
	S1	S2	S3
Left	-1	0	0
Right	**8**	10	0

Bellman equation: $-1 + 1 \cdot [-1 + 0.9 \cdot 10 - (-1)] = +8$

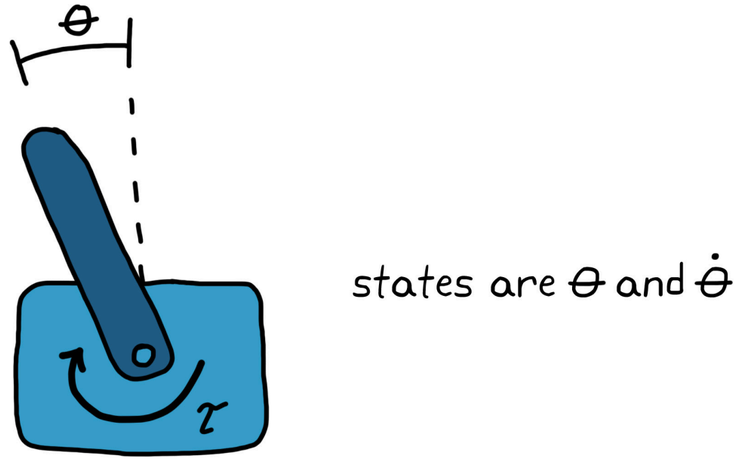
End of episode

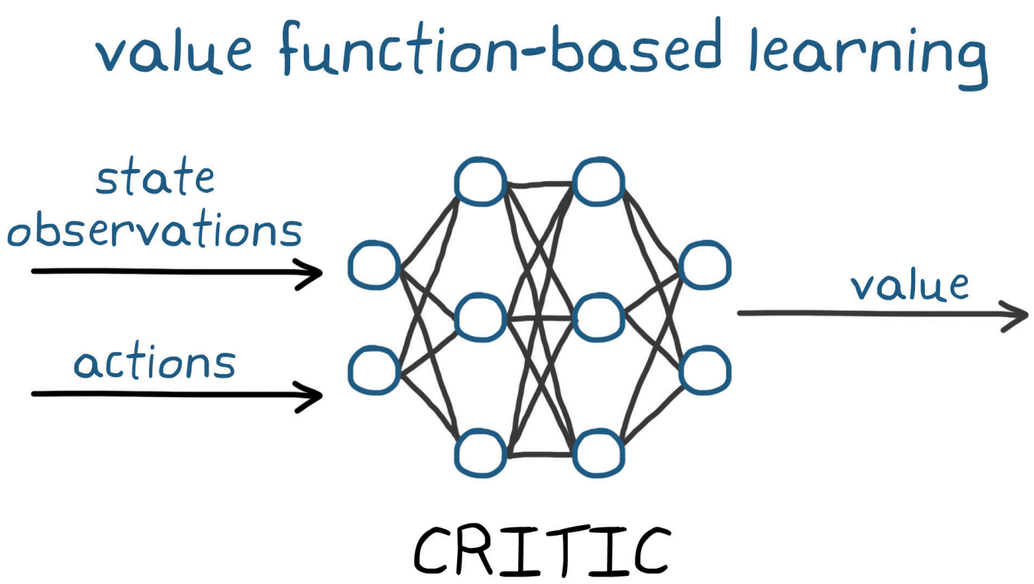
Within just four actions, the agent has already settled on a Q-table that produces the optimal policy; in state **S1**, it will take a right since the value 8 is higher than -1, and in state **S2**, it will take a right again since the value 10 is higher than 0. What's interesting about this result is that the Q-table hasn't settled on the true values of each state/action pair. If it keeps learning, the values will continue to move in the direction of the actual values. However, you don't need to find the true values in order to produce the optimal policy; you just need the value of the optimal action to be the highest number.

The Critic as a Neural Network



Extend this idea to an inverted pendulum. Like Grid World, there are two states, angle and angular rate, except now the states are continuous.

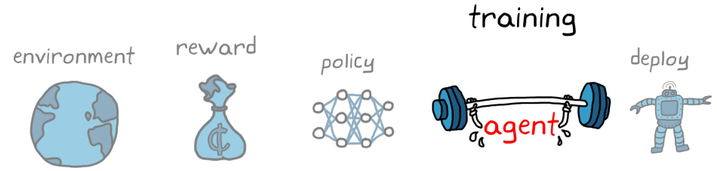




The value function (the critic) is represented with a neural network. The idea is the same as with a table: You input the state observations and an action, the neural network returns the value of that state/action pair, and the policy is to choose the action with the highest value.

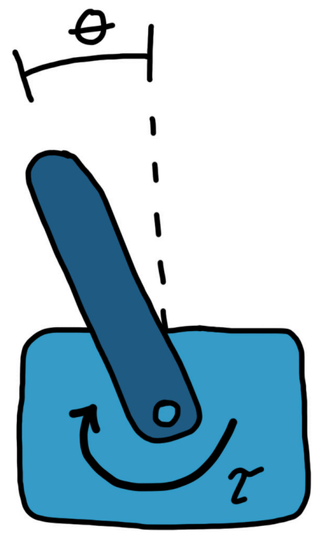
Over time, the network would slowly converge on a function that outputs the true value for every action anywhere in the continuous state space.

The Downside of Value-Based Policies



You can use a neural network to define the value function for continuous **state** spaces. If the inverted pendulum has a discrete action space, you can feed discrete actions into your critic network one at a time.

Value function–based policies won't work well for continuous **action** spaces. This is because there is no way to calculate the value one at a time for every infinite action to find the maximum value. Even for a large (but not infinite) action space, this becomes computationally expensive. This is unfortunate because often in control problems you have a continuous action space, such as applying a continuous range of torques to an inverted pendulum problem.



continuous states: θ and $\dot{\theta}$

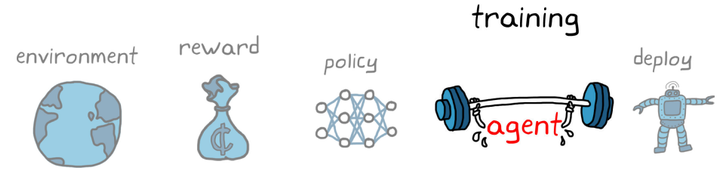
discrete action space: τ = [-2, -1, 0, 1, 2] Nm

So what can you do?

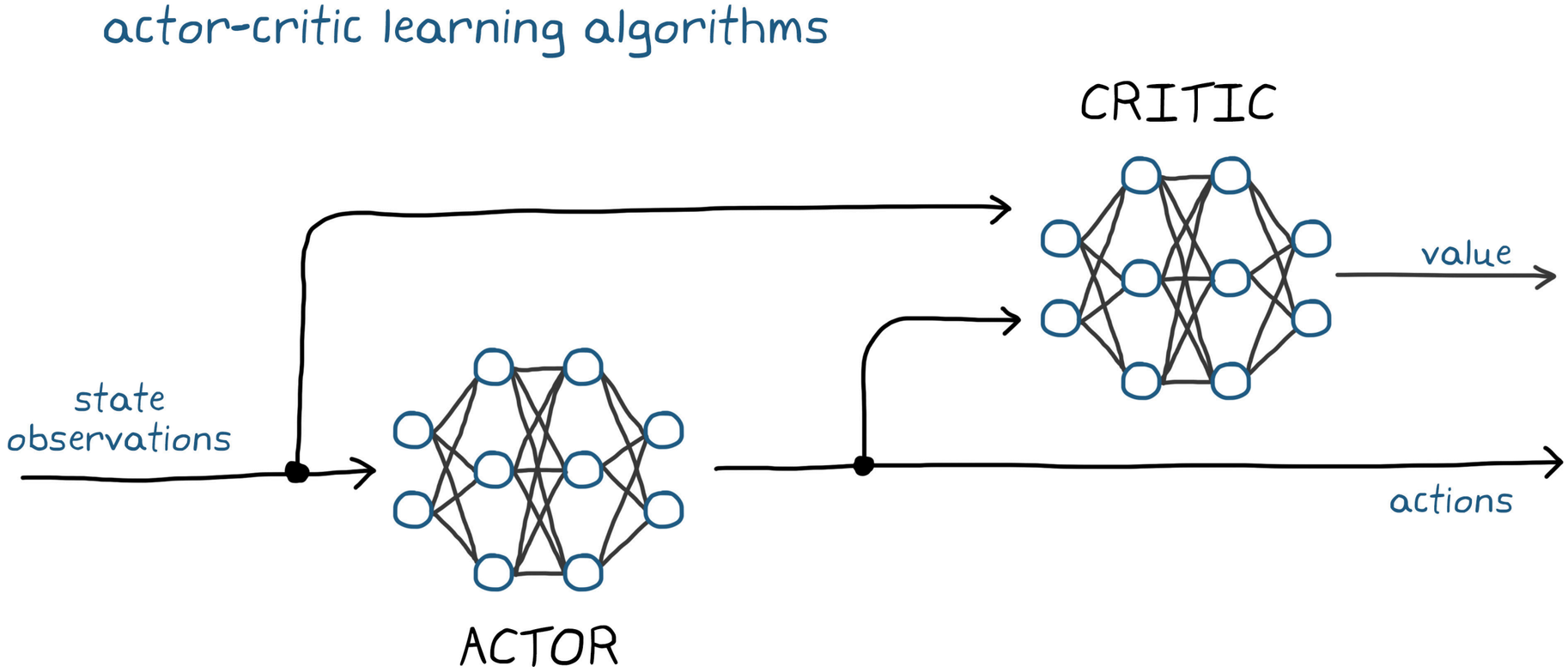
You can implement a vanilla policy gradient method, as covered in the policy function–based algorithm section. These algorithms can handle continuous action spaces, but they have trouble converging when there is high variance in the rewards and the gradient is noisy. Alternatively, you can merge the two learning techniques into a class of algorithms called *actor-critic*.

