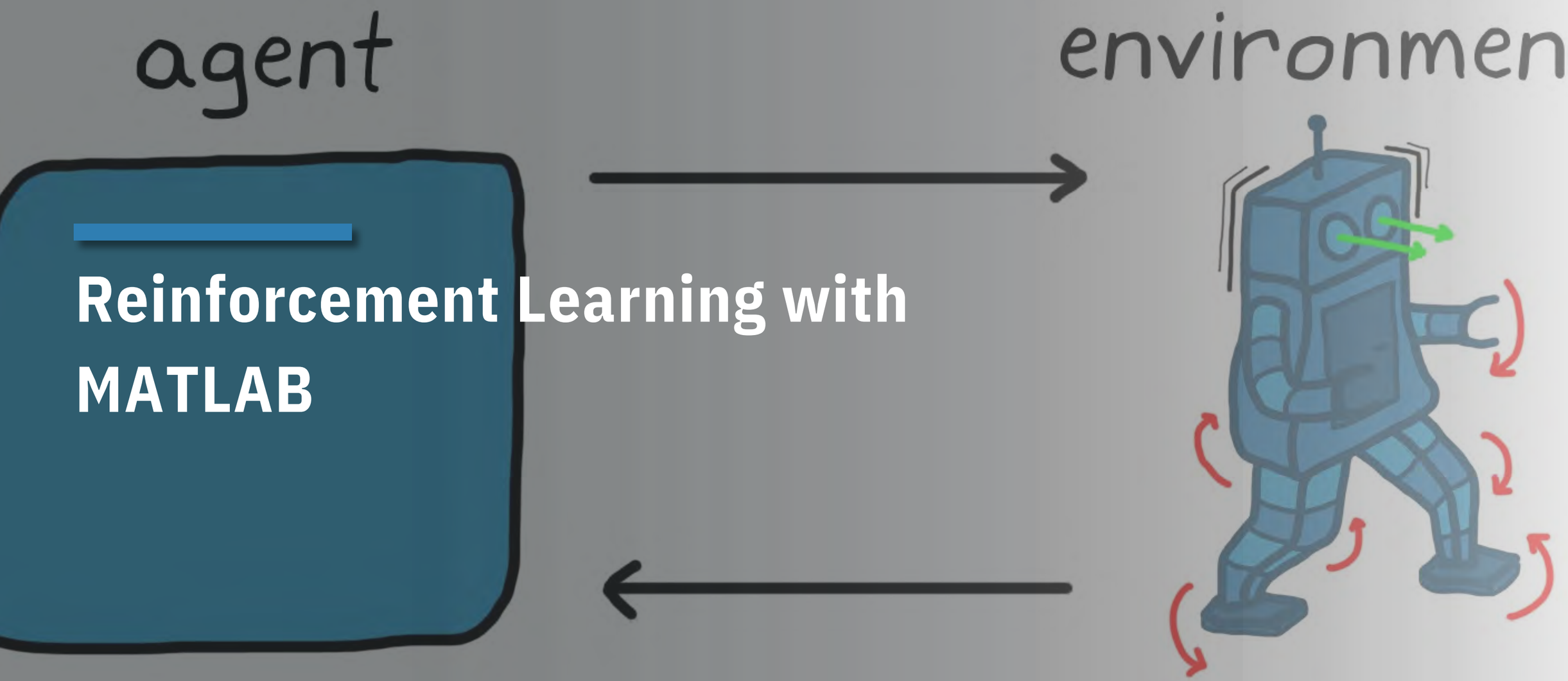
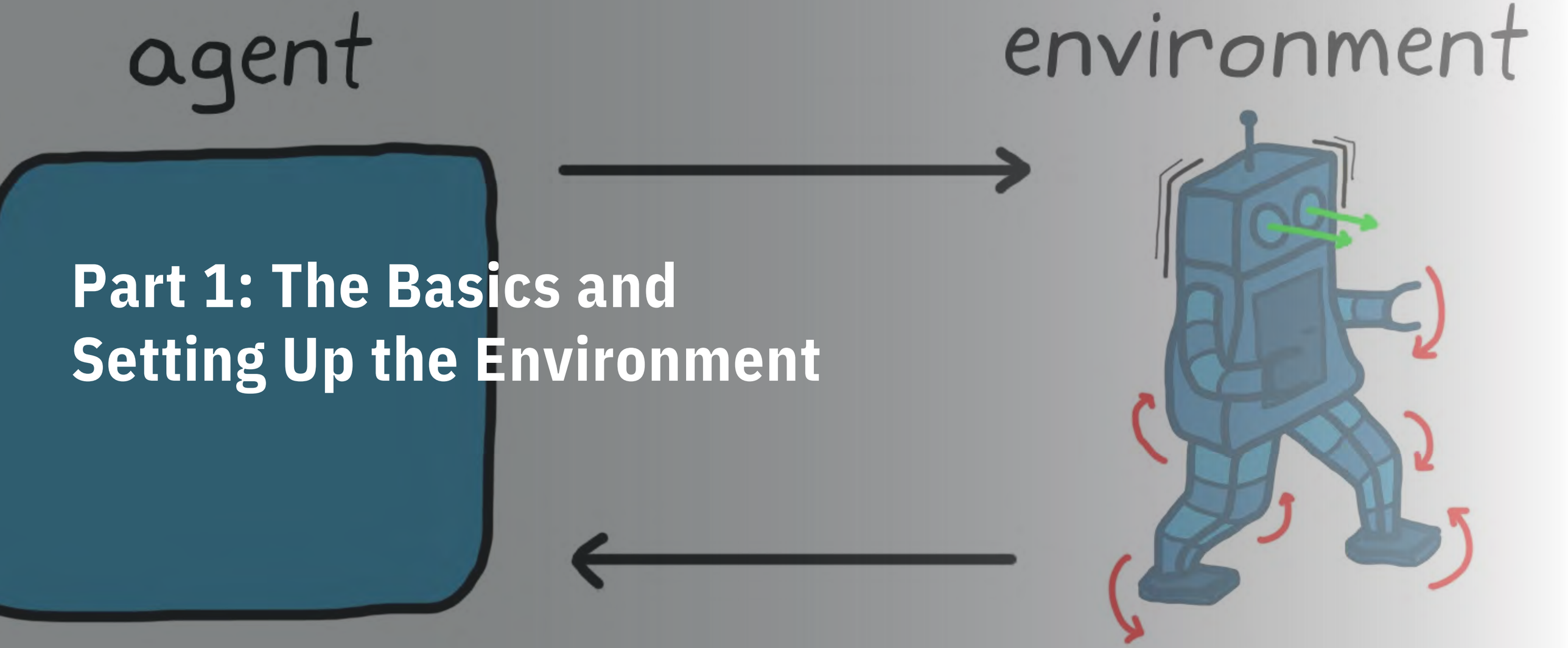


Reinforcement Learning with MATLAB

TECHSOURCE MathWorks®

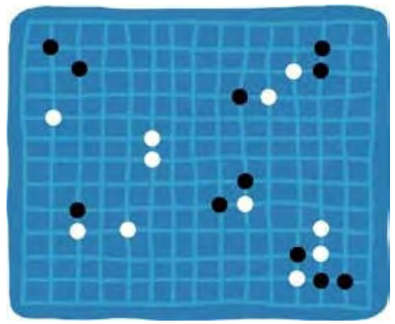
Table of Contents

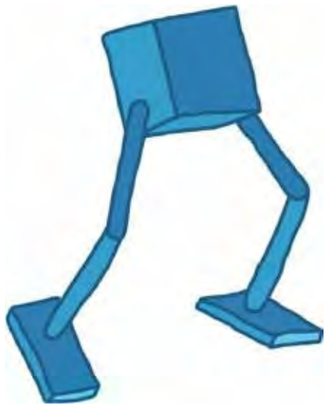


Part 1: The Basics and Setting Up the Environment

What Is Reinforcement Learning?

> *Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them.*
>
> —Sutton and Barto, *Reinforcement Learning: An Introduction*





Reinforcement learning (RL) has successfully trained computer programs to play games at a level higher than the world's best human players.

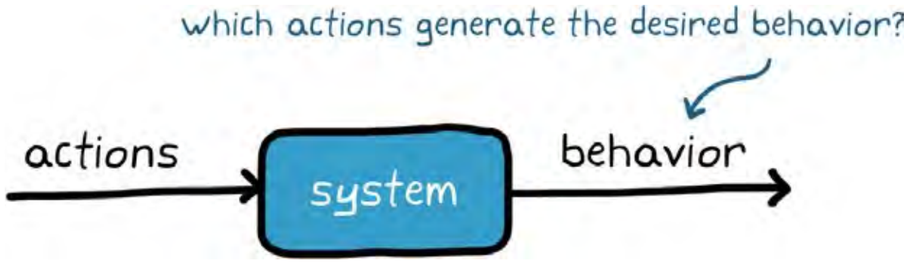
These programs find the best action to take in games with large state and action spaces, imperfect world information, and uncertainty around how short-term actions pay off in the long run.

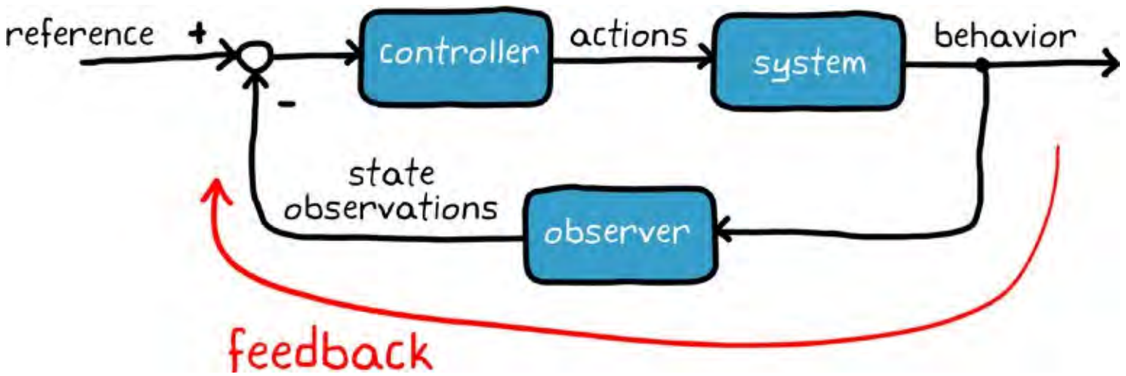
Engineers face the same types of challenges when designing controllers for real systems. Can reinforcement learning also help solve complex control problems like making a robot walk or driving an autonomous car?

This ebook answers that question by explaining what RL is in the context of traditional control problems and helps you understand how to set up and solve the RL problem.

The Goal of Control

Broadly speaking, the goal of a control system is to determine the correct inputs (actions) into a system that will generate the desired system behavior.





With feedback control systems, the controller uses state observations to improve performance and correct for random disturbances and errors. Engineers use that feedback, along with a model of the plant and environment, to design the controller to meet the system requirements.

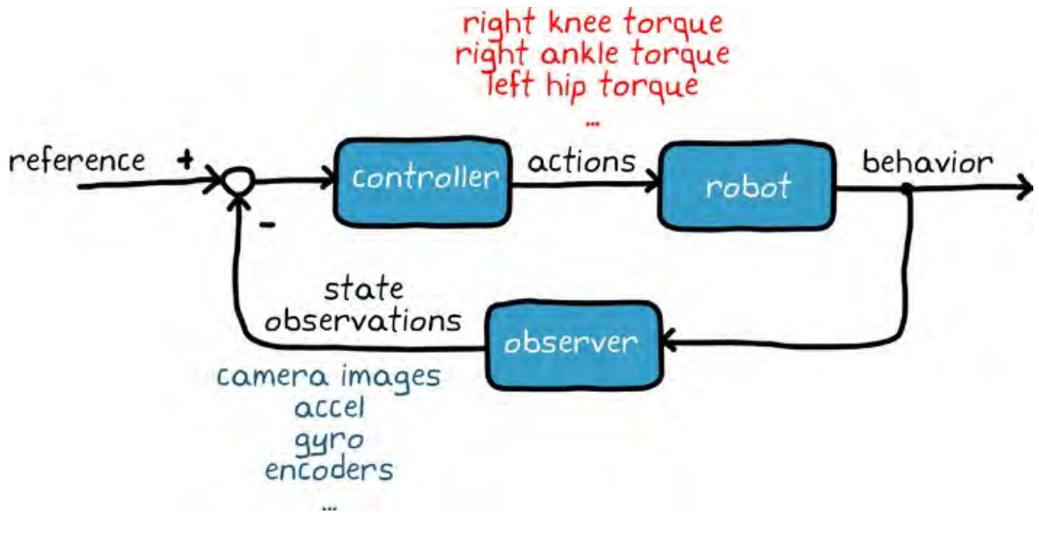
This concept is simple to put into words, but it can quickly become difficult to achieve when the system is hard to model, is highly nonlinear, or has large state and action spaces.

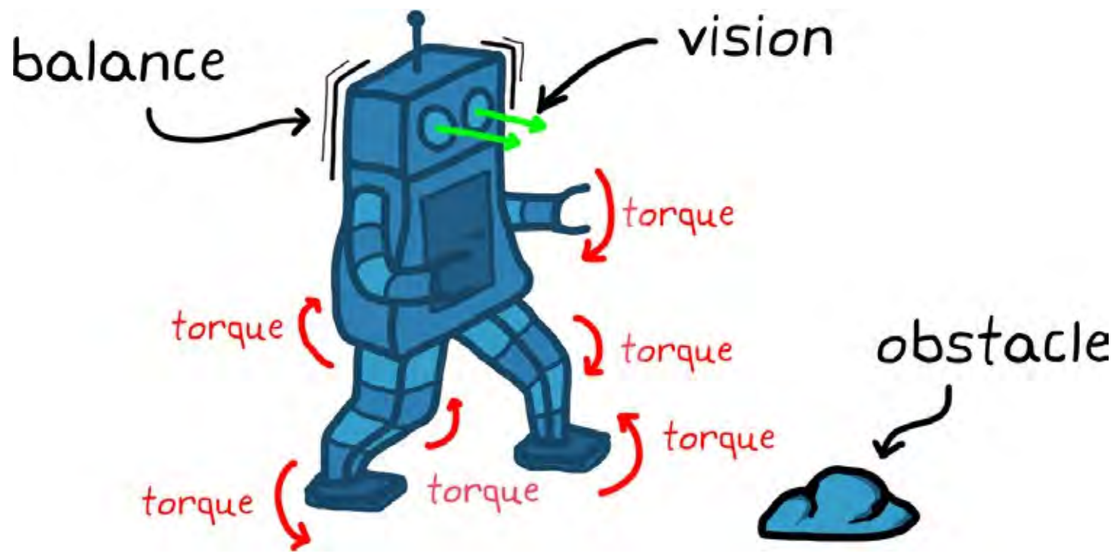
The Control Problem

To understand how complexity complicates a control design problem, imagine developing a control system for a walking robot.

To control the robot (i.e., the system), you command potentially dozens of motors that operate each of the joints in the arms and legs.

Each command is an action you can take. The state observations come from multiple sources, including a camera vision sensor, accelerometers, gyros, and encoders for each of the motors.





The controller has to satisfy multiple requirements:

- Determine the right combination of motor torques to get the robot walking and keep it balanced.
- Operate in an environment that has random obstacles that need to be avoided.
- Reject disturbances like wind gusts.

A control system design would need to handle these as well as any additional requirements like maintaining balance while walking down a steep hillside or across a patch of ice.

The Control Solution

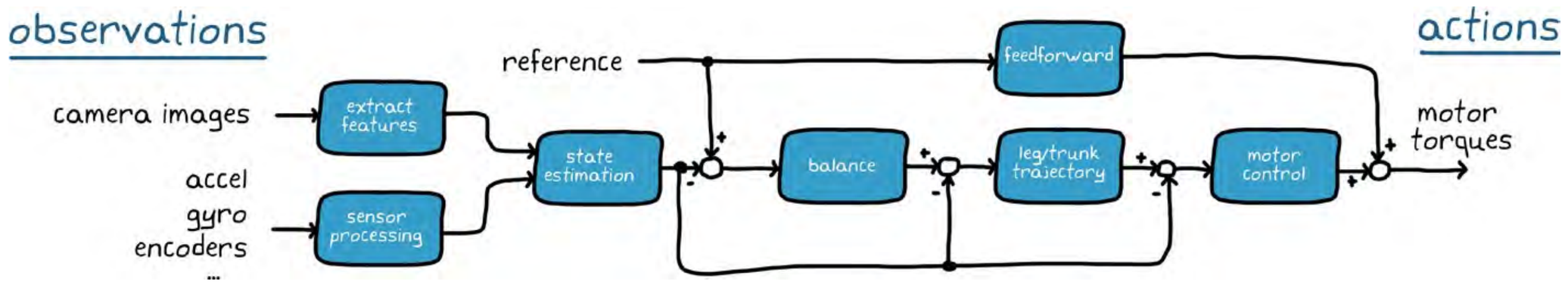
Typically, the best way to approach this problem is to break it up into smaller discrete sections that can be solved independently.

For example, you could build a process that extracts features from the camera images. These might be things like the location and type of obstacle, or the location of the robot in a global reference frame. Combine those states with the processed observations from the other sensors to complete the full state estimation.

The estimated state and the reference would feed into the controller, which would likely consist of multiple nested control loops. The outer loop would be responsible for managing high-level robot behavior (like maybe maintaining balance), and the inner loops manage low-level behaviors and individual actuators.

All solved? Not quite.

The loops interact with each other, which makes design and tuning challenging. Also, determining the best way to structure these loops and break up the problem is not simple.

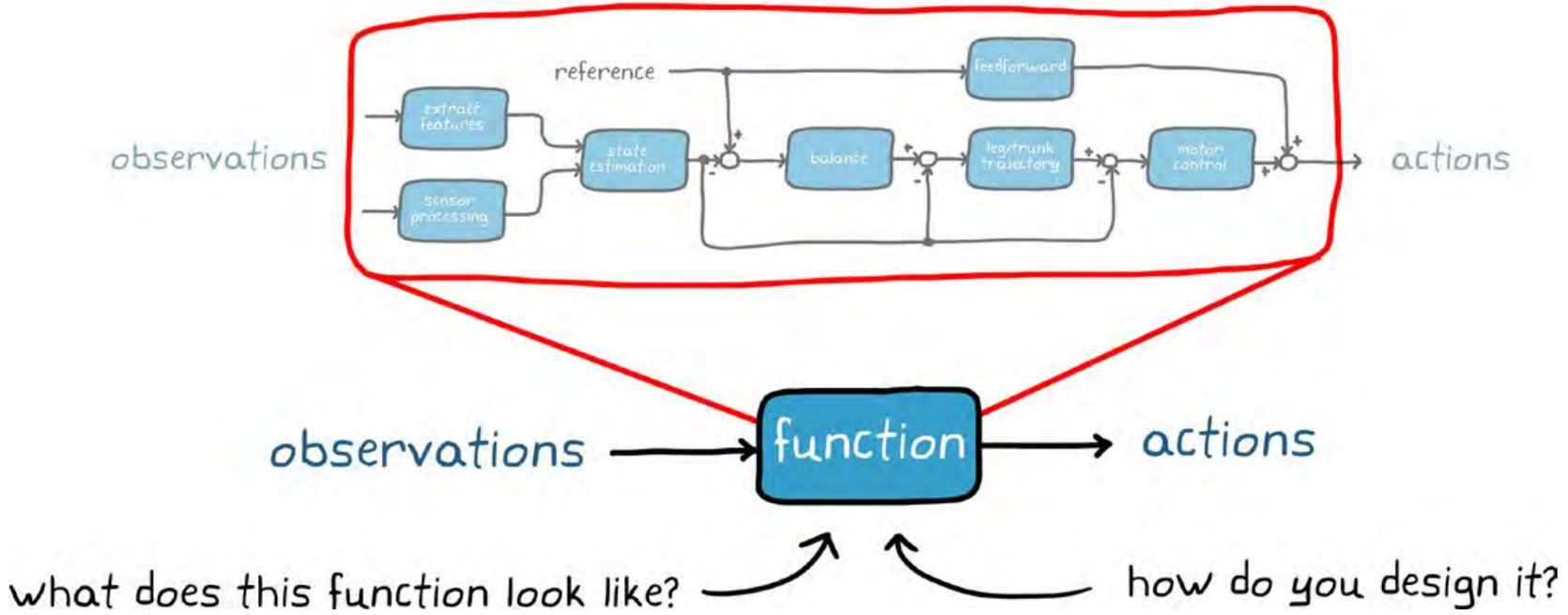


The Appeal of Reinforcement Learning

Instead of trying to design each of these components separately, imagine squeezing everything into a single function that takes in all of the observations and outputs the low-level actions directly.

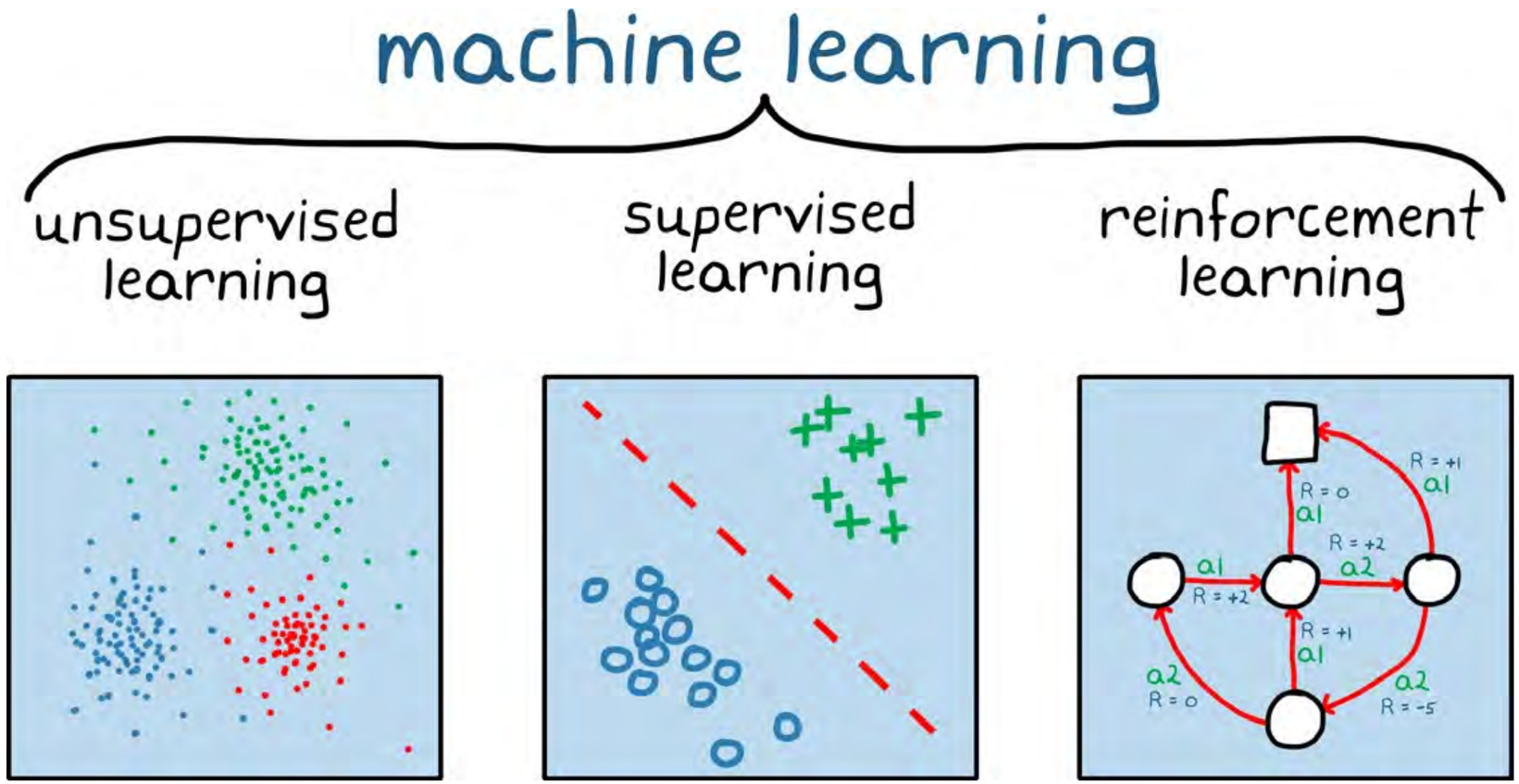
This certainly simplifies the block diagram, but what would this function look like and how would you design it?

It might seem like creating this single large function would be more difficult than building a control system with piecewise subcomponents; however, this is where reinforcement learning can help.



Reinforcement Learning: A Subset of Machine Learning

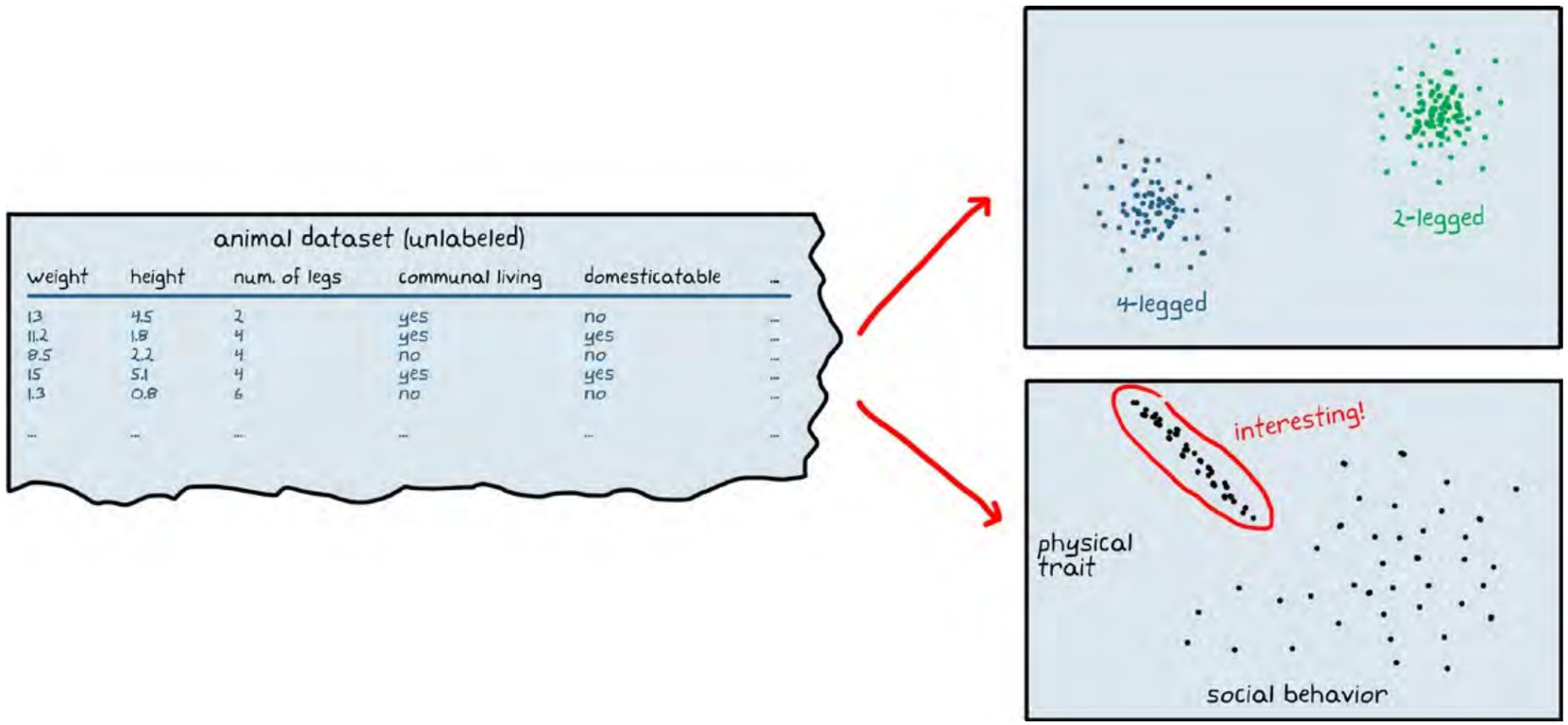
Reinforcement learning is one of three broad categories of machine learning. This ebook does not focus on unsupervised or supervised learning, but it is worth understanding how reinforcement learning differs from these two.



Machine Learning: Unsupervised Learning

Unsupervised learning is used to find patterns or hidden structures in datasets that have not been categorized or labeled.

For example, say you have information on the physical attributes and social tendencies of 100,000 animals. You could use unsupervised learning to group the animals or cluster them into similar features. These groups could be based on number of legs, or based on patterns that might not be as obvious, such as correlations between physical traits and social behavior that you didn't know about ahead of time.

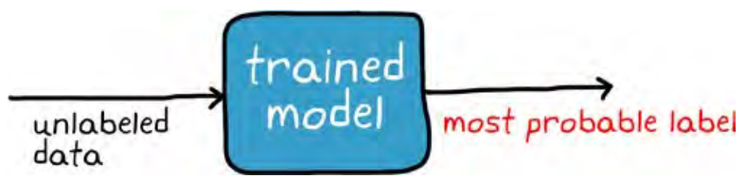


Machine Learning: Supervised Learning

Using supervised learning, you train the computer to apply a label to a given input. For example, if one of the columns of your dataset of animal features is the species, you can treat species as the label and the rest of the data as inputs into a mathematical model.

You could use supervised learning to train the model to correctly label each set of animal features in your dataset. The model guesses the species, and then the machine learning algorithm systematically tweaks the model.

With enough training data to get a reliable model, you could then input the features for a new, unlabeled animal, and the trained model would apply the most probable species label to it.

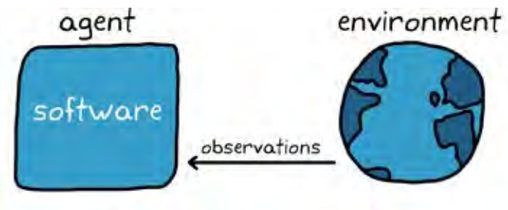


Machine Learning: Reinforcement Learning

Reinforcement learning is a different beast altogether. Unlike the other two learning frameworks, which operate using a static dataset, RL works with data from a dynamic environment. And the goal is not to cluster data or label data, but to find the best sequence of actions that will generate the optimal outcome. The way reinforcement learning solves this problem is by allowing a piece of software called an *agent* to explore, interact with, and learn from the environment.

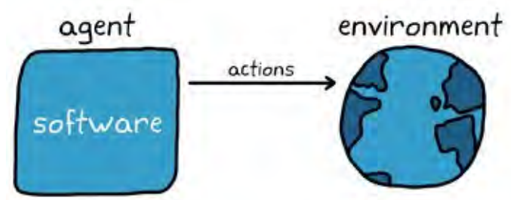
1

The agent is able to observe the current state of the environment.



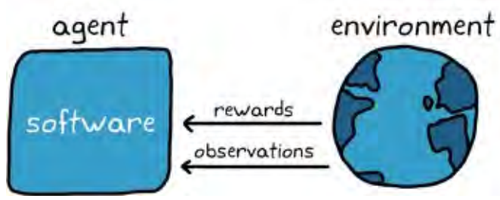
2

From the observed state, it decides which action to take.



3

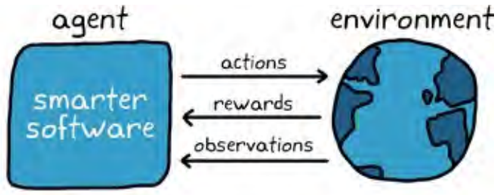
The environment changes state and produces a reward for that action. Both of which are received by the agent.