» *See how to train an actor-critic agent to balance an inverted pendulum in MATLAB*

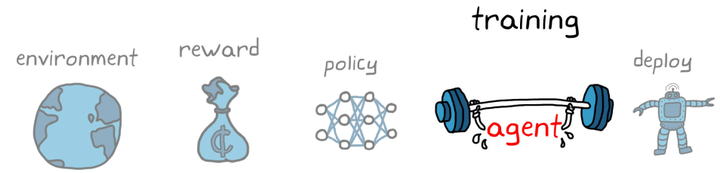
Actor-Critic Methods



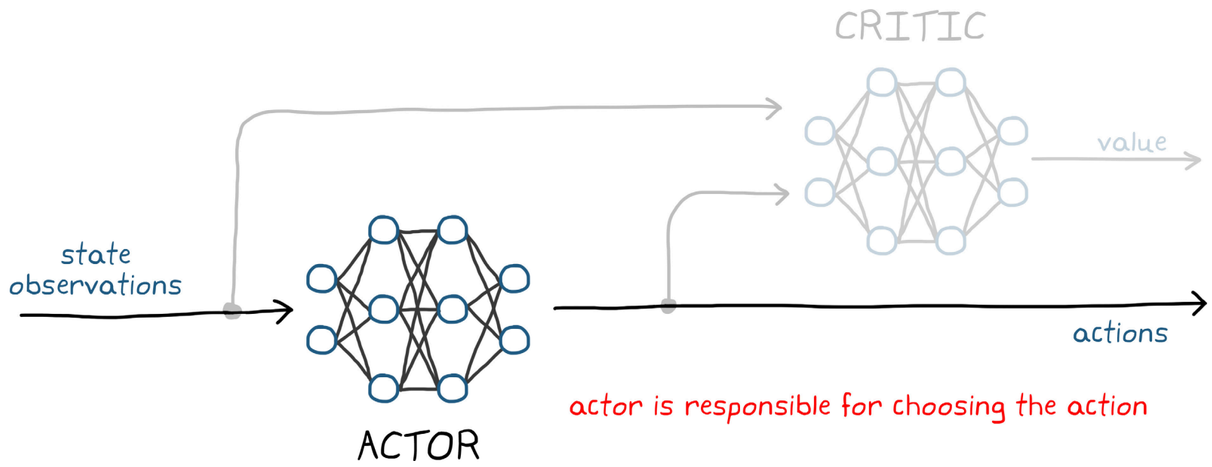
The *actor* is a network that is trying to take what it thinks is the best action given the current state, as seen with the policy function method. The *critic* is a second network that is trying to estimate the value of the state and the action that the actor took, as seen with the value function method. This approach works for continuous action spaces because the critic only needs to look at the single action that the actor took and does not need to try to find the best action by evaluating all of them.

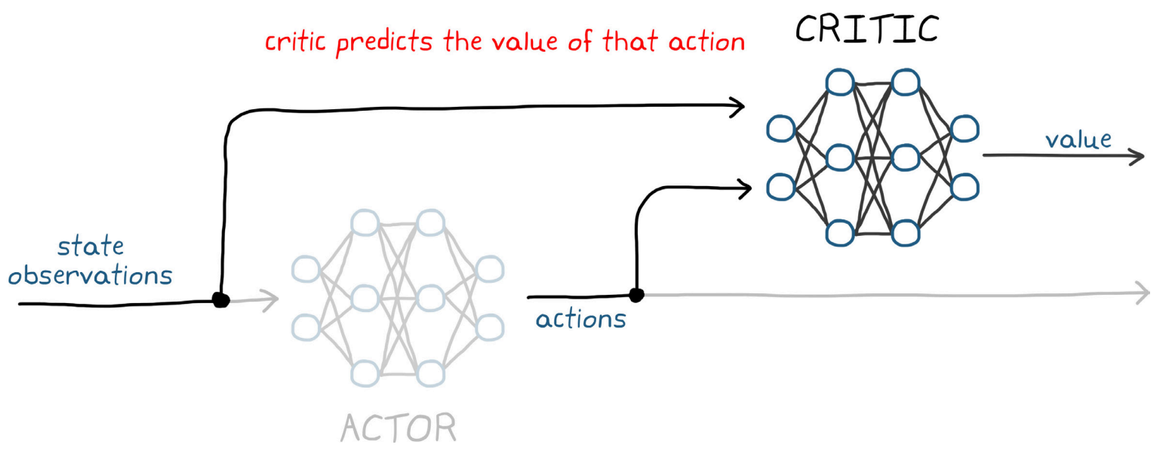


The Actor-Critic Learning Cycle



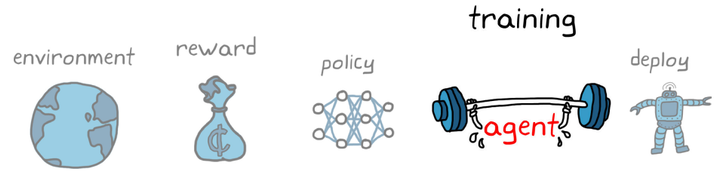
The actor chooses an action in the same way that a policy function algorithm would, and it is applied to the environment.



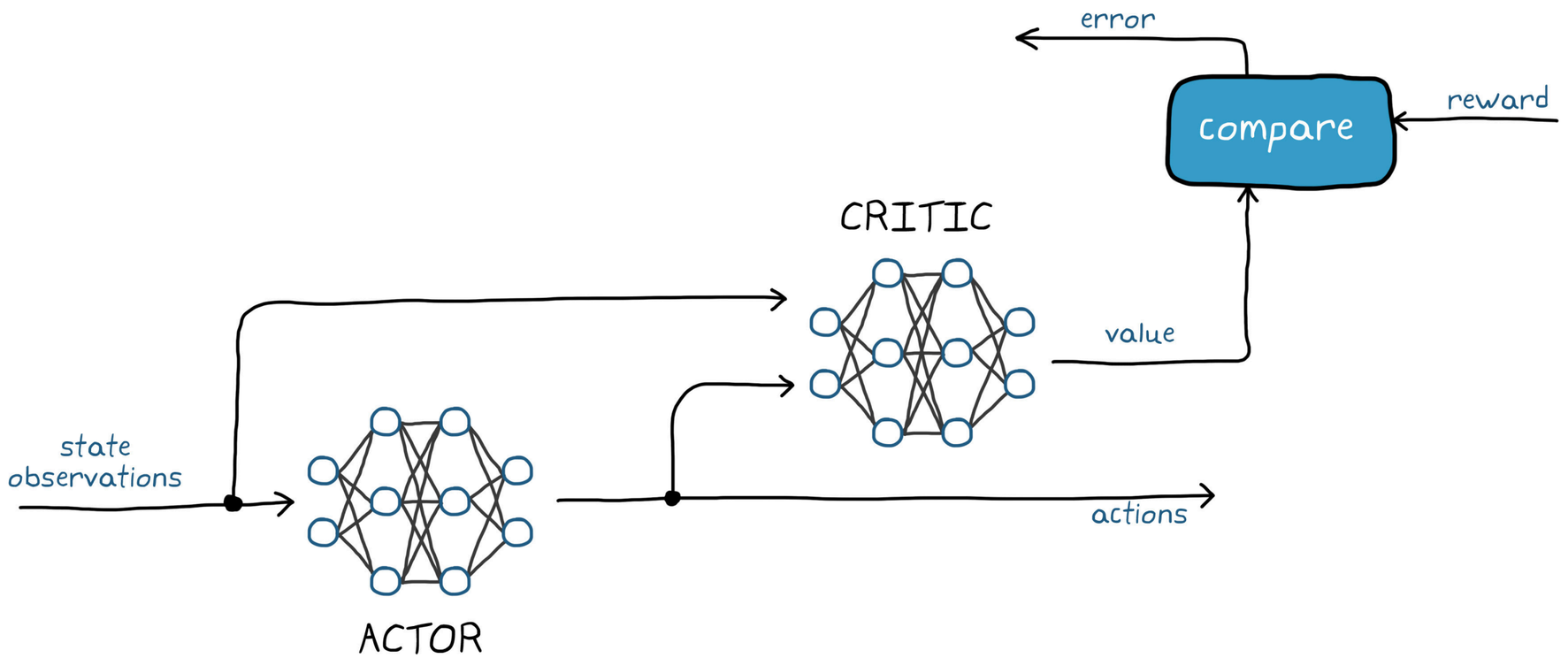


The critic makes a prediction of what the value of that action is for the current state and action pair.

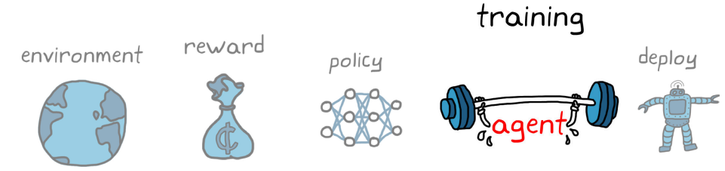
The Actor-Critic Learning Cycle *Continued*



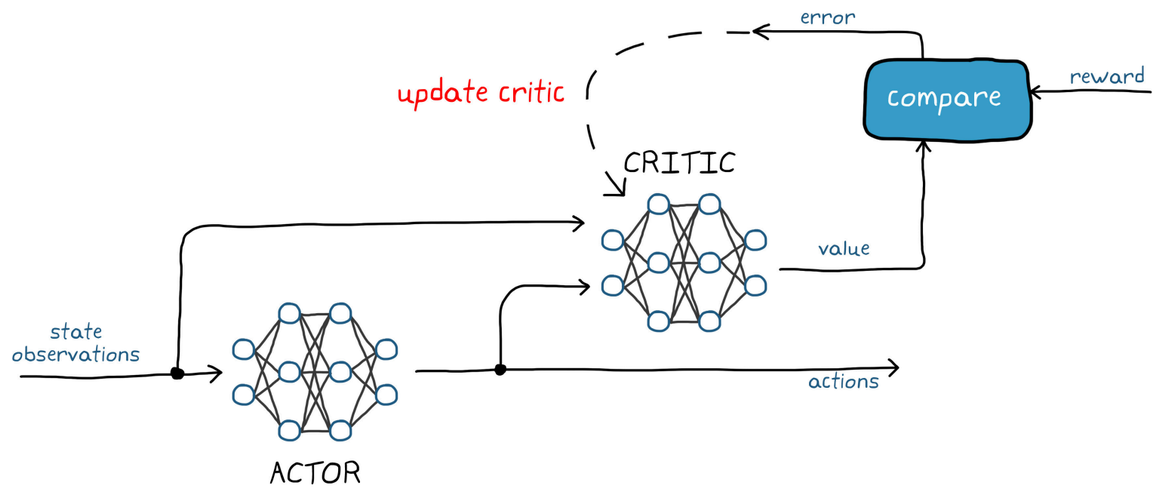
The critic then uses the reward from the environment to determine the accuracy of its value prediction. The error is the difference between the new estimated value of the previous state and the old value of the previous state from the critic network. The new estimated value is based on the received reward and the discounted value of the current state. The error gives the critic a sense of whether things went better or worse than it expected.

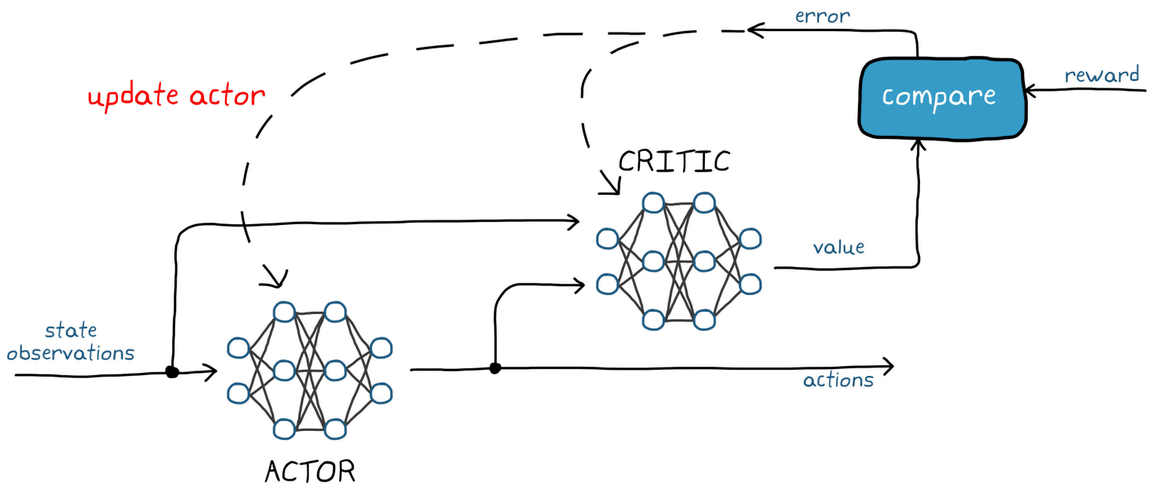


The Actor-Critic Learning Cycle *Continued*



The critic uses this error to update itself in the same way that a value function would so that it has a better prediction the next time it's in this state.

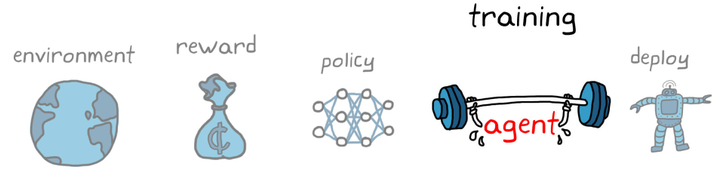




The actor also updates itself with the response from the critic so that it can adjust its probabilities of taking that action again in the future.

In this way, the policy now ascends the reward slope in the direction that the critic recommends rather than using the rewards directly.

Two Complementing Networks

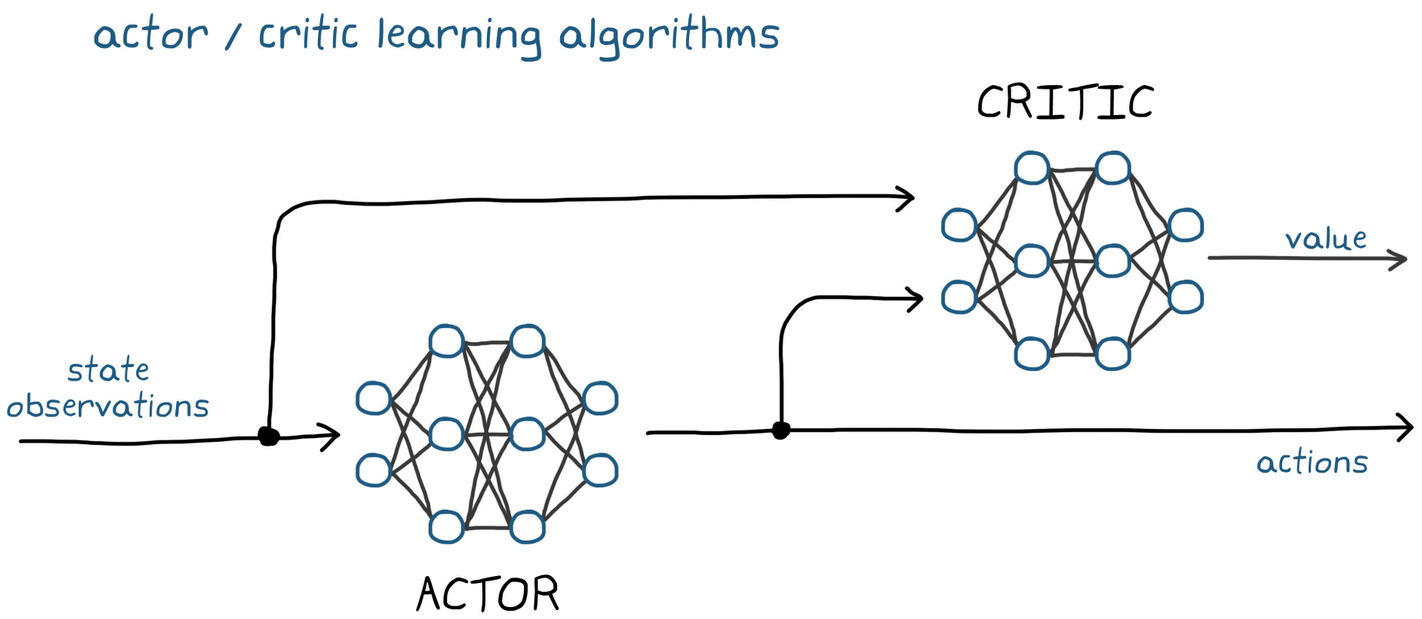


A lot of different types of learning algorithms use an actor-critic policy; this ebook generalizes these concepts to remain algorithm agnostic.

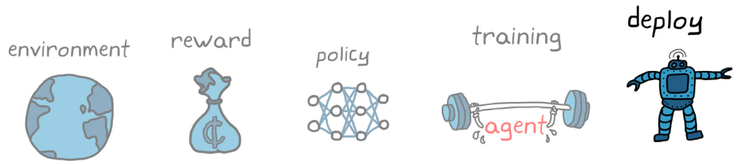
The actor and critic are neural networks that try to learn the optimal behavior. The actor is learning the correct actions to take using feedback from the critic to know which actions are good and bad, and the critic is learning the value function from the received rewards so that it can properly criticize the action that the actor takes.

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.

Hopefully, it's now clear why you may have to set up two neural networks when creating your agent; each one plays a very specific role.