4

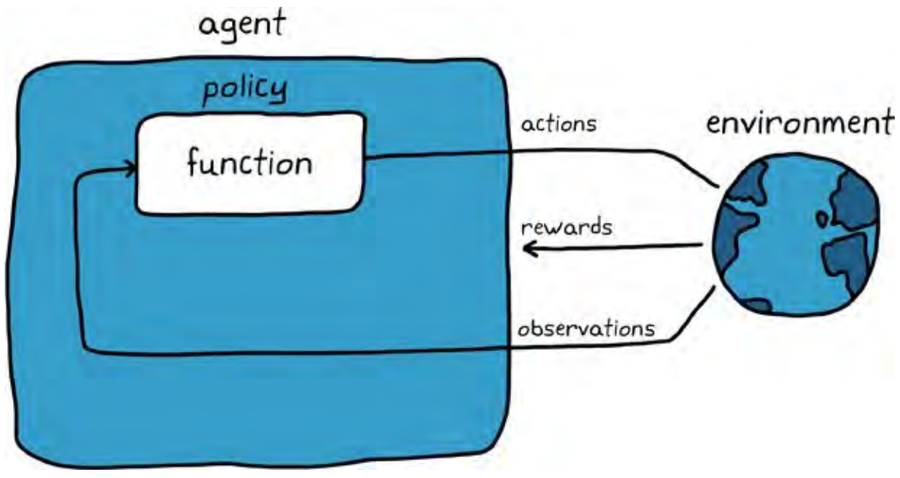
Using this new information, the agent can determine whether that action was good and should be repeated, or if it was bad and should be avoided.

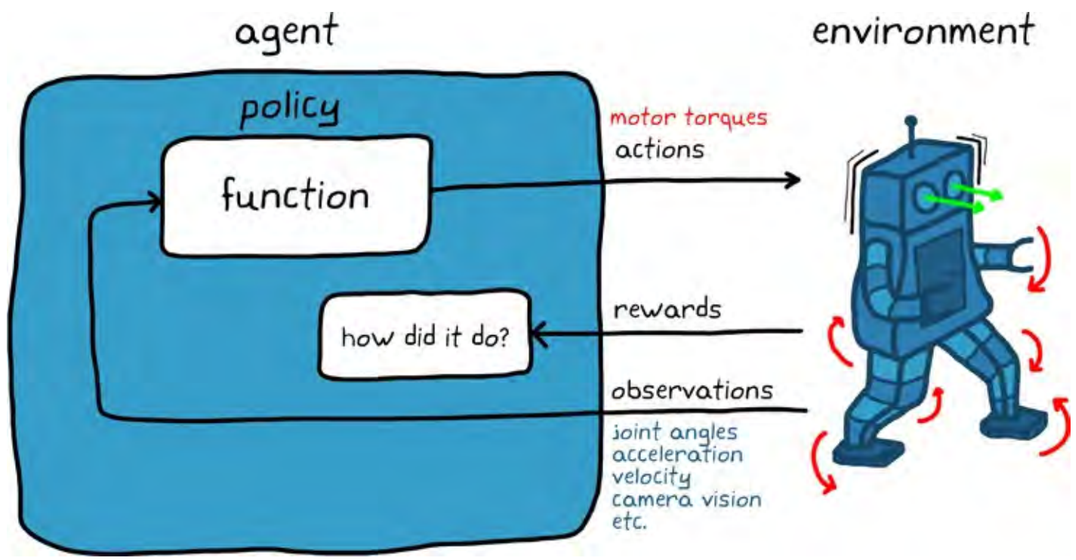
The observation-action-reward cycle continues until learning is complete.



Anatomy of Reinforcement Learning

Within the agent, there is a function that takes in state observations (the inputs) and maps them to actions (the outputs). This is the single function discussed earlier that will take the place of all of the individual subcomponents of your control system. In the RL nomenclature, this function is called the *policy*. Given a set of observations, the policy decides which action to take.





In the walking robot example, the observations would be the angle of every joint, the acceleration and angular velocity of the robot trunk, and the thousands of pixels from the vision sensor. The policy would take in all of these observations and output the motor commands that will move the robot's arms and legs.

The environment would then generate a reward telling the agent how well the very specific combination of actuator commands did. If the robot stays upright and continues walking, the reward will be higher than if the robot falls to the ground.

Learning the Optimal Policy

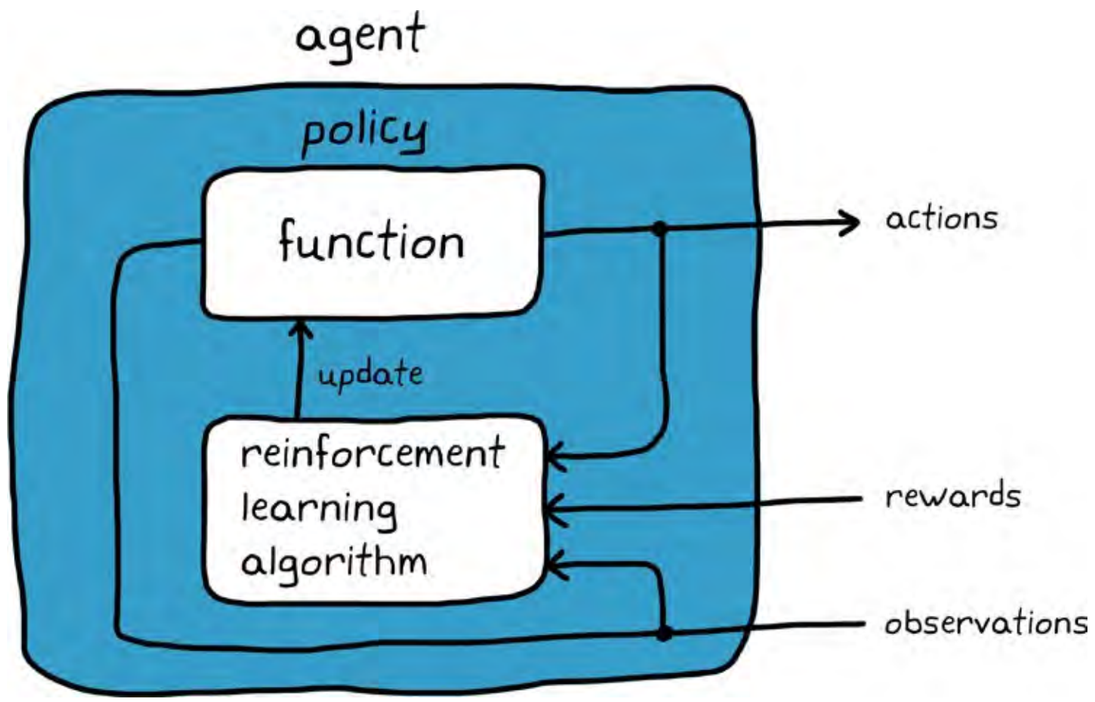
If you were able to design a perfect policy that would correctly command the right actuators for every observed state, then your job would be done.

Of course, that would be difficult to do in most situations. Even if you did find the perfect policy, the environment might change over time, so a static mapping would no longer be optimal.

This brings us to the reinforcement learning algorithm.

It changes the policy based on the actions taken, the observations from the environment, and the amount of reward collected.

The goal of the agent is to use reinforcement learning algorithms to learn the best policy as it interacts with the environment so that, given any state, it will always take the most optimal action—the one that will produce the most reward in the long run.



What Does It Mean to Learn?

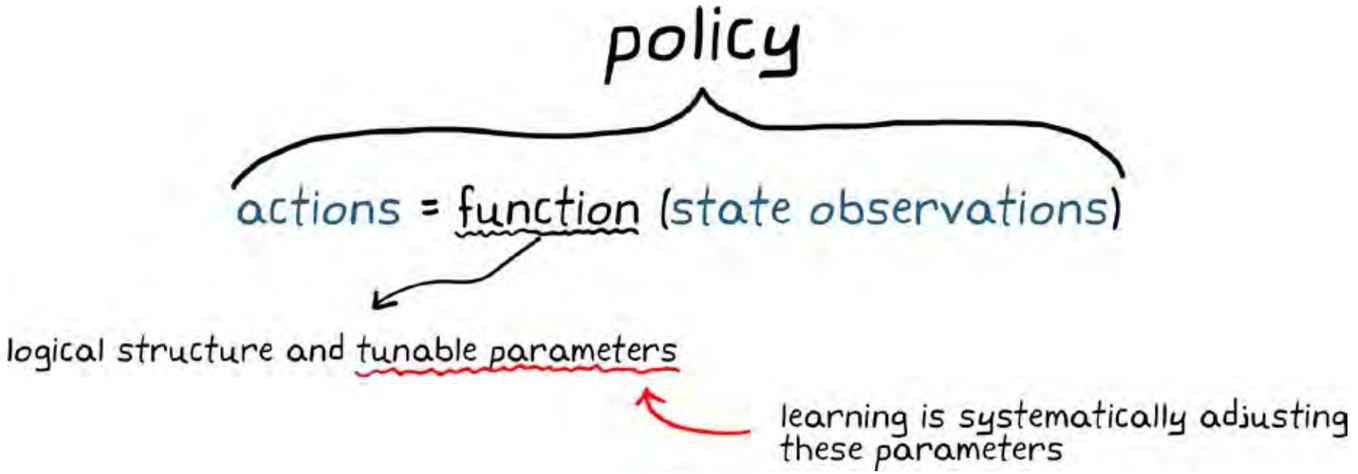
To understand what it means for a machine to learn, think about what a policy actually is: a function made up of logic and tunable parameters.

Given a sufficient policy structure (logical structure), there is a set of parameters that will produce an optimal policy—a mapping of states to actions that produces the most long-term reward.

Learning is the term given to the process of systematically adjusting those parameters to converge on the optimal policy.

In this way, you can focus on setting up an adequate policy structure without manually tuning the function to get the right parameters.

You can let the computer learn the parameters on its own through a process that will be covered later on, but for now you can think of as fancy trial and error.



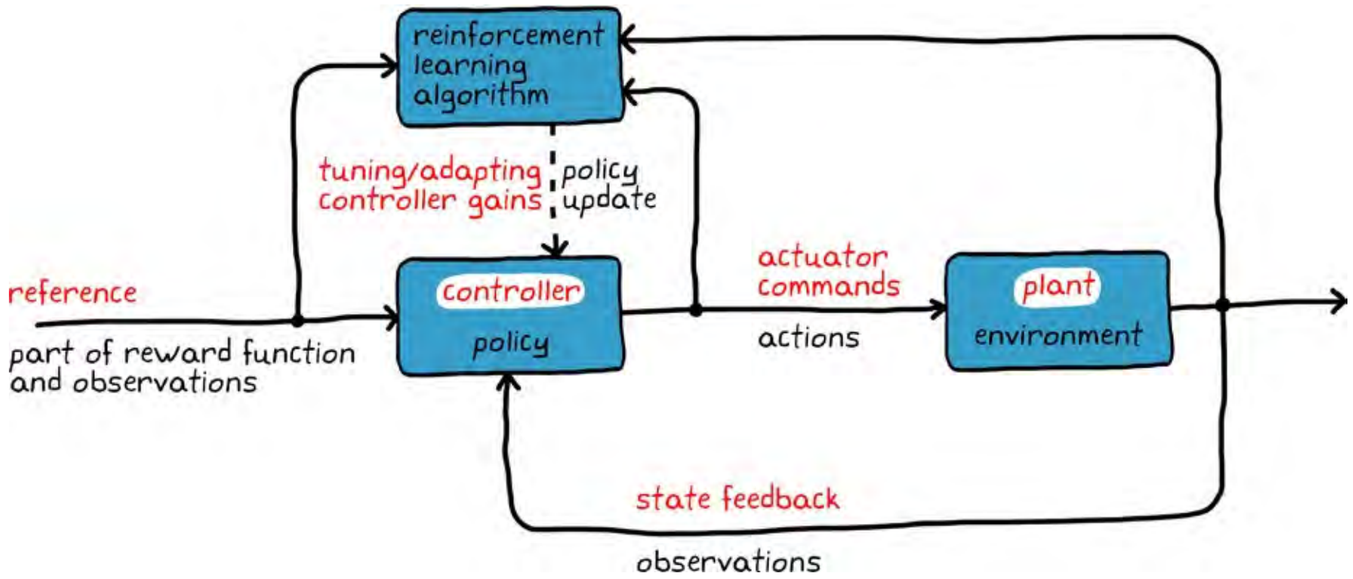
How Is Reinforcement Learning Similar to Traditional Controls?

The goal of reinforcement learning is similar to the control problem; it's just a different approach and uses different terms to represent the same concepts.

With both methods, you want to determine the correct inputs into a system that will generate the desired system behavior.

You are trying to figure out how to design the policy (or the controller) that maps the observed state of the environment (or the plant) to the best actions (the actuator commands).

The state feedback signal is the observations from the environment, and the reference signal is built into both the reward function and the environment observations.



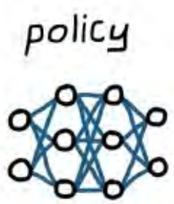
Reinforcement Learning Workflow Overview

In general, five different areas need to be addressed with reinforcement learning. This ebook focuses on the first area, setting up the environment. Other ebooks in this series will explore reward, policy, training, and deployment in more depth.

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.

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.

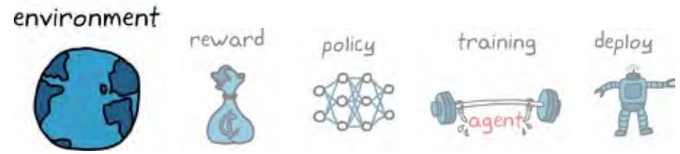
3 You need to choose a way to represent the policy. Consider how you want to structure the parameters and logic that make up the decision-making part of the agent.



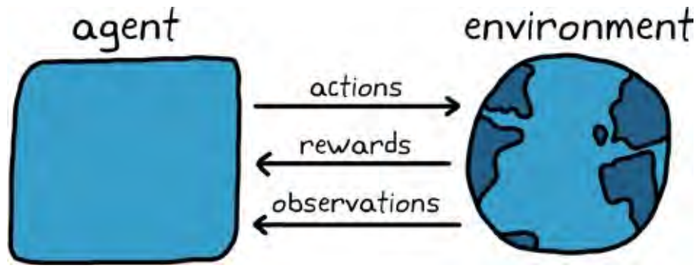
4 You need to choose an algorithm to train the agent that works to find the optimal policy parameters.

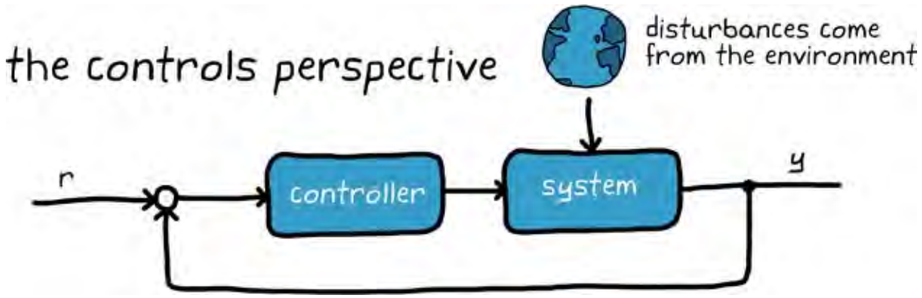
5 Finally, you need to exploit the policy by deploying it in the field and verifying the results.

Environment



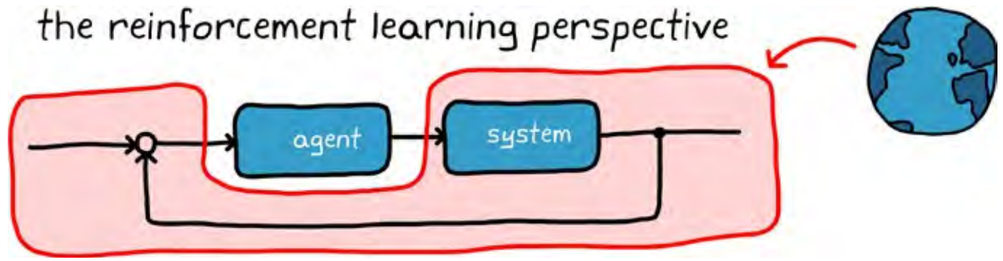
The environment is everything that exists outside of the agent. It is where the agent sends actions, and it is what generates rewards and observations.



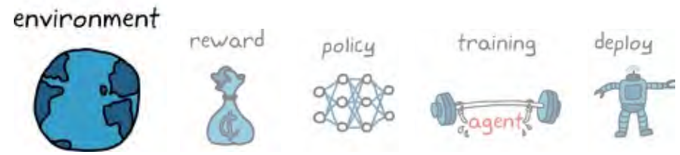


This definition can be confusing if you're coming from a controls perspective because you may tend to think of the environment as disturbances that impact the system you're trying to control.

However, in reinforcement learning nomenclature, the environment is everything but the agent. This includes the system dynamics. In this way, most of the system is actually part of the environment. The agent is just the bit of software that is generating the actions and updating the policy through learning.



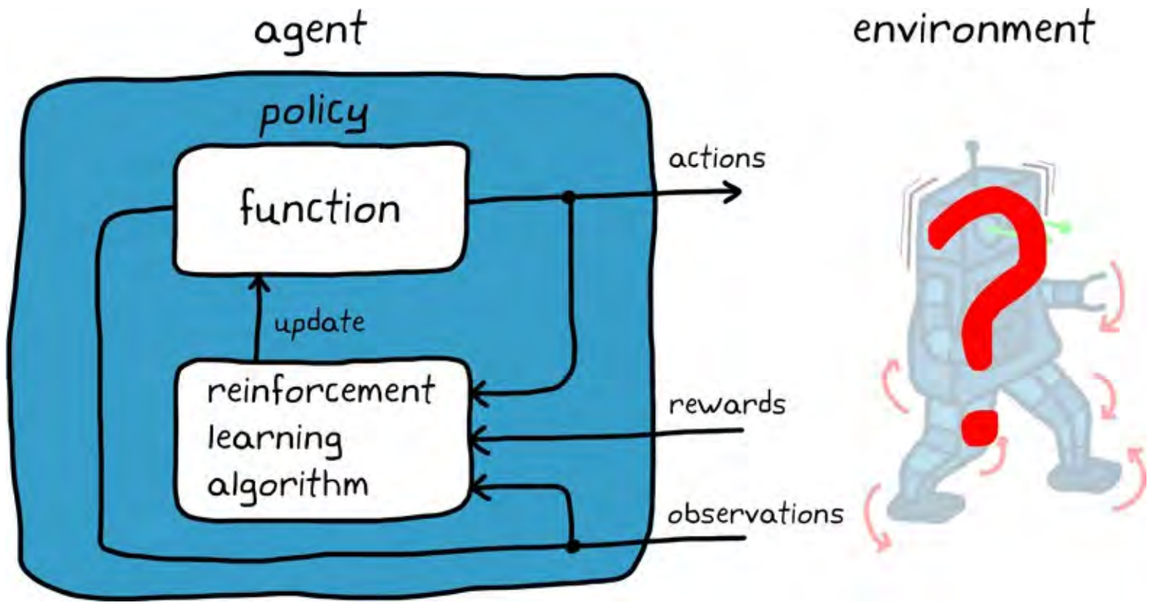
Model-Free Reinforcement Learning



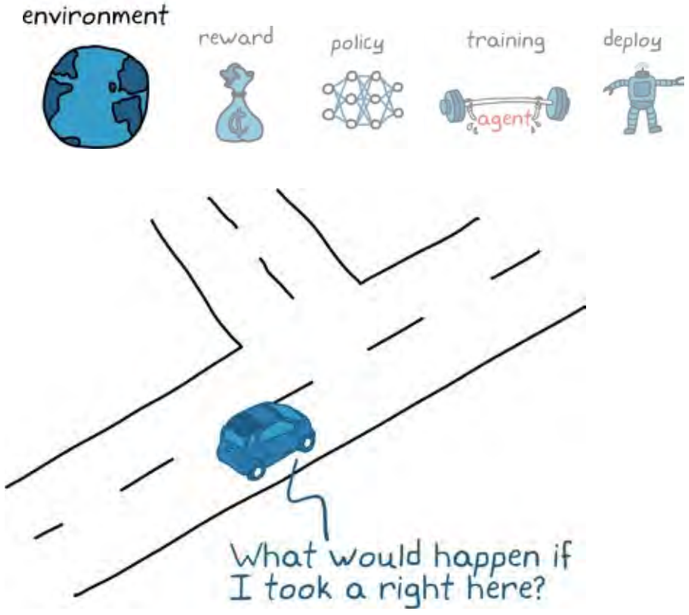
One reason reinforcement learning is so powerful is that the agent does not need to know anything about the environment. It can still learn how to interact with it. For example, the agent doesn't need to know the dynamics or kinematics of the walking robot. It will still figure out how to collect the most reward without knowing how the joints move or the lengths of the appendages.

This is called *model-free reinforcement learning*.

With model-free RL, you can put an RL-equipped agent into any system and the agent will be able to learn the optimal policy. (This assumes you've given the policy access to the observations, rewards, actions, and enough internal states.)



Model-Based Reinforcement Learning



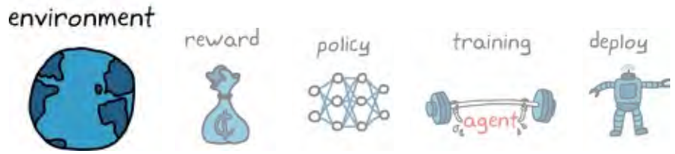
Here is the problem with model-free RL. If the agent has no understanding of the environment, then it must explore all areas of the state space to figure out how to collect the most reward.

This means the agent will need to spend some time exploring low-reward areas during the learning process.

However, you may know some parts of the state space that are not worth exploring. By providing a model of the environment, or part of the environment, you provide the agent with this knowledge.

Using a model, the agent can explore parts of the environment without having to physically take that action. A model can complement the learning process by avoiding areas that are known to be bad and exploring the rest.

Model Free vs. Model Based



Model-based reinforcement learning can lower the time it takes to learn an optimal policy because you can use the model to guide the agent away from areas of the state space that you know have low rewards.

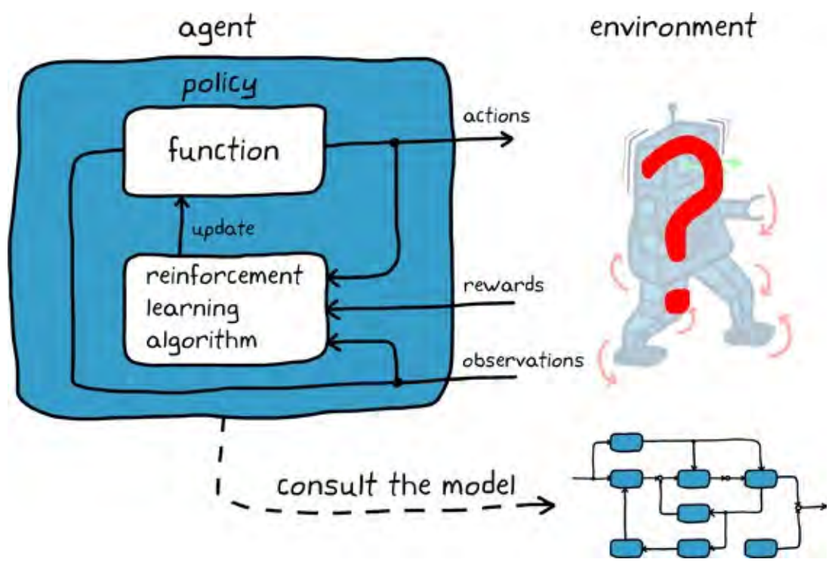
You don't want the agent to reach these low-reward states in the first place, so you don't need it to spend time learning what the best actions would be in those states.

With model-based reinforcement learning, you don't need to know the full environment model; you can provide the agent with just the parts of the environment you know.

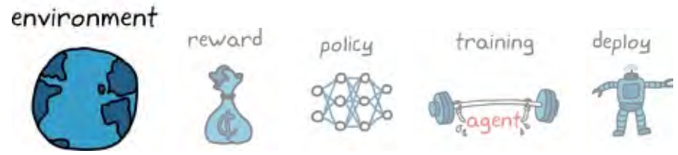
Model-free reinforcement learning is the more general case and will be the focus for the rest of this ebook.

If you understand the basics of reinforcement learning without a model, then continuing on to model-based RL is more intuitive.

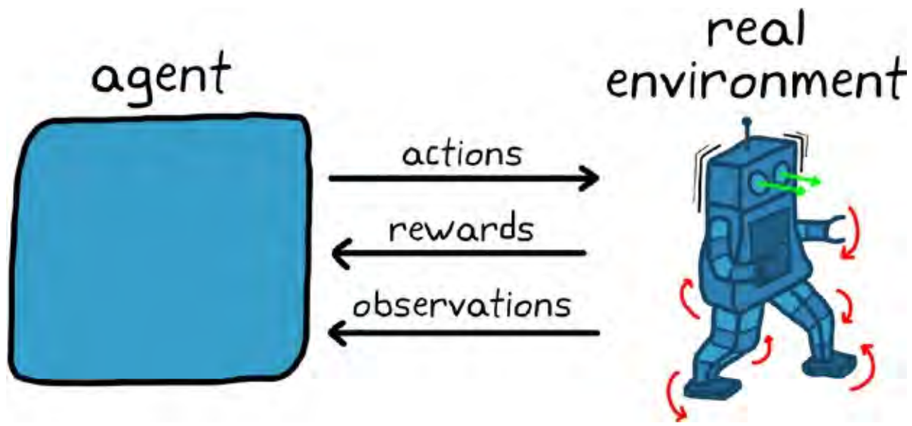
Model-free RL is popular right now because people hope to use it to solve problems where developing a model—even a simple one—is difficult. An example is controlling a car or a robot from pixel observations. It's not intuitively obvious how pixel intensities relate to car or robot actions in most situations.



Real vs. Simulated Environments



Since the agent learns through interaction with the environment, you need a way for the agent to actually interact with it. This might be a real physical environment or a simulation, and choosing between the two depends on the situation.

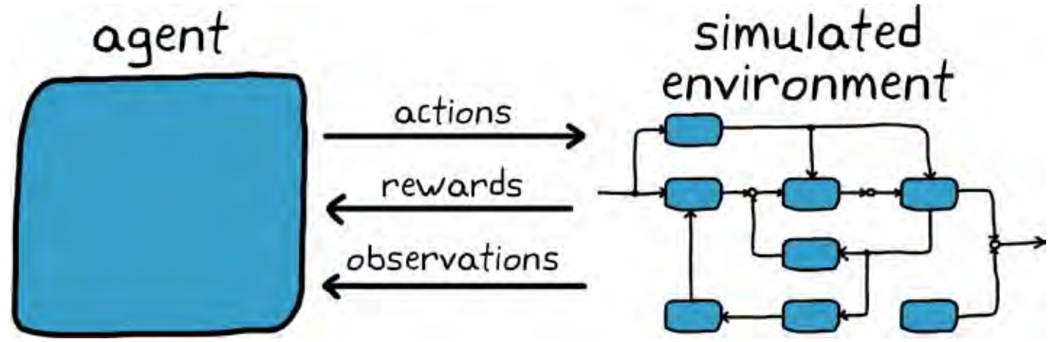


Real

Accuracy: Nothing represents the environment more completely than the real environment.

Simplicity: There is no need to spend the time creating and validating a model.

Necessary: It might be necessary to train with the real environment if it is constantly changing or difficult to model accurately.



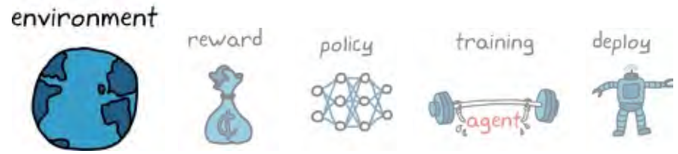
Simulated

Speed: Simulations can run faster than real time or be parallelized, speeding up a slow learning process.

Simulated conditions: It is easier to model situations that would be difficult to test.

Safety: There is no risk of damage to hardware.

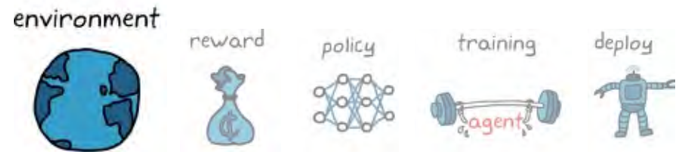
Real vs. Simulated Environments



For example, you could let an agent learn how to balance an inverted pendulum by running it with a physical pendulum setup. This might be a good solution since it's probably hard for the hardware to damage itself or others. Since the state and action spaces are relatively small, it probably won't take too long to train.

With the walking robot this might not be such a good idea. If the policy is not sufficiently optimal when you start training, the robot is going to do a lot of falling and flailing before it even learns how to move its legs, let alone how to walk. Not only could this damage the hardware, but having to pick the robot up each time would be extremely time consuming. Not ideal.

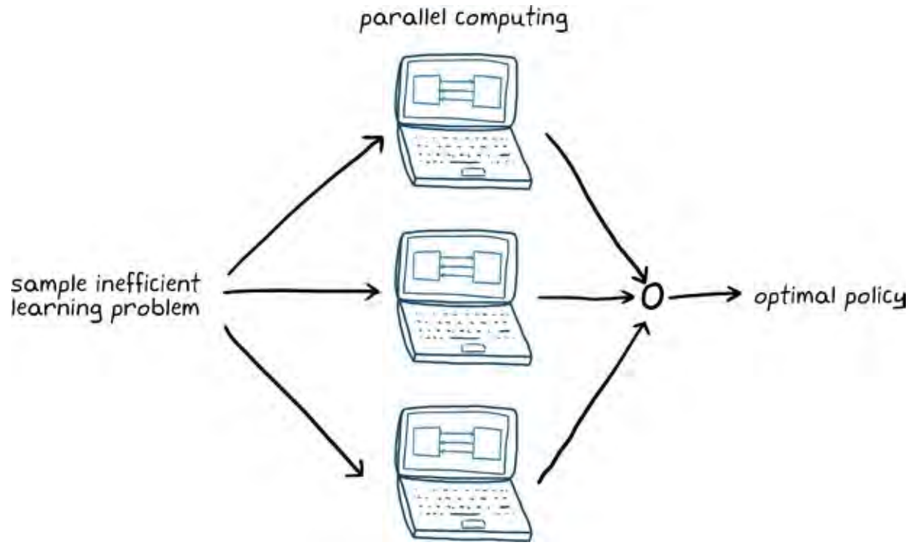
Benefits of a Simulated Environment

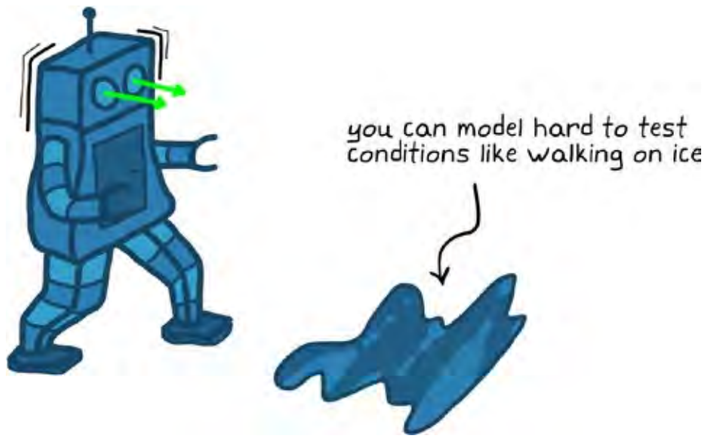


Simulated environments are the most common way to train an agent. One nice benefit for control problems is that you usually already have a good model of the system and environment since you typically need it for traditional control design. If you already have a model built in MATLAB® or Simulink®, you can replace your existing controller with a reinforcement learning agent, add a reward function to the environment, and start the learning process.

Learning is a process that requires lots of samples: trials, errors, and corrections. It is very inefficient in this sense because it can take thousands or millions of episodes to converge on an optimal solution.

A model of the environment may run faster than real time, and you can spin up lots of simulations to run in parallel. Both of these approaches can speed up the learning process.





You have a lot more control over simulating conditions than you do exposing your agent to them in the real world.

For example, your robot may have to be capable of walking on any number of different surfaces. Simulating walking on a low-friction surface like ice is much simpler than testing on actual ice. Additionally, training an agent in a low-friction environment would actually help the robot stay upright on all surfaces. It's possible to create a better training environment with simulation.