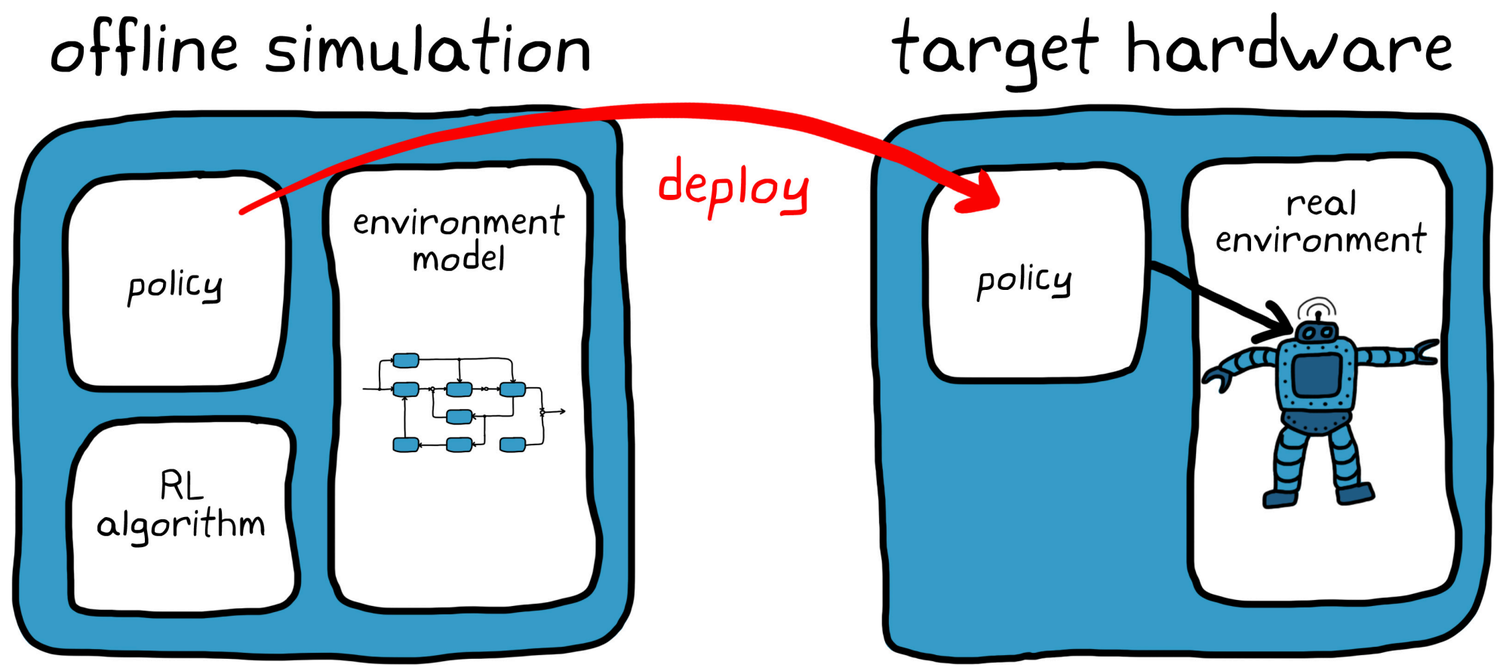


Policy Deployment

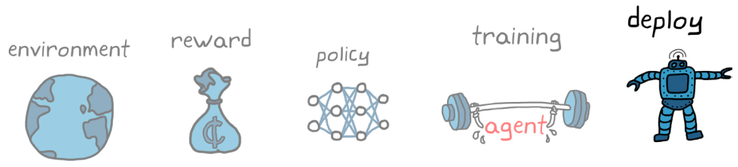


The last step in the reinforcement learning workflow is to deploy the policy.

If learning is done with the physical agent in the real environment, then the learned policy is already on the agent and can be exploited. This ebook has assumed that the agent has learned offline by interacting with a simulated environment. Once the policy is sufficiently optimal, the learning process can be stopped and the static policy deployed onto any number of targets, just like you would deploy any traditionally developed control law.



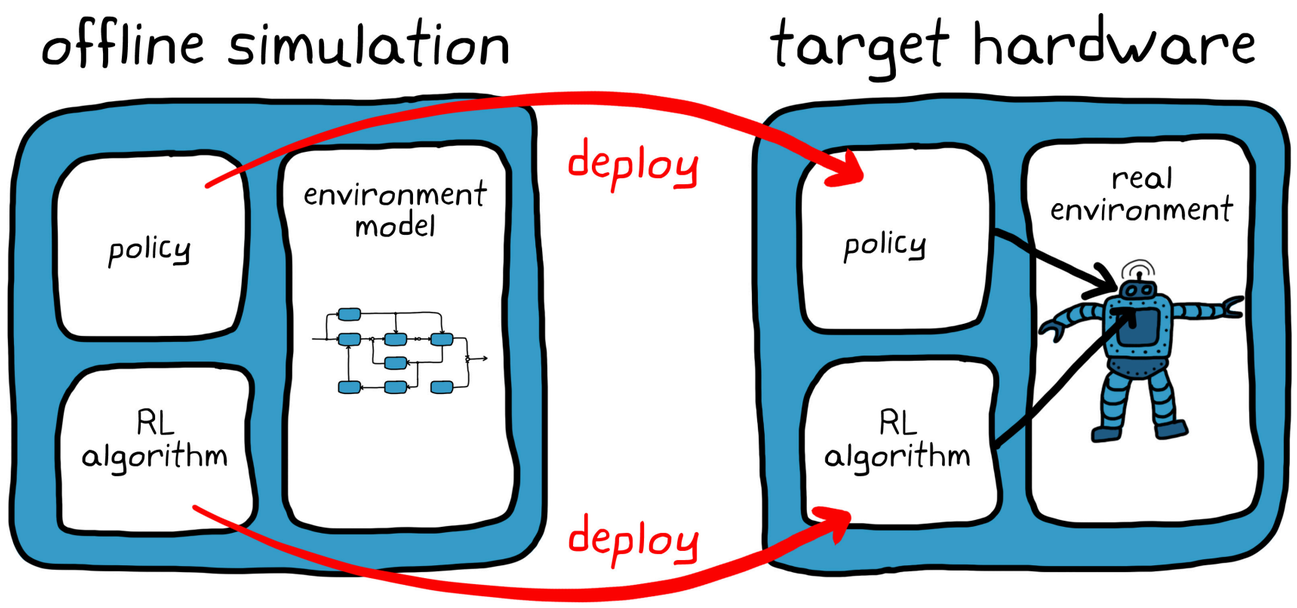
Deploying the Learning Algorithm



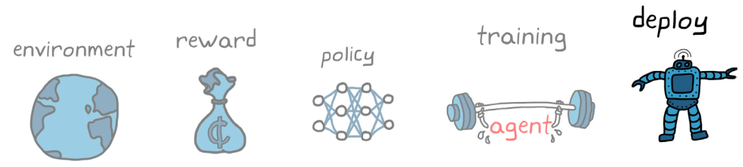
Even if the majority of learning is done offline with a simulated environment, it may be necessary to continue learning with the real physical hardware after deployment.

This is because some environments might be hard to model accurately, so a policy that is optimal for the model might not be optimal for the real environment. Another reason might be that the environment slowly changes over time and the agent must continue to learn occasionally so that it can adjust to those changes.

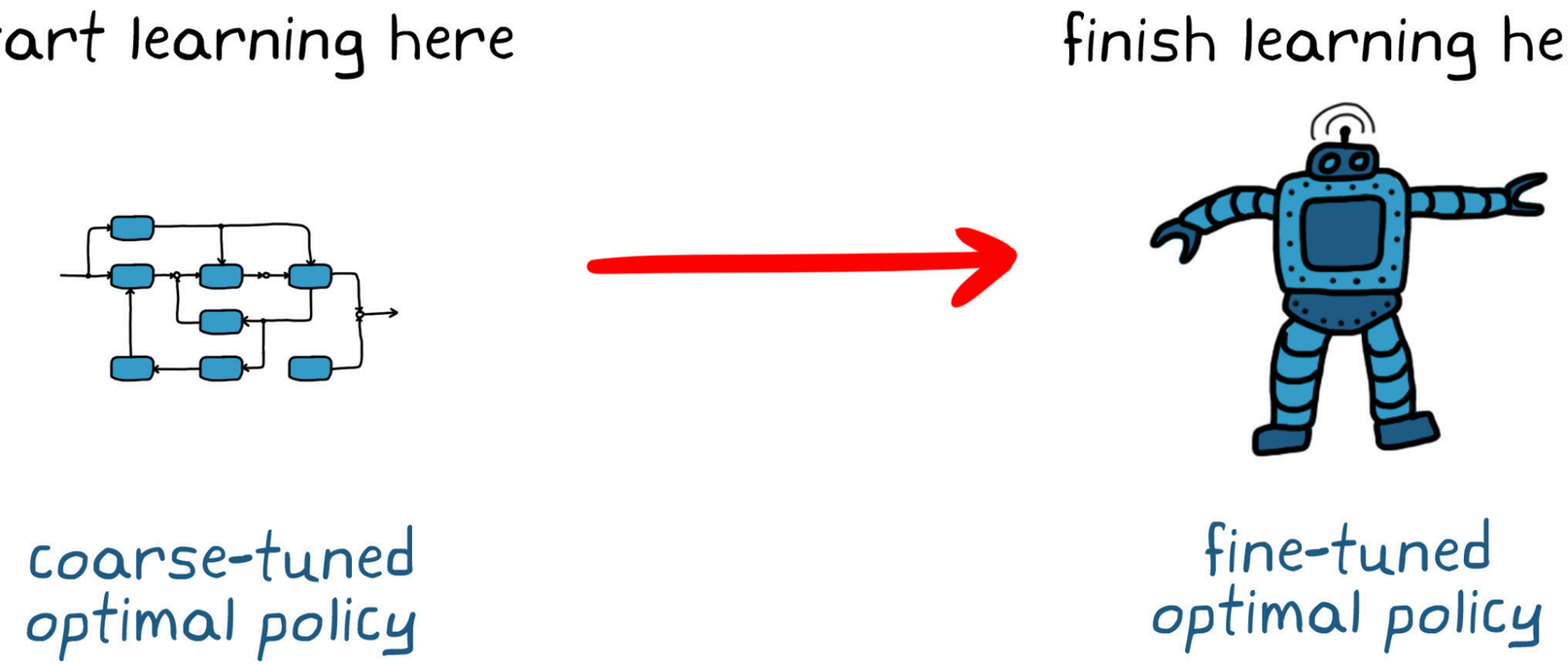
For these reasons, you deploy both the static policy and the learning algorithms to the target. With this setup, you have the option of executing the static policy (turn learning off) or continuing to update the policy (turn learning on).



The Complementary Relationship



There is a complementary relationship between learning with a simulated environment and learning with the real environment. With the simulation, you can safely and relatively quickly learn a sufficiently optimal policy—one that will keep the hardware safe and get close to the desired behavior even if it's not perfect. Then you can tweak the policy using the physical hardware and online learning to create something that is fine-tuned to the environment.

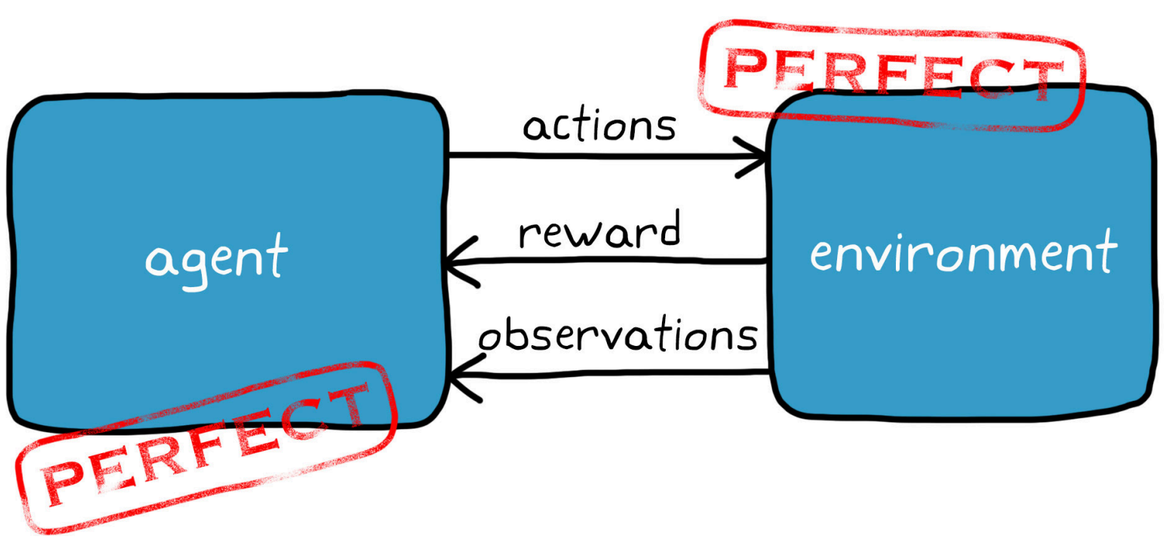


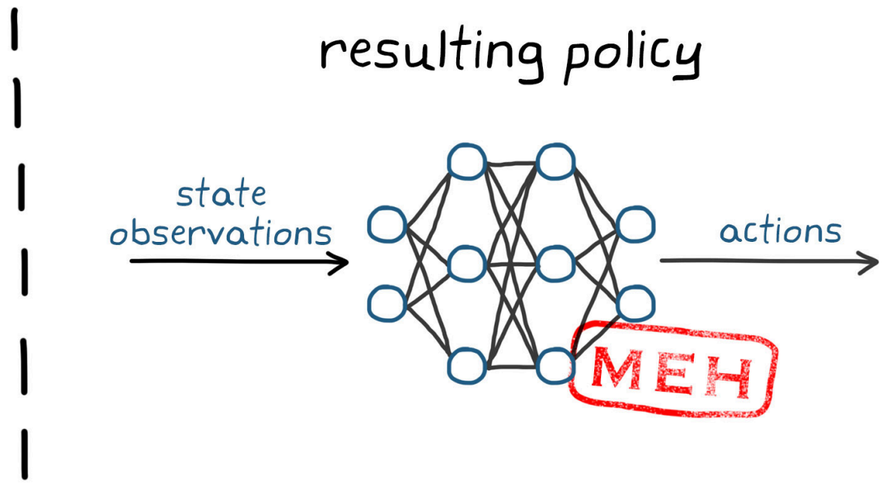
The Drawbacks of RL

At this point, you may think that you can set up an environment, place an RL agent in it, and then let the computer solve your problem while you go off and get a coffee. Unfortunately, even if you set up a perfect agent and a perfect environment and the learning algorithm converges on a solution, there are still drawbacks to this method.

These challenges come down to two main questions:

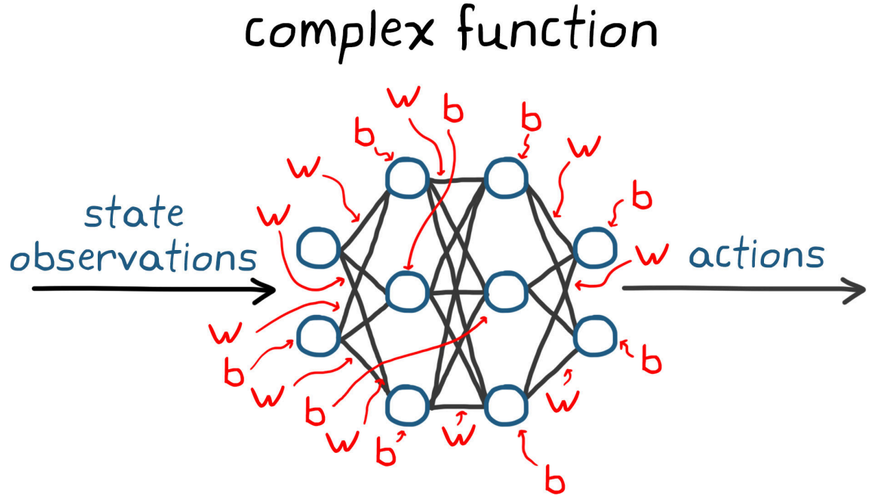
- How do you know the solution is going to work?
- Is there a way to manually adjust it if it's not quite perfect?

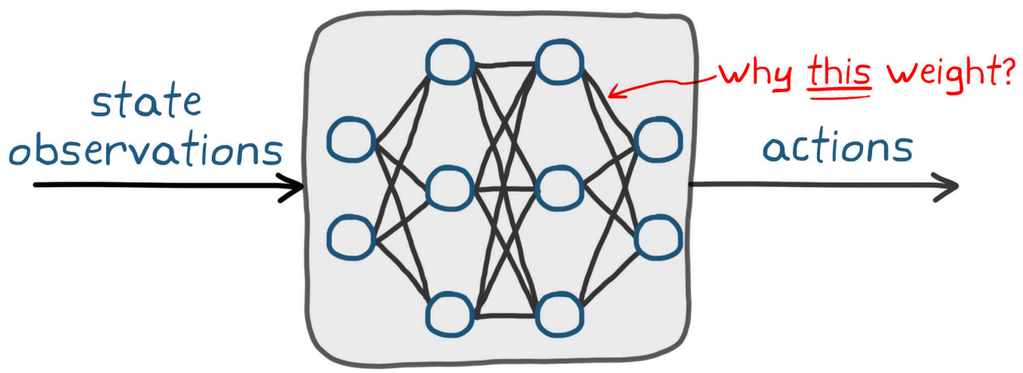




The Unexplainable Neural Network

Mathematically, a policy is 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 create a complex function that maps high-level observations to low-level actions.





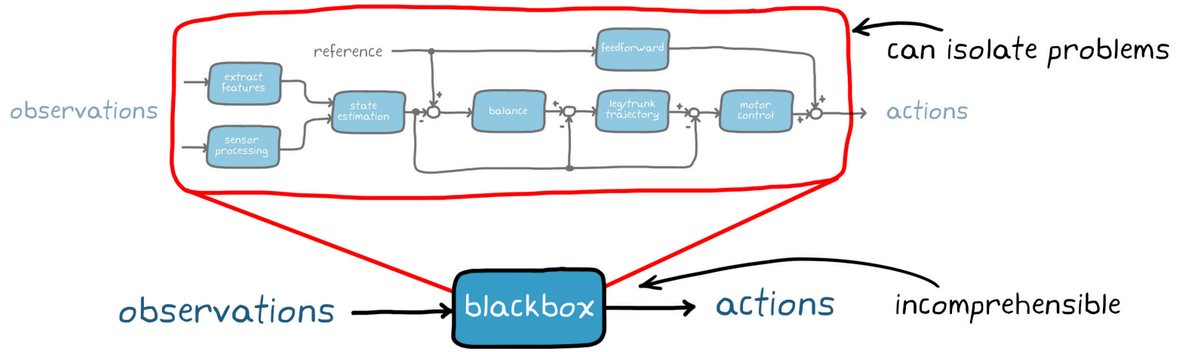
This function is a black box to the designer. You may have an intuitive sense of how this function operates and the hidden features that this network has identified, but you don't know the reason behind the value of any given weight or bias. So if the policy doesn't meet a specification or if the operating environment changes, you won't know how to adjust the policy to address that problem.