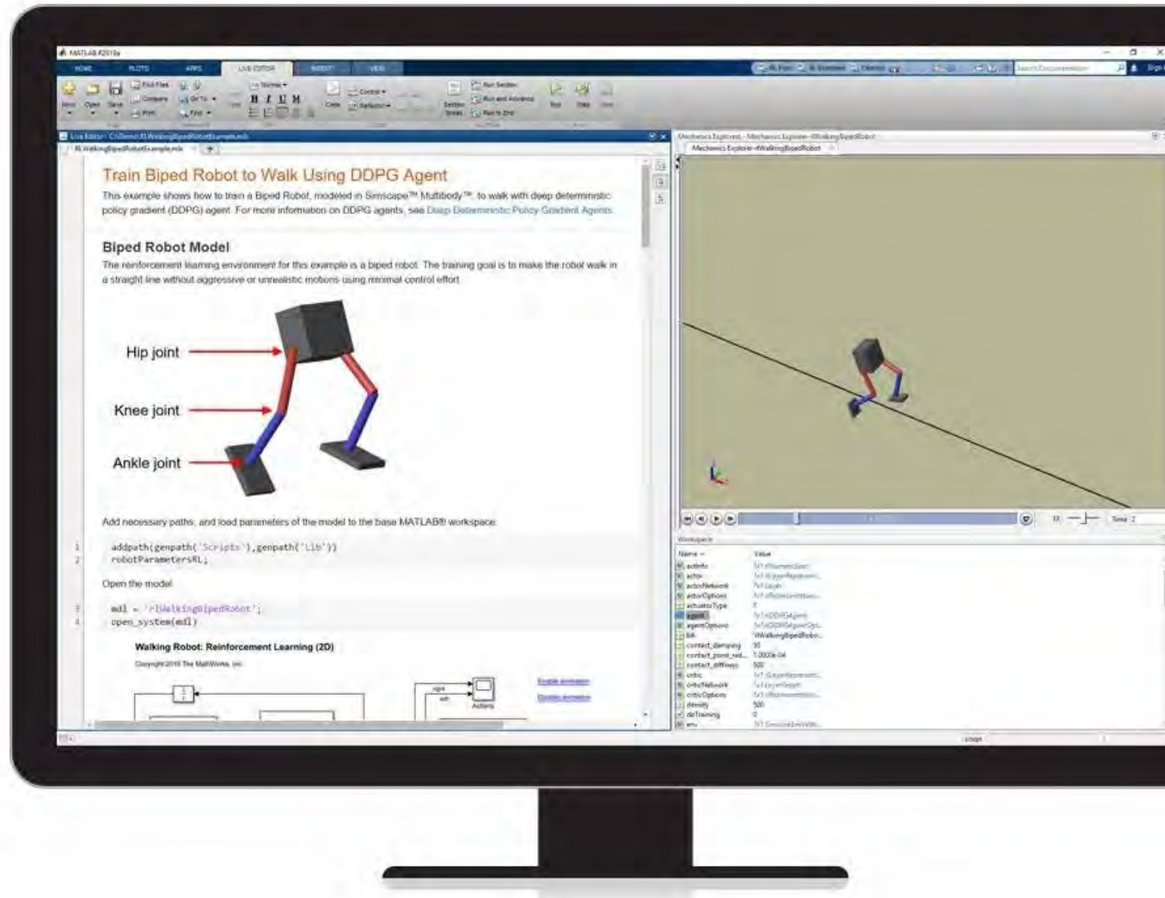
Reinforcement Learning with MATLAB and Simulink

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 train policies by enabling them to interact with environments represented in MATLAB or using Simulink models.

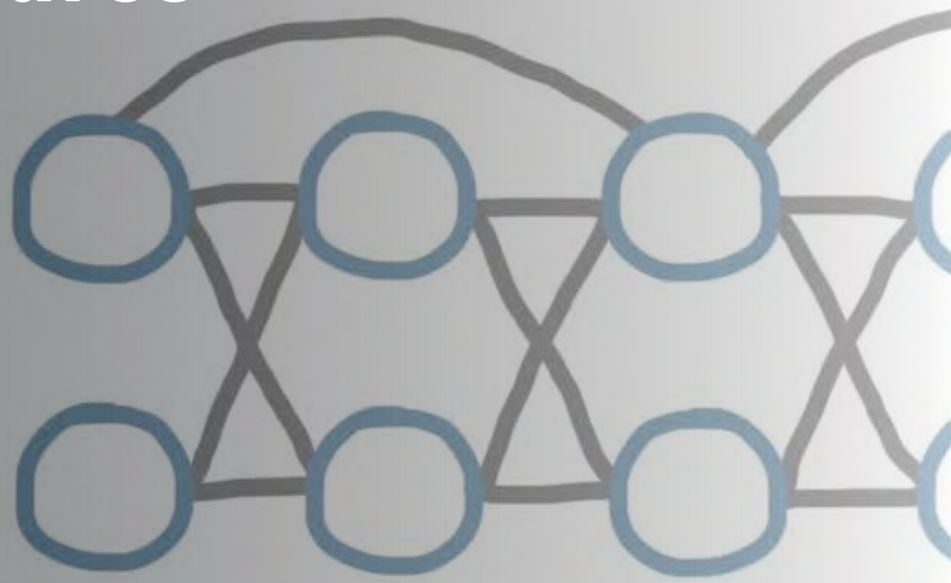
For example, for defining reinforcement learning environments in MATLAB, you can use provided template scripts and classes and modify the environment dynamics, reward, observations, and actions as needed depending on the application.

In Simulink, you can model the many different types of environments that are often needed to solve controls and reinforcement learning problems. For example, you can model vehicle dynamics and flight dynamics; a variety of physical systems with Simscape™; dynamics approximated from measured data with System Identification Toolbox™; sensors such as radars, lidars, and IMUs; and more.

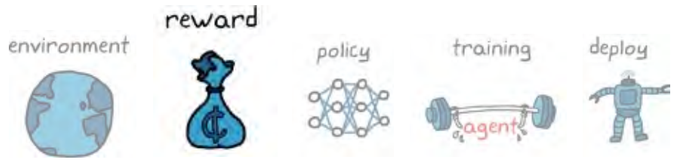
mathworks.com/products/reinforcement-learning



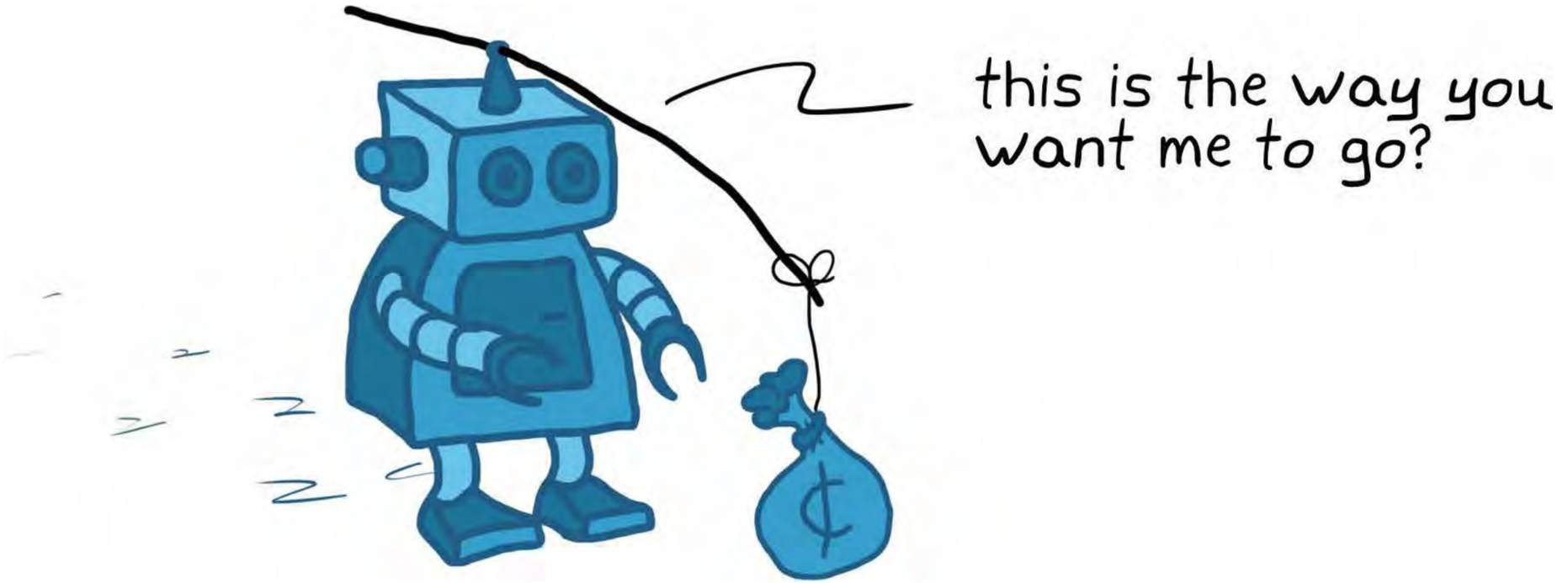
Part 2: Rewards and Policy Structures



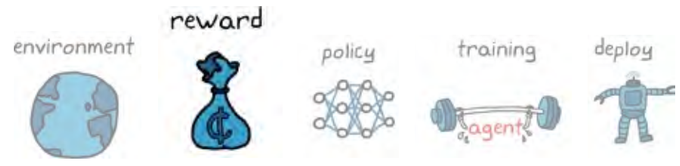
The Reward



With the environment set, the next step is to think about what you want your agent to do and how you'll reward it for doing what you want. This requires crafting a reward function so that the learning algorithm "understands" when the policy is getting better and ultimately converges on the result you're looking for.

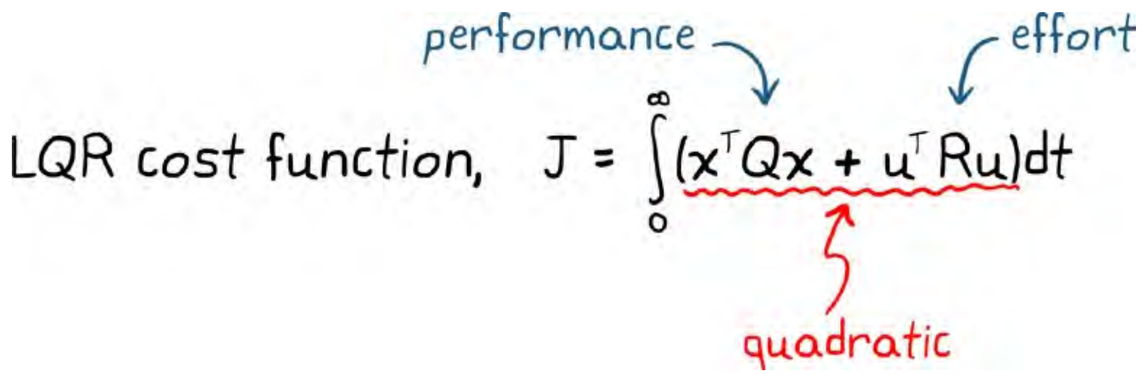


What Is Reward?



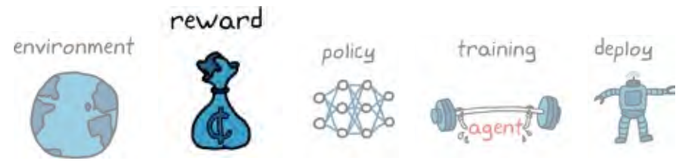
Reward is a function that produces a scalar number that represents the "goodness" of an agent being in a particular state and taking a particular action.

The concept is similar to the cost function in LQR, which penalizes bad system performance and increased actuator effort. The difference, of course, is that a cost function is trying to minimize the value, whereas a reward function tries to maximize the value. But this is solving the same problem since rewards can be thought of as the negative of cost.



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

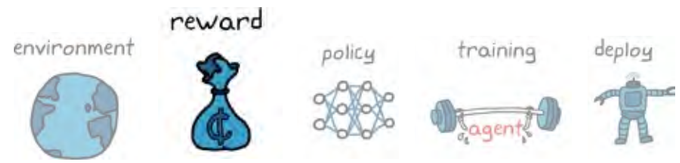
Sparse Rewards



Since there are no constraints on how you create your reward function, you can get into situations where the rewards are sparse. This means that the goal you want to incentivize comes after a long sequence of actions. This would be the case for the walking robot if you set up the reward function such that the agent only receives a reward after the robot successfully walks 10 meters. Since that is ultimately what you want the robot to do, it makes perfect sense to set up the reward like this.

The problem with sparse rewards is that your agent may stumble around for long periods of time, trying different actions and visiting a lot of different states without receiving any rewards along the way and, therefore, not learning anything in the process. The chance that your agent will randomly stumble on the exact action sequence that produces the sparse reward is very unlikely. Imagine the luck needed to generate all of the correct motor commands to keep a robot upright and walking 10 meters rather than just flopping around on the ground!

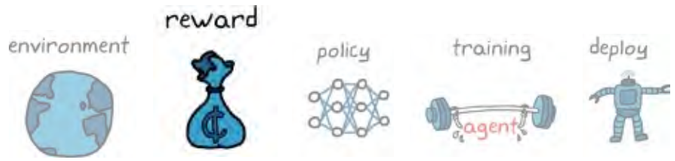
Reward Shaping



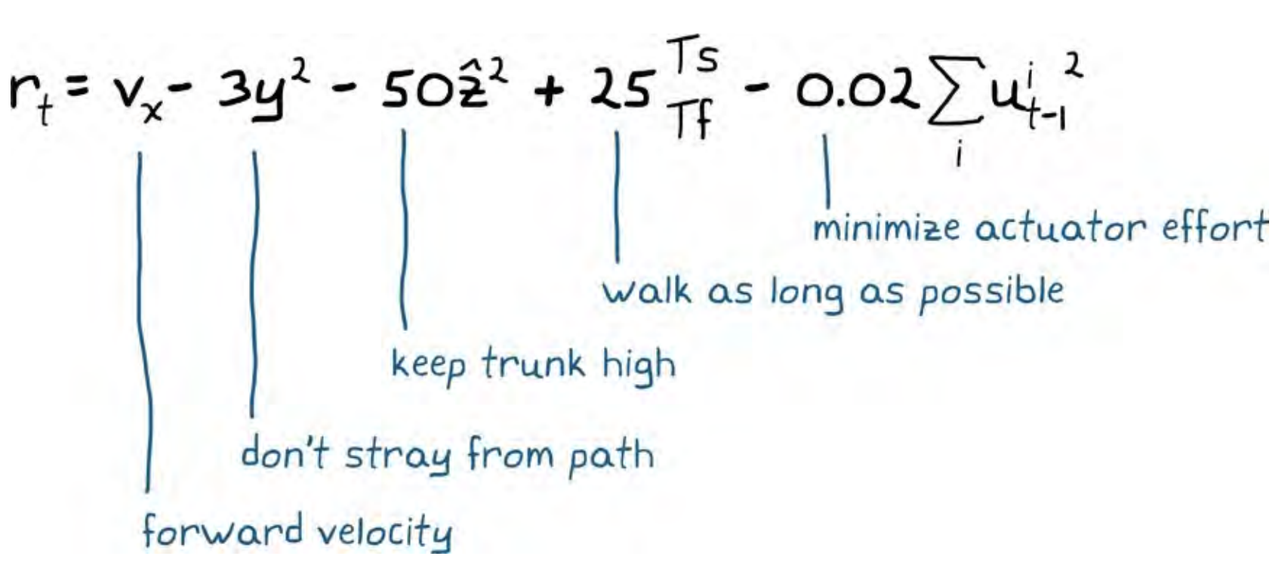
You can improve sparse rewards through reward shaping—providing smaller intermediate rewards that guide the agent along the right path.