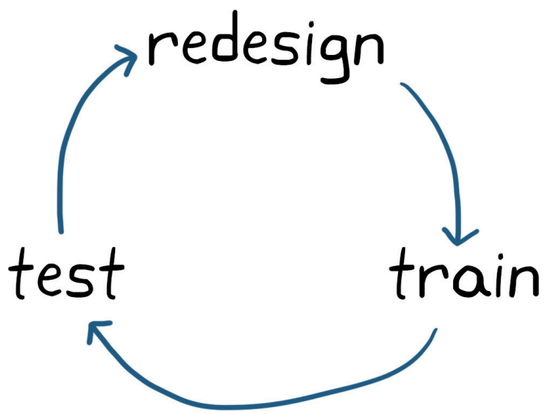
There is active research that is trying to push the concept of *explainable artificial intelligence*. This is the idea that you can set up your network so that it can be easily understood and audited by humans. At the moment, the majority of RL-generated policies are still categorized as a black box, which is an issue that needs to be addressed.

Pinpointing Problems

There is an issue where the very thing that has made solving the control problem easier—condensing the difficult logic down to a single black-box function—has made our final solution incomprehensible. 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 are 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 can isolate a controller and test and modify it to ensure it's performing under the specified conditions, and then bring it back into the larger system.



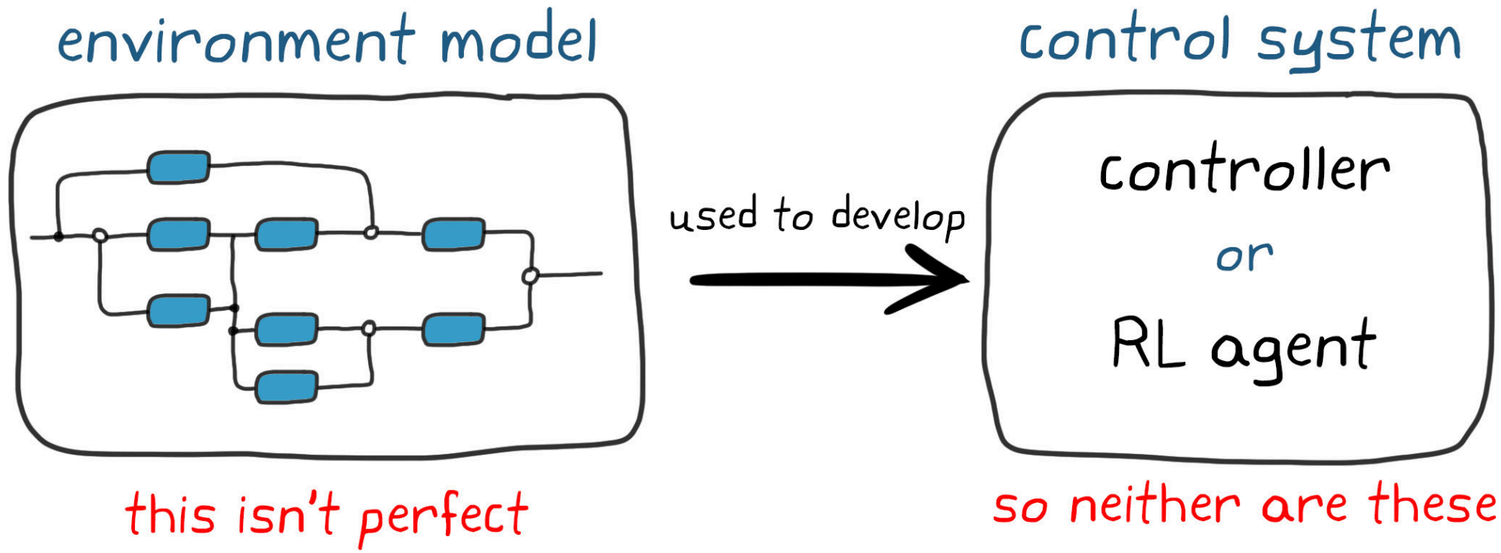


Isolating issues is difficult to do when the solution is a monolithic collection of neurons and weights and biases. So, if you end up with a policy that isn't quite right, rather than being able to fix the offending part of the policy, we have to redesign the agent or the environment model and then train it again. This cycle of redesigning, training, and testing can be time consuming.

The Larger Problem

There is a larger issue looming here that goes beyond the length of time it takes to train an agent, and it comes down to the accuracy of the environment model.

It is difficult to develop a sufficiently realistic model that takes into account all of the important system dynamics as well as disturbances and noise. At some point, it's not going to perfectly reflect reality, and so any control system you develop with that model is also not going to be perfect. This is why you still have to do physical tests rather than just verify everything with a model.



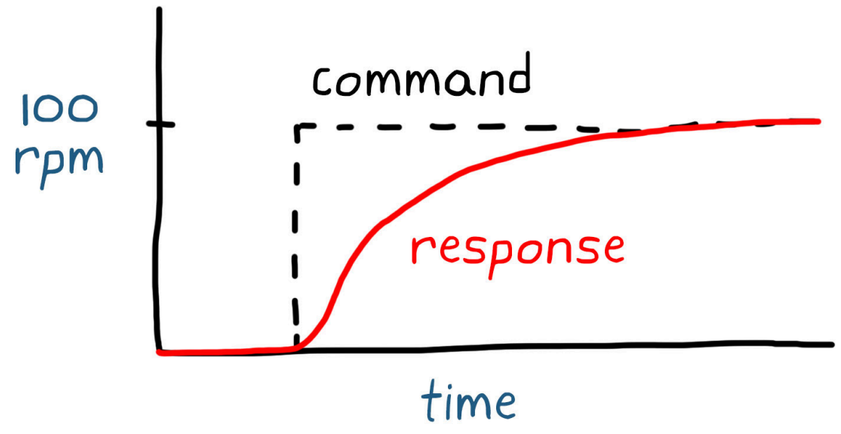
This is less of an issue if you use the model to design a traditional control system, since you can understand the functions and can tune the controllers. However, with a neural network policy, you don't have that luxury. As you can never build an absolutely realistic model, any agent you train with that model will be slightly wrong. The only option to fix it is to finish training the agent on the physical hardware, which can be challenging in its own right.

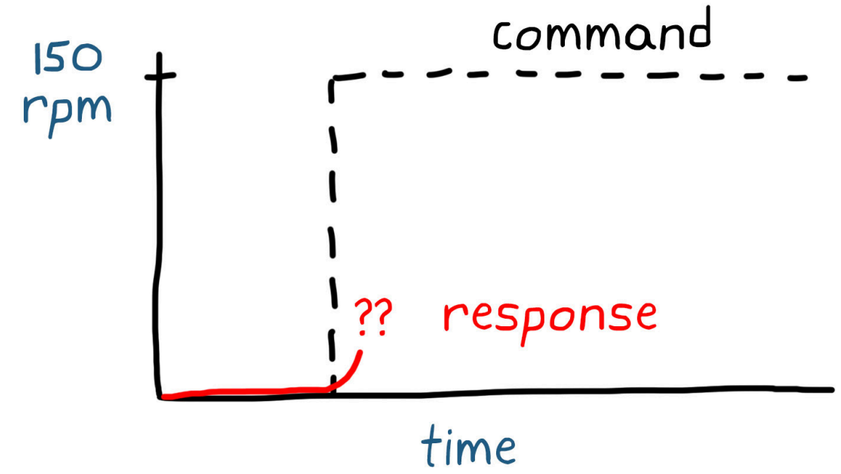
Verifying the Learned Policy

Verifying that a policy meets the specifications is also difficult with a neural network. For one reason, with a learned policy, it's hard to predict how the system will behave in one state based on its behavior in another. As an example, if you 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, you 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.

A slight change may cause a completely different set of neurons to activate and produce an undesired result. You won't know that unless you test it. Testing more conditions does reduce risk, but it doesn't guarantee a policy is 100% correct unless you can test every input combination.

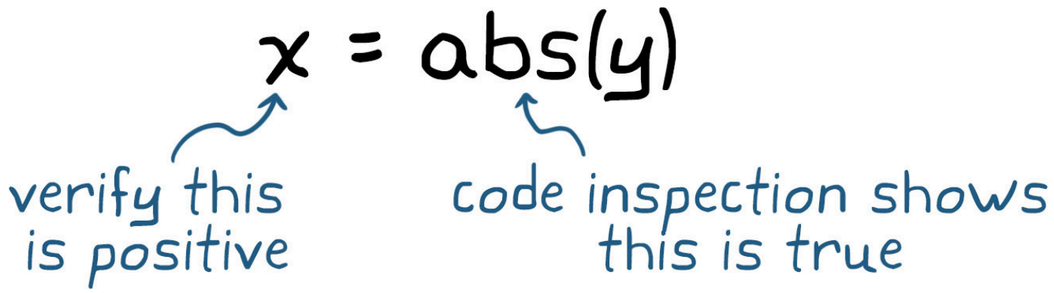
Having to run a few extra tests might not seem like a big deal, but you have to remember that one of the benefits of deep neural networks is that they can handle data from rich sensors, like images from a camera that have extremely large input spaces; think thousands of pixels that each can have a value between 0 and 255. Testing every combination in this scenario would be impossible.

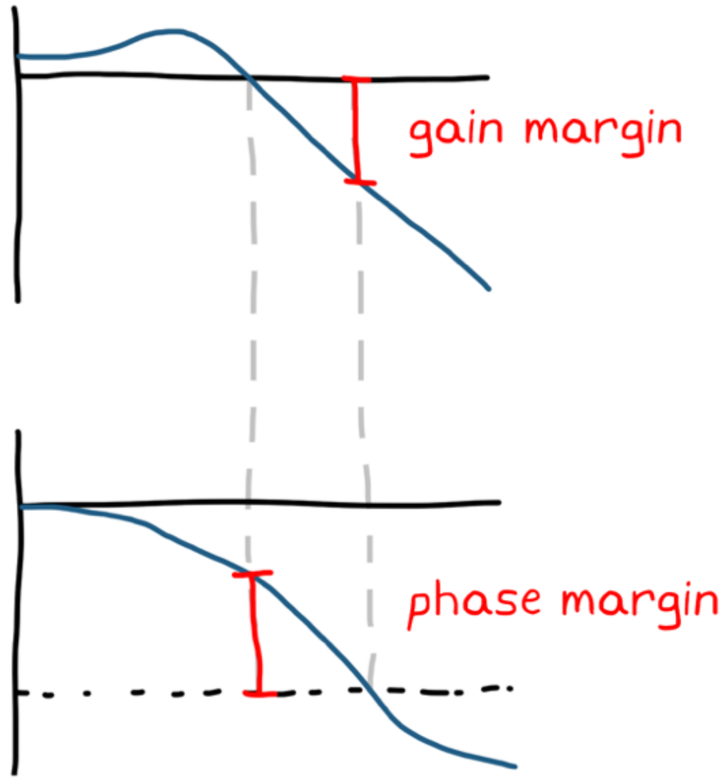




Formal Verification Methods

Learned neural networks also make formal verification difficult. These methods involve guaranteeing that some condition will be met by providing a formal proof rather than using a test. For example, you don't have to test to make sure a signal will always be nonnegative if the absolute value operation of that signal is performed in the software. You 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.



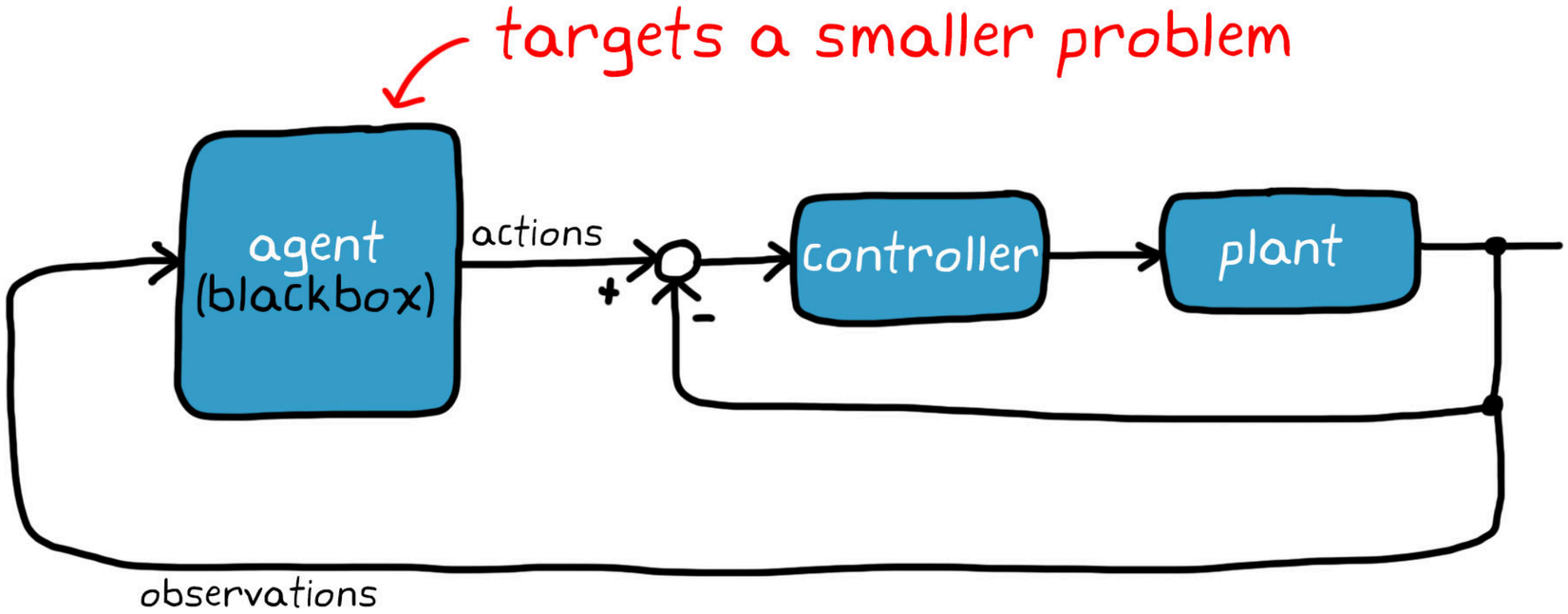


For neural networks, this type of formal verification is more difficult. As we've discussed, it's hard to inspect the code and make any guarantees about how it will behave. You also don't have methods to determine its robustness or its stability. It all comes back to the fact that you can't explain what the function is doing internally.

Shrinking the Problem

A good way to reduce the scale of these problems is to narrow the scope of the RL agent. 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.

A smaller policy is more focused so it's easier to understand what it's doing, its impact on the whole system is limited, and the training time is reduced. However, shrinking the policy doesn't solve your problem; it just decreases its complexity. You still don't know if it is robust to uncertainties, if there are stability guarantees, or if can you verify that the system will meet the specifications.



Working Around These Issues

Even though you can't quantify robustness, stability, and safety, you can address those issues with workarounds in the design.

For robustness and stability, you can train the agent in an environment where the important environment parameters are adjusted each time the simulation is run.

For example, if you choose a different max torque value for the walking robot at the start of each episode, the policy will eventually converge to something that is robust to manufacturing tolerances. Tweaking all of the important parameters like this will help you end up with an overall robust design. You may not be able to claim a specific gain or phase margin, but you will have more confidence that the result can handle a wider range within the operational state space.

episode	torque	length	delay	reference	...
1	2 Nm	1 cm	10 ms	step	.
2	2.5 Nm	1.3 cm	8 ms	ramp	.
3	2.1 Nm	1.7 cm	14 ms	impulse	.
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

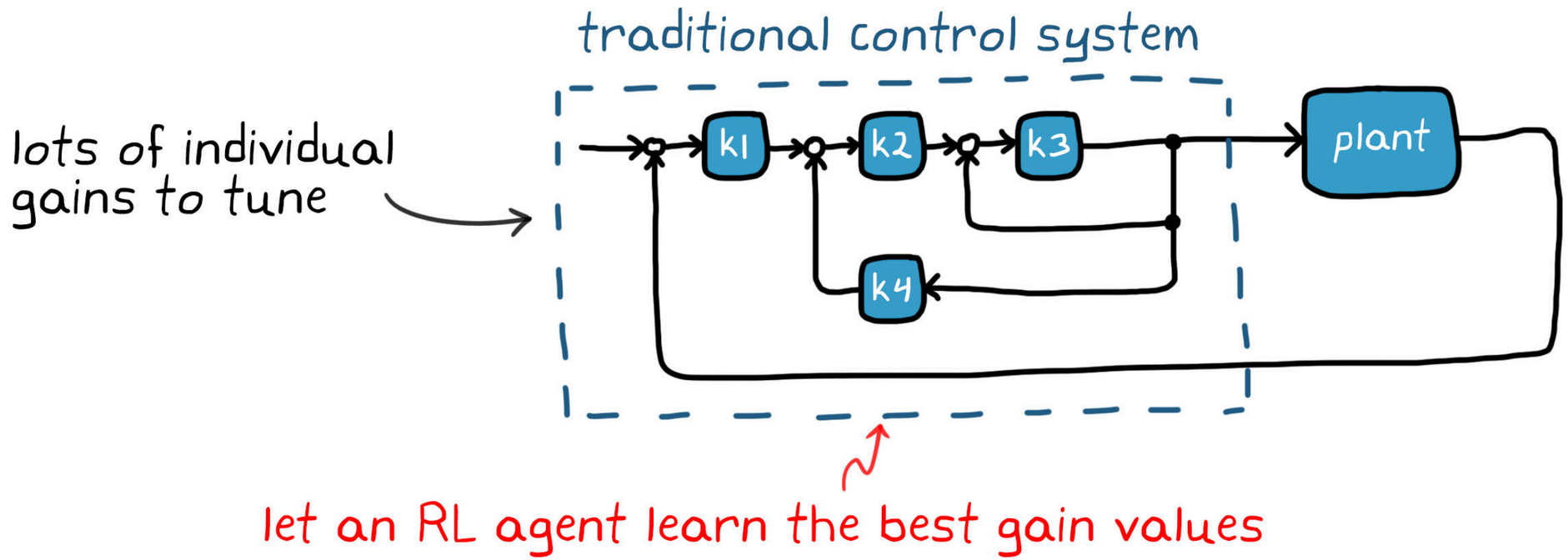
```
% software monitor
if abs(body_angle) > 45;    % monitor for falling
    mode = "safe";          % set safe mode
    extend_arms();          % prepare for impact
end
```

You can also increase safety by determining situations that you want the system to avoid no matter what, and build software outside of the policy that monitors for that situation. If that monitor is triggered, you can 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.