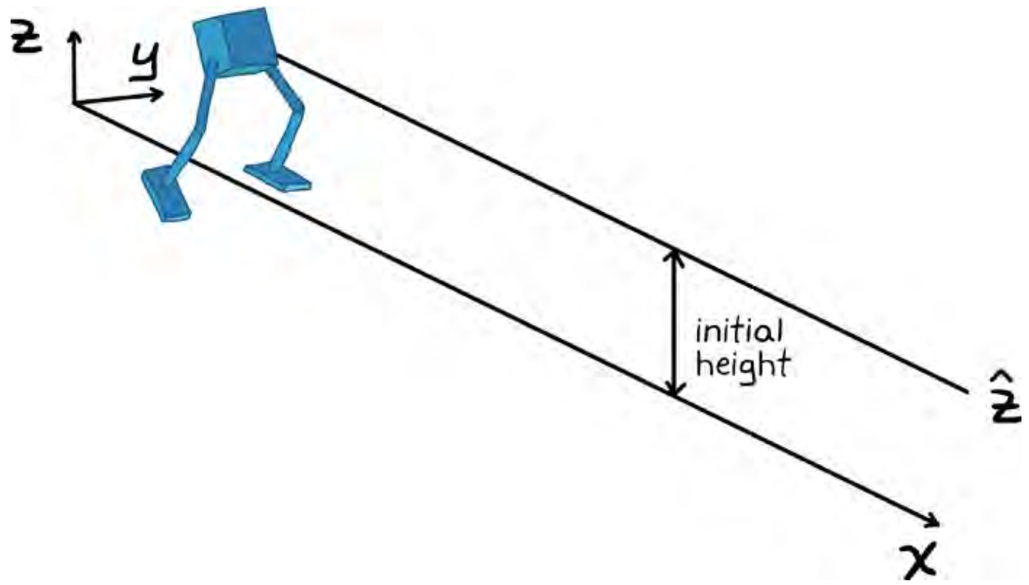
Reward shaping, however, comes with its own set of problems. If you give an optimization algorithm a shortcut, it'll take it! And shortcuts are hidden within reward functions—more so when you start shaping them. A poorly shaped reward function might cause your agent to converge on a solution that is not ideal, even if that solution produces the most rewards for the agent. It might seem like our intermediate rewards will guide the robot to successfully walk toward the 10 meter goal, but the optimal solution might not be to walk to that first reward. Instead it may fall ungracefully toward it, collect the reward, thereby reinforcing that behavior. Beyond that, the robot might converge on inchworming along the ground to collect the rest of the reward. To the agent, that is a perfectly reasonable high-reward solution, but obviously to the designer it's not a preferred result.

Domain-Specific Knowledge



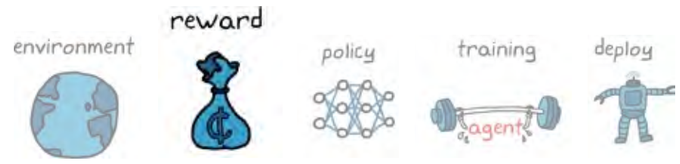
Reward shaping isn't always used to fill in for sparse rewards. It is also a way that engineers can inject domain-specific knowledge into the agent. For example, if you know that you want the robot to walk, rather than crawl along the ground, you can reward the agent for keeping the trunk of the robot at a walking height. You could also reward low actuator effort, staying on its feet longer, and not straying from the intended path.



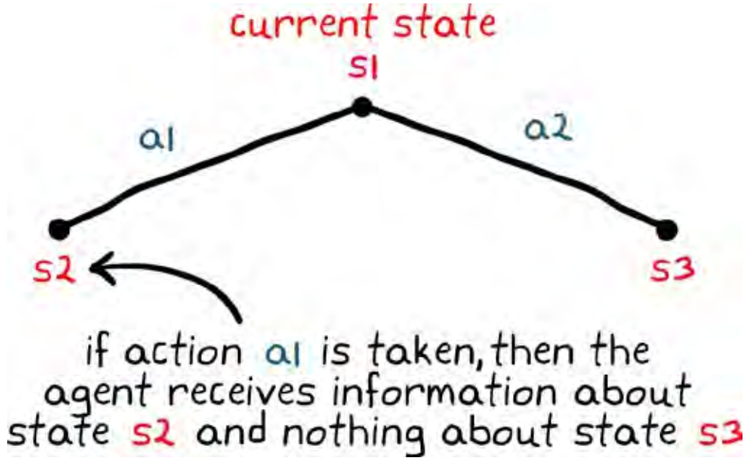


This is not intended to make crafting a reward function sound easy; getting it right is possibly one of the more difficult tasks in reinforcement learning. For example, you might not know if your reward function is poorly crafted until after you've spent a lot of time training your agent and it failed to produce the results you were looking for. However, with this general overview, you'll be in a better position to at least understand some of the things that you need to watch out for and that might make crafting the reward function a little easier.

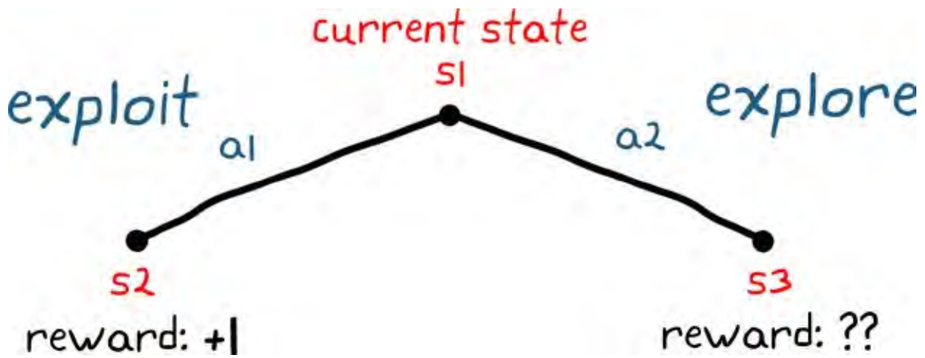
Exploration vs. Exploitation



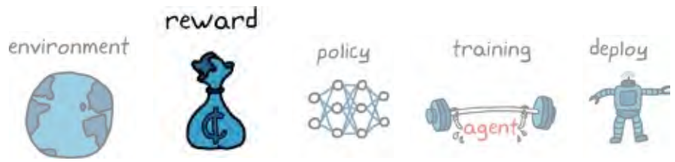
A critical aspect of reinforcement learning is the tradeoff between exploration and exploitation while an agent interacts with an environment. The reason this decision comes up with reinforcement learning is that learning is done online. Instead of working from a static dataset, the agent's actions determine which data is returned from the environment. The choices the agent makes determine the information it receives and, therefore, the information from which it can learn.



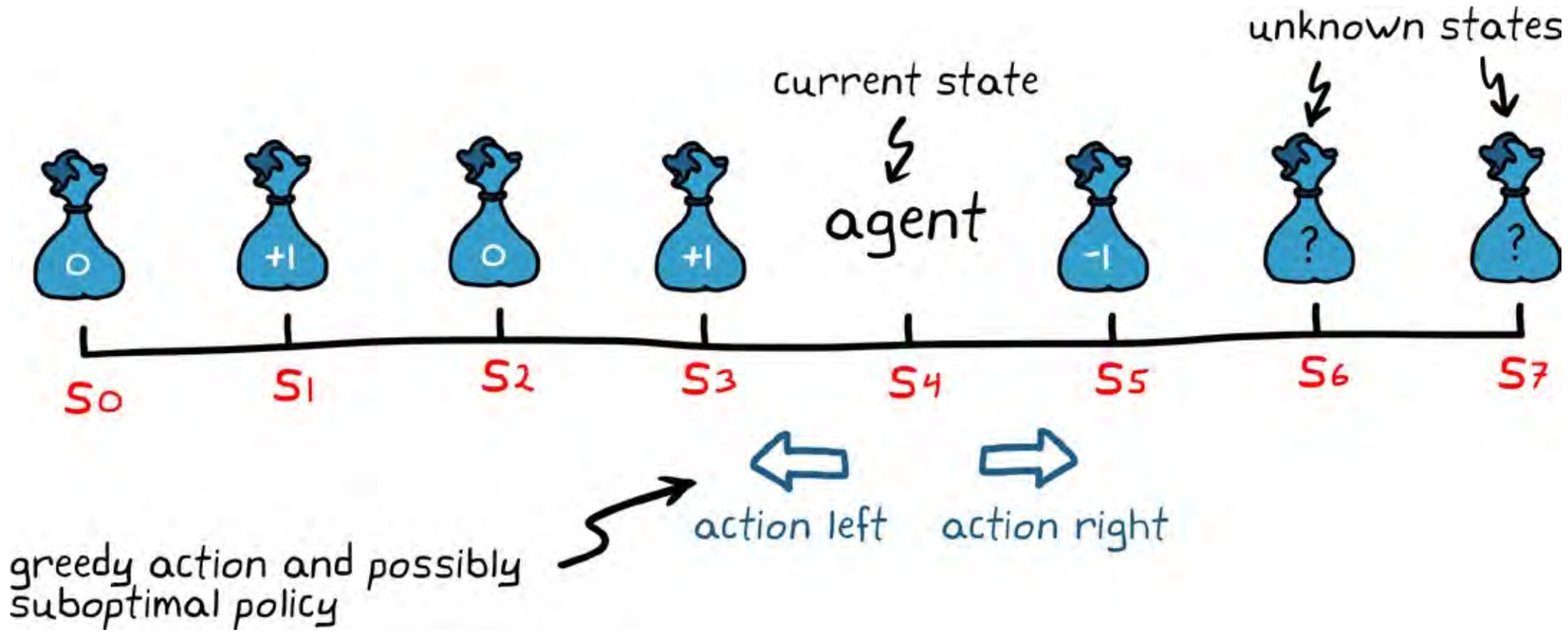
The idea is this: Should the agent exploit the environment by choosing the actions that collect the most rewards that it already knows about, or should it choose actions that explore parts of the environment that are still unknown?



The Problem with Pure Exploitation

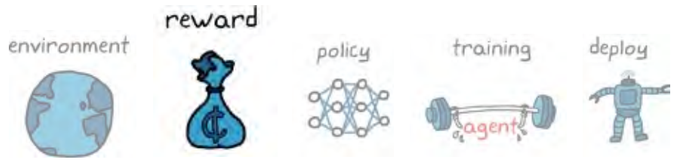


For example, let's say an agent is in a particular state and it can take one of two actions: go left or go right. It knows that going left will produce +1 reward and going right produces -1 reward. The agent doesn't know anything else about the environment to the right of that initial low-reward state. If the agent takes the greedy approach by always exploiting the environment, it would go left to collect the highest reward it knows about and ignore the other states completely.



So you can see that if the agent is always exploiting what it thinks is the best action at any given time, it may never receive additional information about the states that exist beyond a low-reward action. This pure exploitation can increase the amount of time it takes to find the optimal policy or may cause the learning algorithm to converge on a suboptimal policy since whole sections of the state space may never be explored.

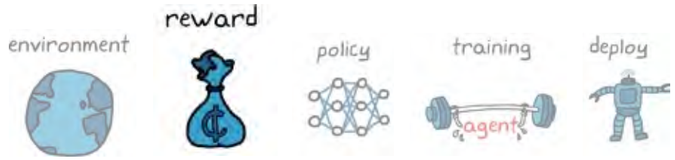
The Problem with Pure Exploration



Instead, if you occasionally let the agent explore, even at the risk of collecting fewer rewards, it can expand its policy for the new states. This opens up the possibility of finding higher rewards it didn't know about and increases the chances of converging on the global solution. But you don't want the agent to explore too much because there is a downside with this approach as well. For one, pure exploration is not a good approach when training on physical hardware because the agent runs a risk of exploring an action that causes damage. Think about the damage that can be caused by an autonomous car that is exploring random steering wheel inputs while on the highway.

However, even with a simulated environment where damage isn't an issue, pure exploration is not an efficient way to learn because the agent will likely spend time covering a bigger portion of the state space. While this is beneficial for finding a global solution, excessive exploration can slow the learning rate by so much that no sufficient solution is found in a reasonable amount of learning time. Therefore, the best learning algorithms strike a balance between exploring and exploiting the environment.

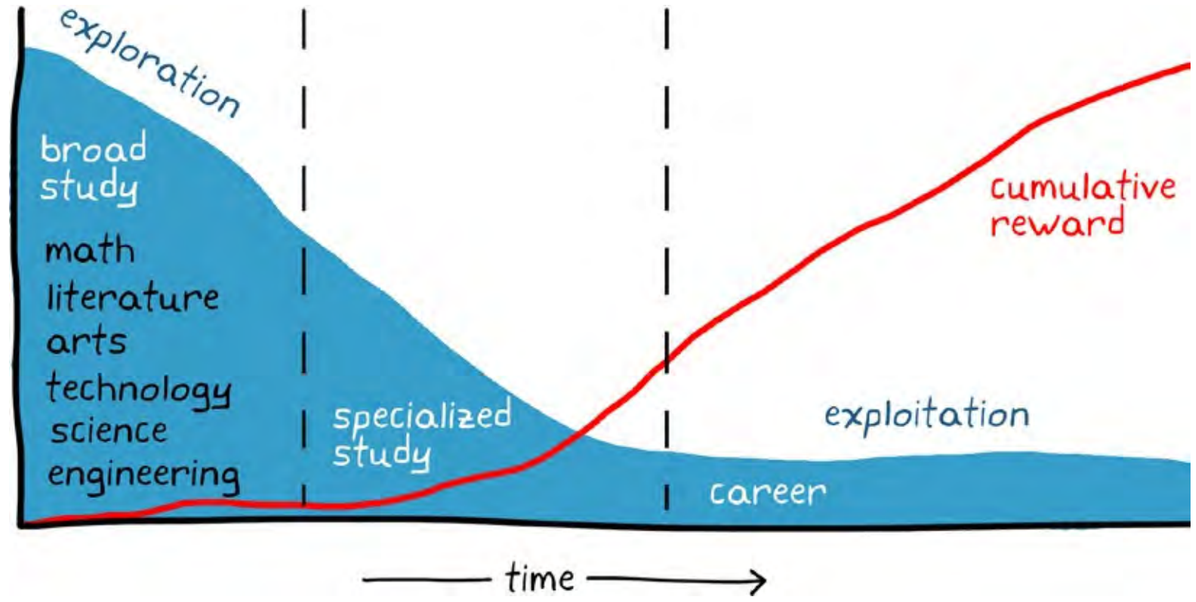
Balancing Exploration and Exploitation



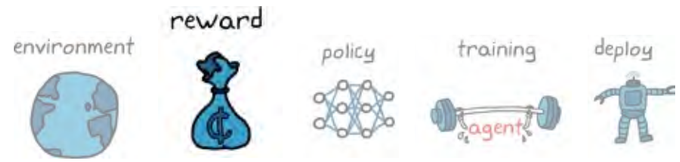
Consider how student might approach choosing a career path. When students are young, they explore different subjects and classes and are generally open to new experiences. After a certain amount of exploration, they are then likely to converge on learning more about a specialized subject and then finally converge on a career that they feel will have the highest combination of financial return and job satisfaction (reward).

One lifetime would probably not be enough to explore every possible career option. Therefore, students will have to decide on the most optimal career path of the options they've explored so far. If they put off exploiting their knowledge for too long and continue to explore new career options, then there won't be as much time available to collect the return on their effort.

Even though reinforcement learning algorithms provide a simple way to balance exploration and exploitation, it might not be obvious where to set that balance throughout the learning process so that the agent settles on a sufficient policy within the time allotted for learning. In general, however, an agent explores more at the start of learning and gradually transitions to more of an exploitation role by the end, just like the students.



The Value of Value

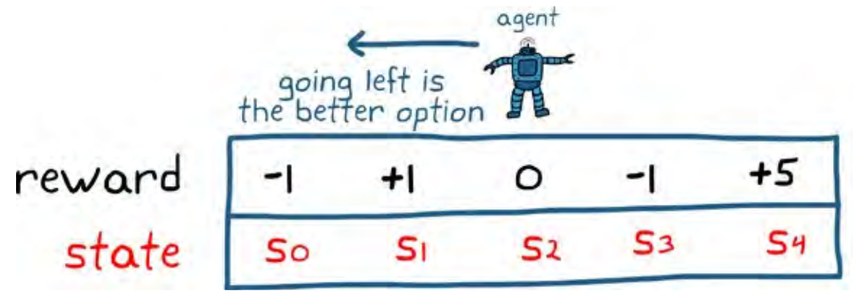


A second critical aspect of reinforcement learning is the concept of *value*. Assessing the value of a state or an action, rather than reward, helps the agent choose the action that will collect the most rewards over time rather than a short-term benefit.

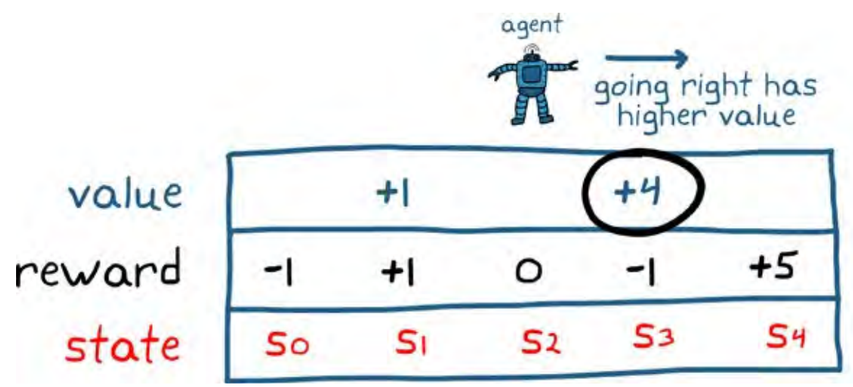
reward: the instantaneous benefit of being in a state and taking a specific action

value: the total rewards an agent expects to receive from a state and onwards into the future

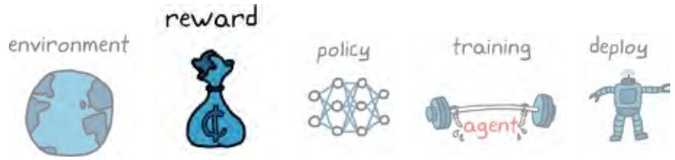
For example, imagine our agent is trying to collect the most rewards within two steps. If the agent looks only at the reward for each action, it will step left first since that produces a higher reward than right. Then it'll go back right since that again is the highest reward, to ultimately collect a total of +1.



However, if the agent is able to estimate the value of a state, then it will see that going right has a higher value than going left even though the reward is lower. Using value as its guide, the agent will ultimately end up with +4 total reward.



The Benefit of Being Short-Sighted



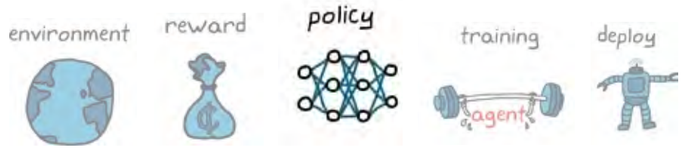
Of course, the promise of receiving a high reward after many sequential actions doesn't mean that the first action is necessarily the best; there are at least two good reasons for this.

First, like with the financial market, money in your pocket now can be And second, your prediction of rewards further into the future becomes better than a little more money in your pocket a year from now. less reliable; therefore, that high reward might not be there by the time the agent reaches it.

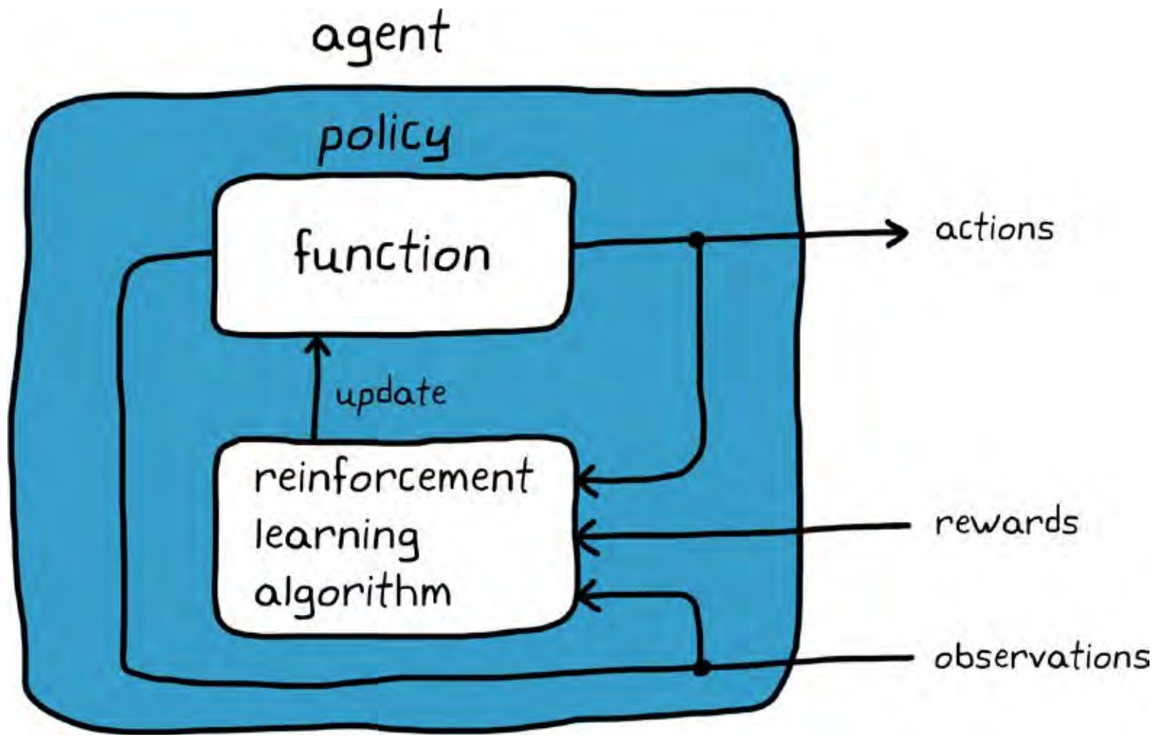
In both of these cases, it's advantageous to be a little more short-sighted when estimating value. In reinforcement learning, you can set how short-sighted you want your agent to be by discounting rewards by a larger amount the further they are in the future. This is done by setting the discount factor, gamma, between 0 and 1.

$$\text{total discounted reward} = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \ldots = \sum_{i=1}^{T} \gamma^{i-1} r_i$$

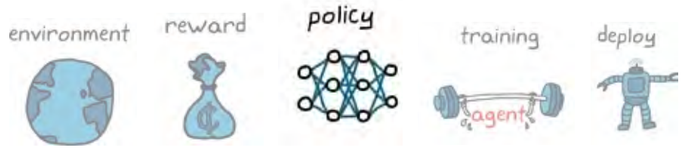
What Is the Policy?



Now that you understand the environment and its role in providing the state and the rewards, you're ready to start work on the agent itself. The agent is composed of the policy and the learning algorithm. The *policy* is the function that maps observations to actions, and the *learning algorithm* is the optimization method used to find the optimal policy.



Representing a Policy



At the most basic level, a policy is a function that takes in state observations and outputs actions. So if you're looking for ways to represent a policy, any function with that input and output relationship can work.

policy ⇒ actions = function (state observations)

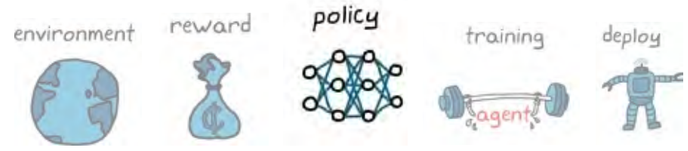
In general, there are two approaches for structuring the policy function:

- Direct: There is a specific mapping between state observations and action.
- Indirect: You look at other metrics like value to infer the optimal mapping.*

The next few pages show how to use a value-based method to highlight the different types of mathematical structures you can use to represent a policy. But keep in mind that these structures can be applied to policy-based functions as well.

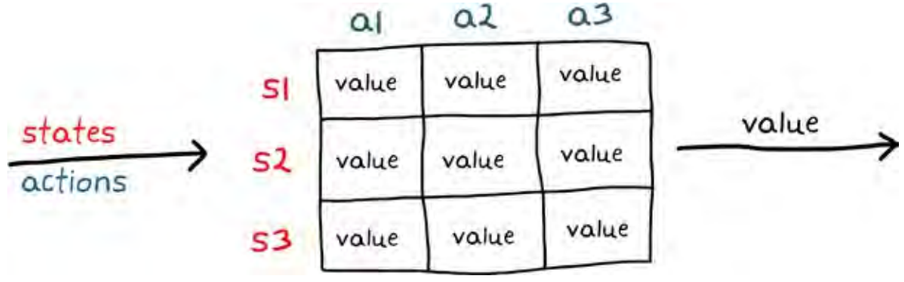
** Spoiler alert! You can combine the benefits of direct policy mapping and value-based mapping in a third method called actor-critic, which will be covered a bit later.*

Representing a Policy with a Table

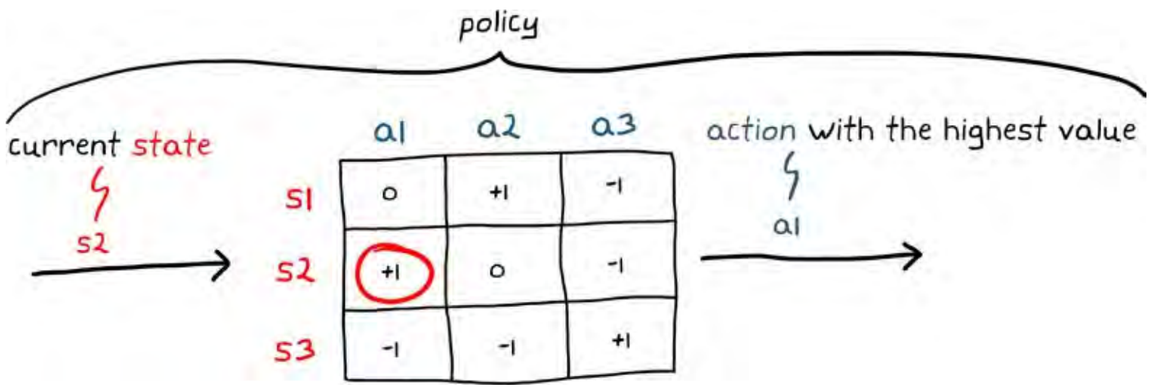


If the state and action spaces for the environment are discrete and few in number, you could use a simple table to represent policies.

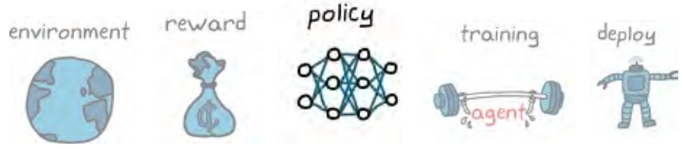
Tables are exactly what you'd expect: an array of numbers where an input acts as a lookup address and the output is the corresponding number in the table. One type of table-based function is the Q-table, which maps states and actions to value.



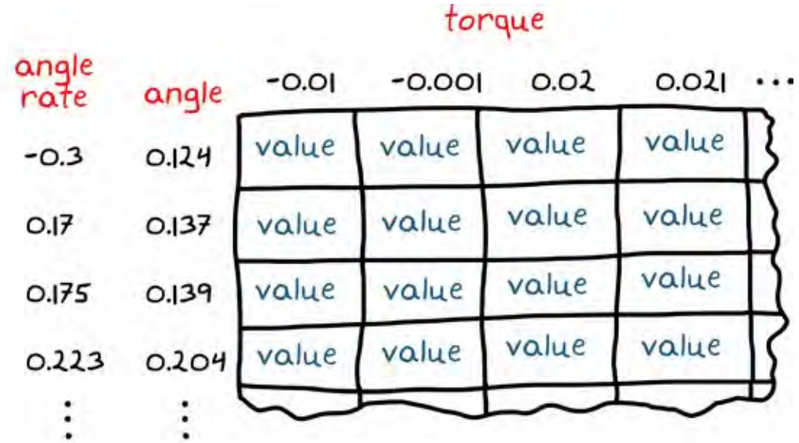
With a Q-table, the policy is to check the value of every possible action given the current state and then choose the action with the highest value. Training an agent with a Q-table would consist of determining the correct values for each state/action pair in the table. Once the table has been fully populated with the correct values, choosing the action that will produce the most long-term return of reward is pretty straightforward.



Continuous State/Action Spaces



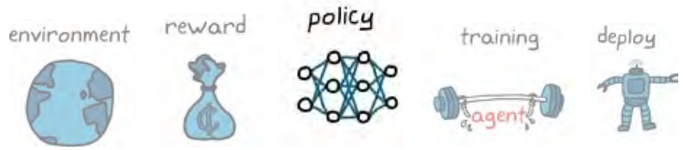
Representing policy parameters in a table is not feasible when the number of state/action pairs becomes large or infinite. This is the so-called *curse of dimensionality*. To get a feel for this, let's think about a policy that could control an inverted pendulum. The state of the pendulum can be any angle from -π to π and any angular rate. Also, the action space is any motor torque from the negative limit to the positive limit. Trying to capture every combination of every state and action in a table is impossible.



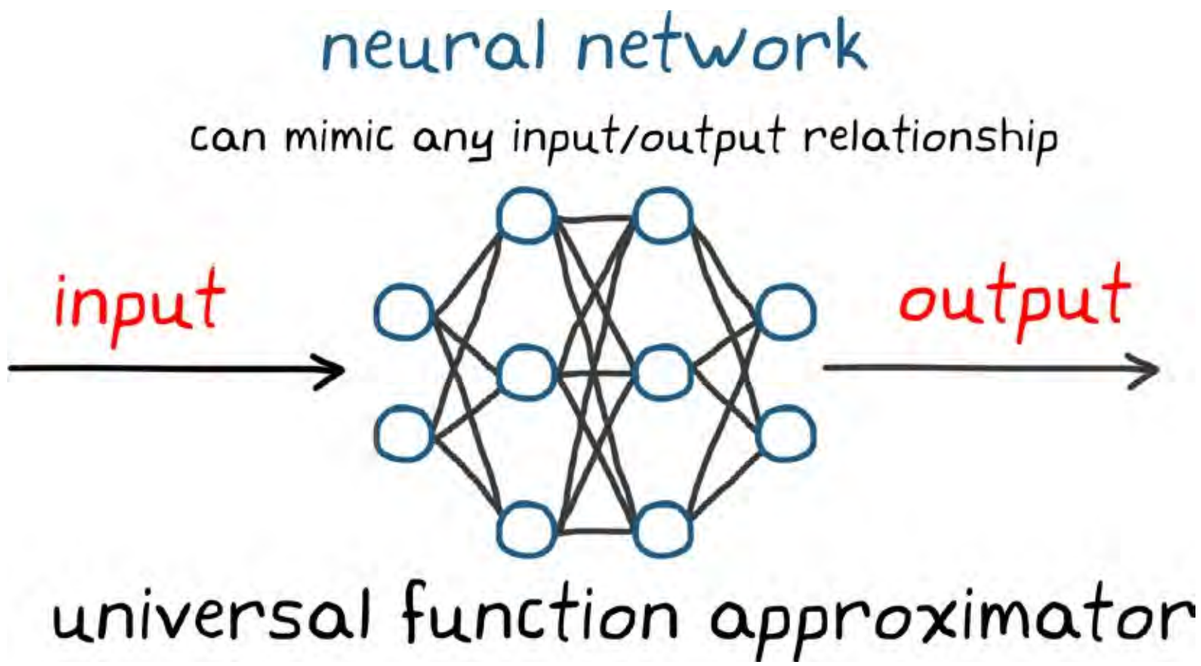
You could represent the continuous nature of the inverted pendulum with a continuous function—something that takes in states and outputs actions. However, before you could start learning the right parameters in this function, you would need to define the logical structure. This might be difficult to craft for high-degree-of-freedom systems or nonlinear systems.

So you need a way to represent a function that can handle continuous states and actions, and one that doesn't require a difficult-to-craft logical structure for every environment. This is where neural networks come in.