Solving a Different Problem

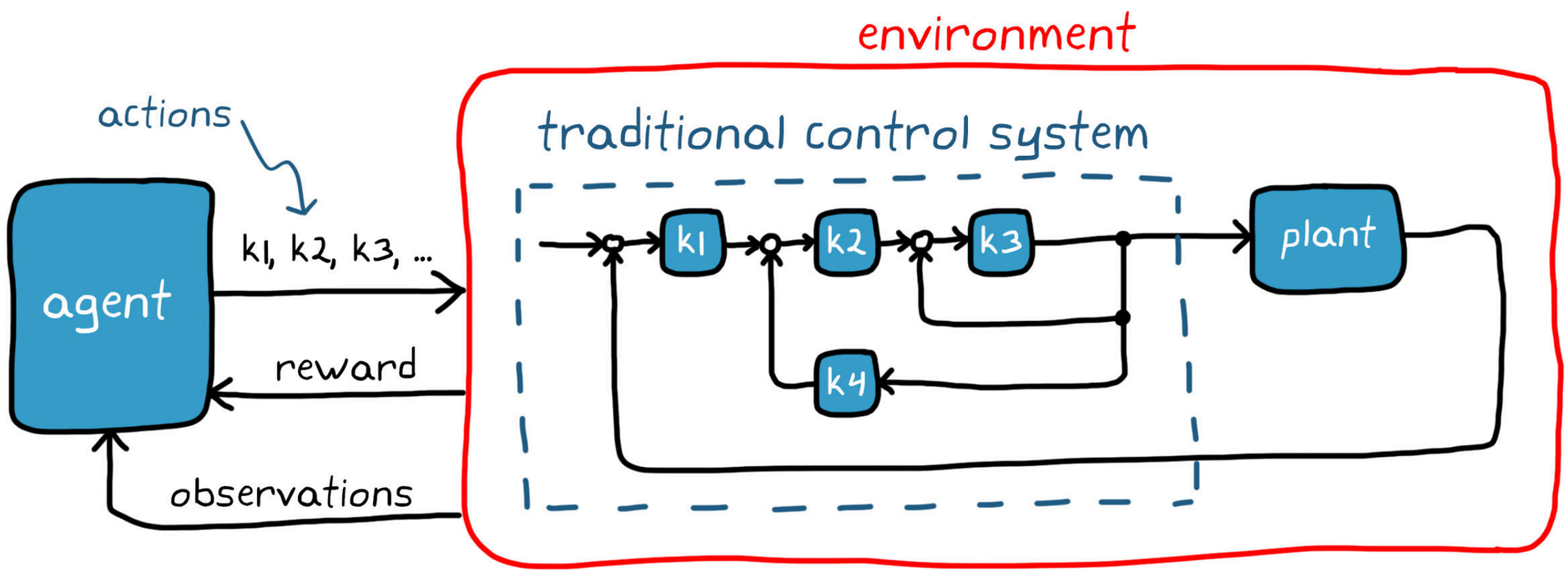
Workarounds are nice, but you can fix the issues directly by solving a different problem altogether. You can use reinforcement learning as a tool to optimize the controller gains in a traditionally architected control system. Imagine designing an architecture with dozens of nested loops and controllers, each with several gains. You can end up with a situation where you have a hundred 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.



RL Complementing Traditional Methods

Imagine an environment comprising a control system and the plant. The reward would be how well the system performs and how much effort it takes to get that performance, and the actions would be the gains for the system. After each episode, the learning algorithm would tweak the neural network in a way that the gains move in the direction that increases reward (i.e., it improves performance and lowers effort).

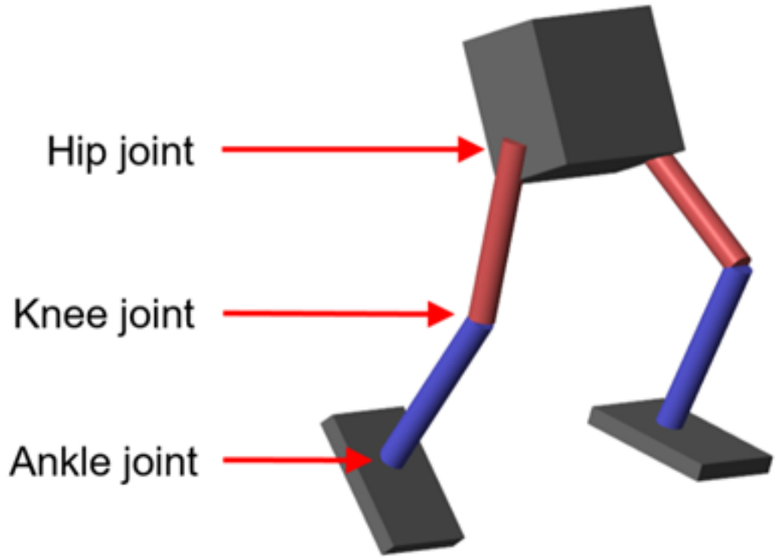
You get the best of both worlds with this method. You don't have to deploy any neural networks, verify them, or worry about having to change them; you just need to code the final static gain values into the controller. This way, you still have a traditionally architected system, one that can be verified and manually adjusted on the hardware, but you populated it with gain values that were optimally selected using reinforcement learning.



The Future of Reinforcement Learning

Reinforcement learning is a powerful tool for solving hard problems. There are some challenges regarding understanding the solution and verifying that it will work, but as we covered, you have a few ways right now to work around those challenges. While reinforcement learning is nowhere near its full potential, it may not be too long before it becomes the design method of choice for all complex control systems.

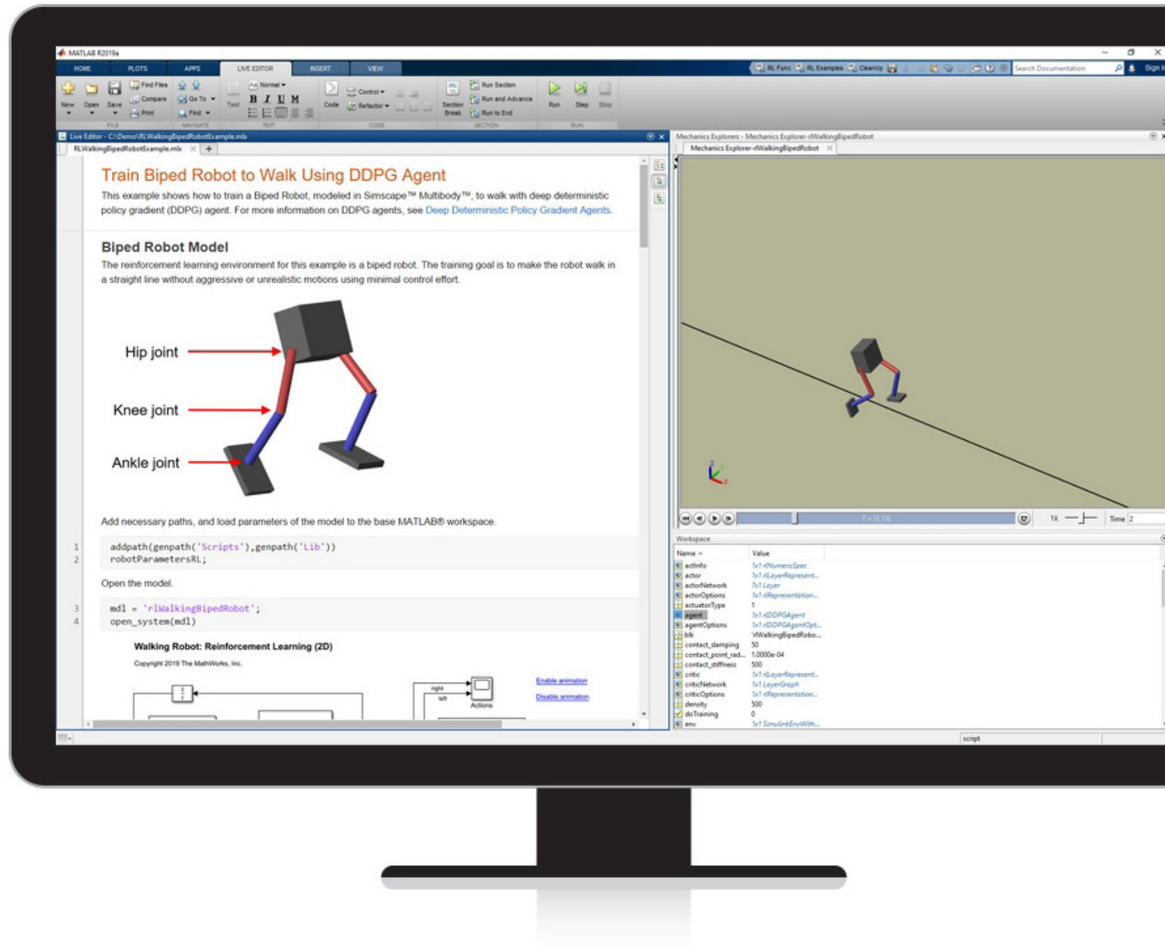
Learning algorithms, reinforcement learning design tools such as MATLAB and Reinforcement Learning Toolbox™, and verification methods are advancing all the time.



Reinforcement Learning with MATLAB

Reinforcement Learning Toolbox provides functions and blocks for training policies using reinforcement learning algorithms. You can use these policies to implement controllers and decision-making algorithms for complex systems such as robots and autonomous systems.

The toolbox lets you implement policies using deep neural networks, polynomials, or look-up tables. You can then train policies by enabling them to interact with environments represented by MATLAB or Simulink models.



Learn More

Train a Reinforcement Learning Agent in Basic Grid World - Documentation

Train an Actor-Critic Agent to Balance Cart-Pole System - Documentation

Train a Biped Robot to Walk Using DDPG Agent - Documentation

Getting Started with Reinforcement Learning - Code examples

Reinforcement Learning Tech Talks - Video Series