A Universal Function Approximator

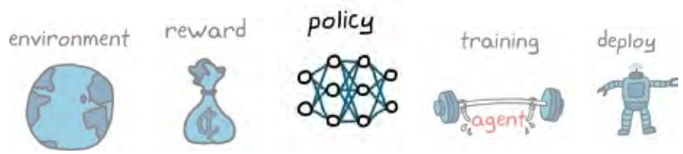


A *neural network* is a group of nodes, or artificial neurons, that are connected in a way that allows them to be a universal function approximator. This means that given the right combination of nodes and connections, you can set up the network to mimic any input and output relationship. Even though the function might be extremely complex, the universal nature of neural networks ensures that there is a neural network of some kind that can achieve it.



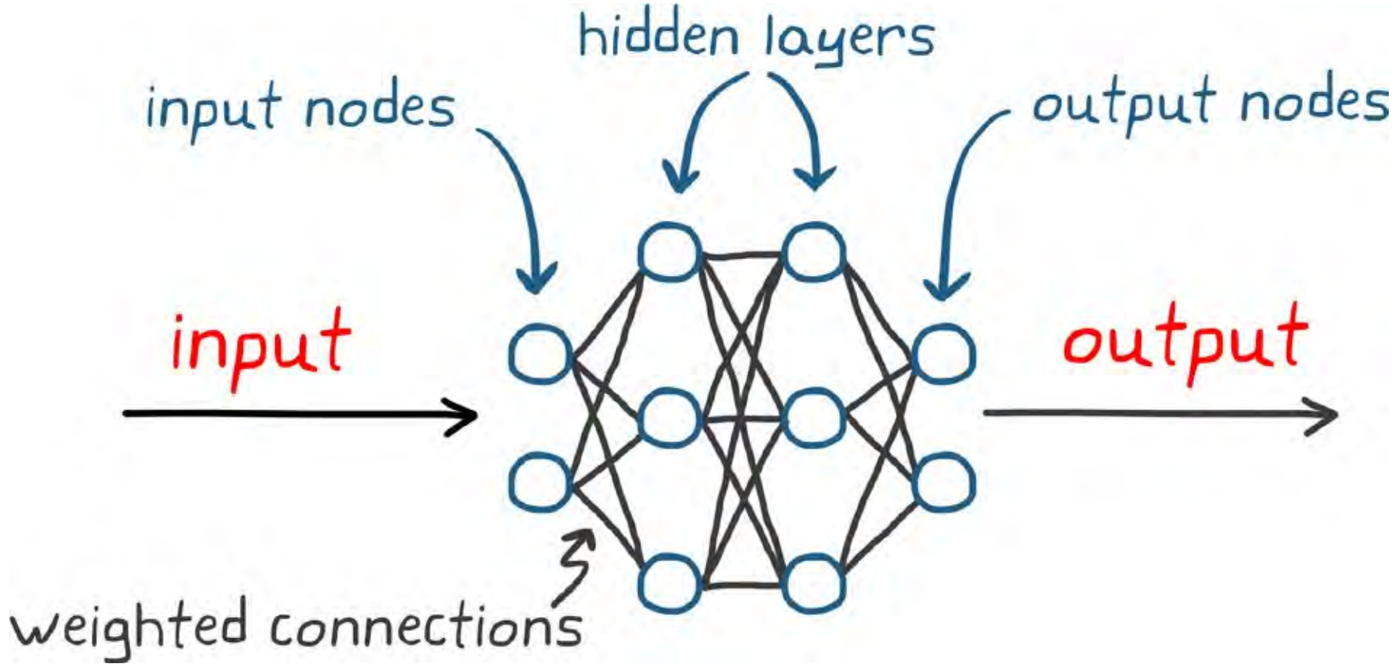
So instead of trying to find the perfect nonlinear function structure that works with a specific environment, with a neural network you can use the same combination of nodes and connections in many different environments. The only difference is in the parameters themselves. The learning process would then consist of systematically adjusting the parameters to find the optimal input/output relationship.

What Is a Neural Network?

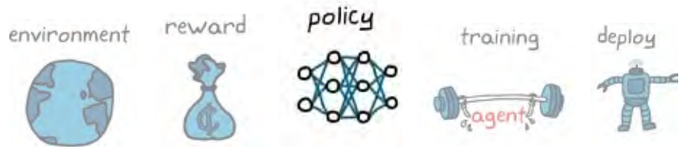


The mathematics of neural networks are not covered in depth here. But it's important to highlight a few things to help explain some of the decisions later on when setting up the policies.

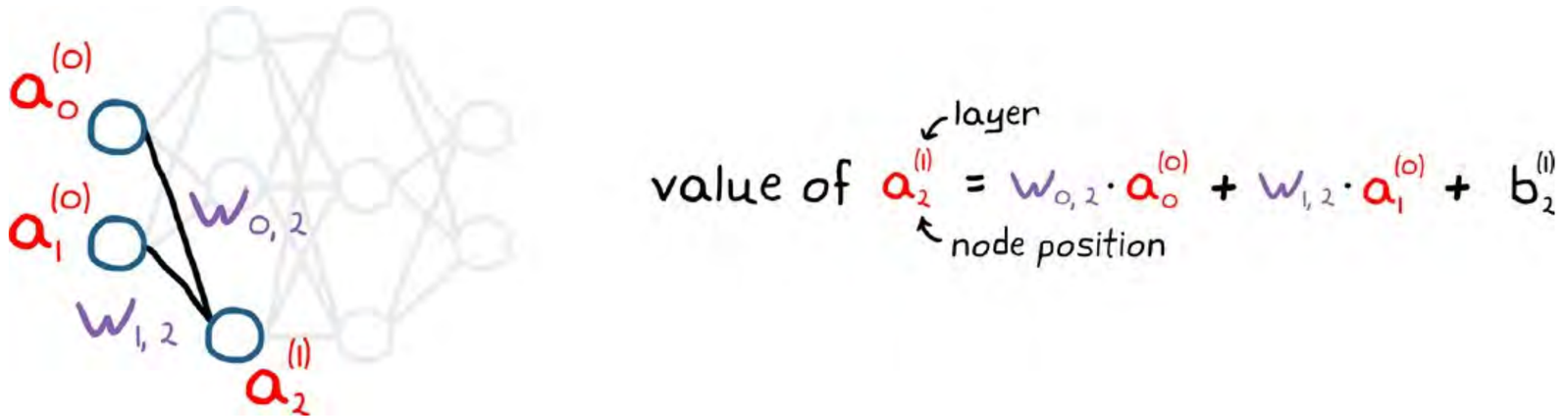
On the left are the input nodes, one for each input to the function, and on the right are the output nodes. In between are columns of nodes called hidden layers. This network has 2 inputs, 2 outputs, and 2 hidden layers of 3 nodes each. With a fully connected network, there is a weighted connection from 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.



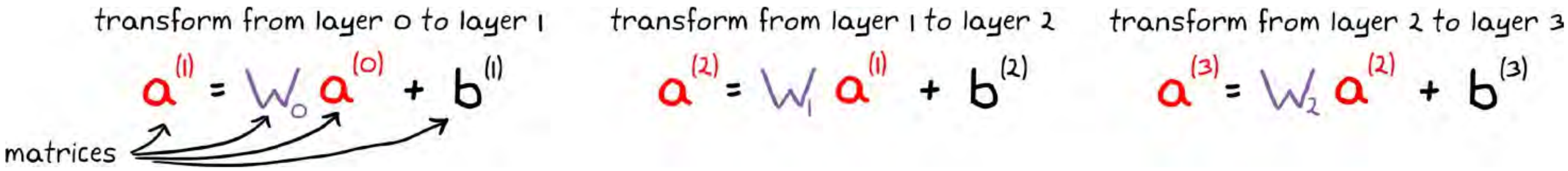
The Math Behind the Graphic



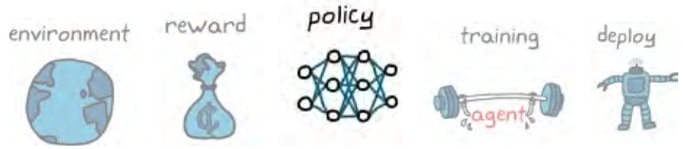
The value of any given node is equal to the sum of every node that feeds into it multiplied by its respective weighting factor plus a bias.



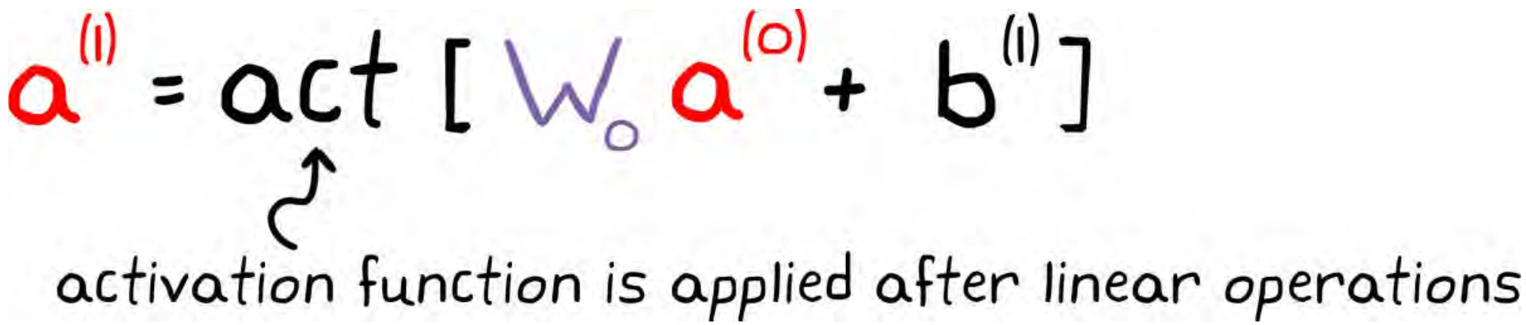
You can perform this calculation for every node in a layer and write it out in a compact matrix form as a system of linear equations. This set of matrix operations essentially transforms the numerical values of the nodes in one layer to the values of the nodes in the next layer.



The Missing Crucial Step



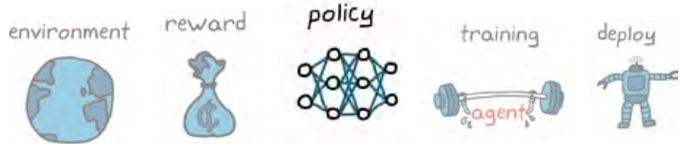
How can a bunch of linear equations operating one after another act as a universal function approximator? Specifically, how can they represent a nonlinear function? Well, there's a step that is possibly one of the most important aspects of an artificial neural network. After the value of a node has been calculated, an activation function is applied that changes the value of the node prior to it being fed as an input into the next layer.



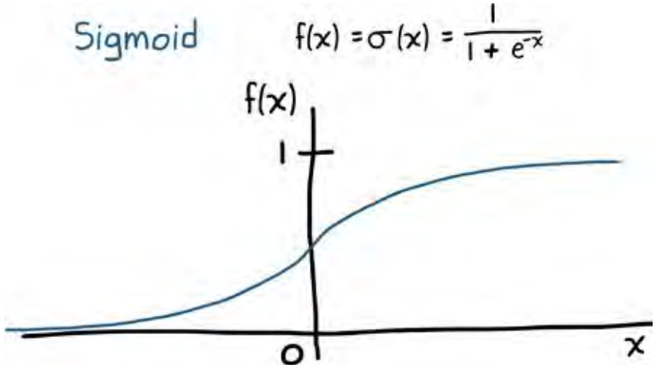
There are a number of different activation functions. What they all have in common is that they are nonlinear, which is critical to making a network that can approximate any function. Why is this the case? Because many nonlinear functions can be broken down into a weighted combination of activation function outputs.

For more detail, read *Visualizing the Universal Approximation Theorem.*

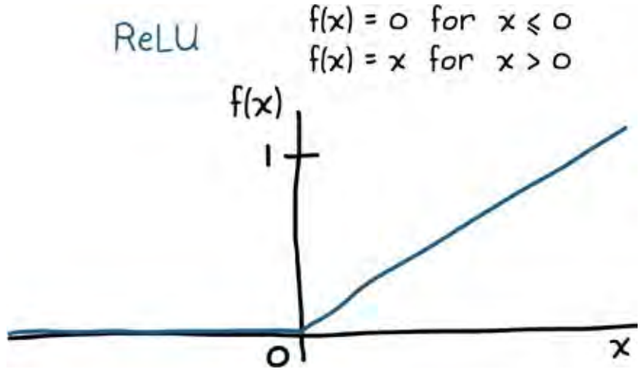
ReLU and Sigmoid Activations



The sigmoid activation function generates a smooth curve in a way that any input between negative and positive infinity is squished down to between 0 and 1.



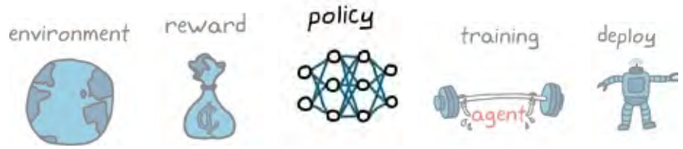
The rectified linear unit (ReLU) function zeroes out any negative node values and leaves the positive values unmodified.



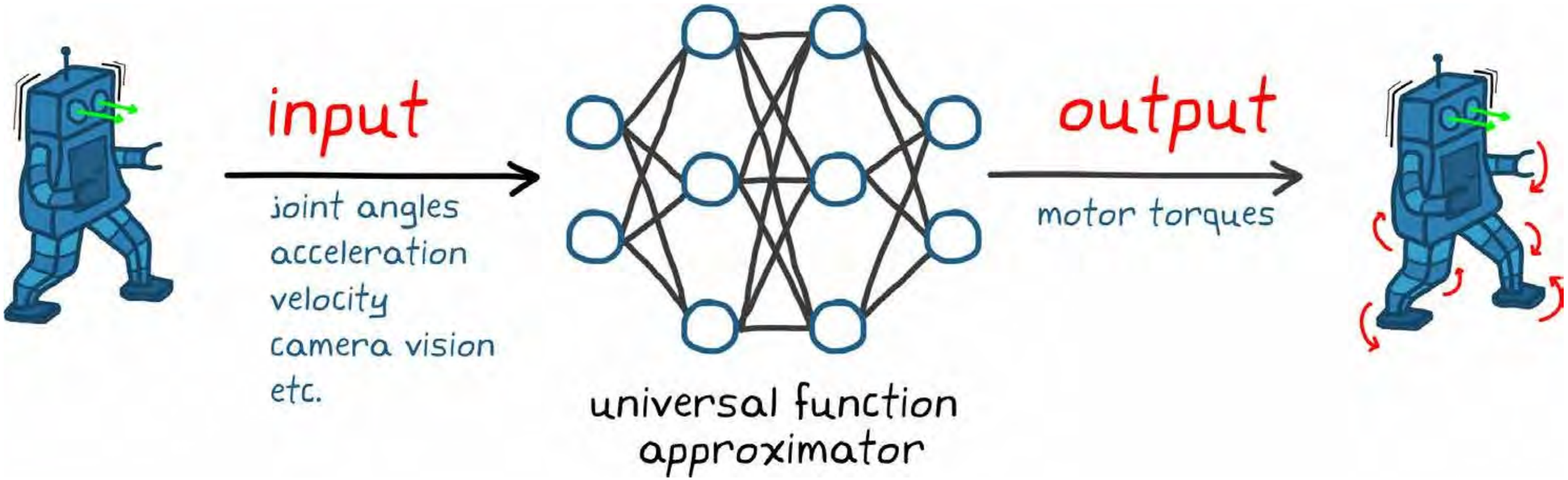
As an example, a pre-activation node value of -2 would become 0.12 with a sigmoid activation and 0 with a ReLU activation.

preactivation node value	postactivation	
	sigmoid	ReLU
-2	0.12	0
-1	0.27	0
1	0.73	1
2	0.88	2

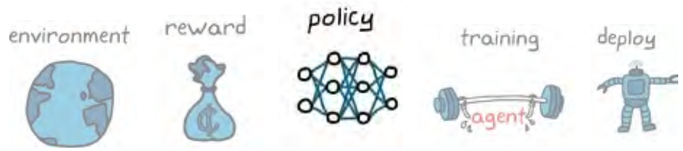
Representing a Policy with a Neural Network



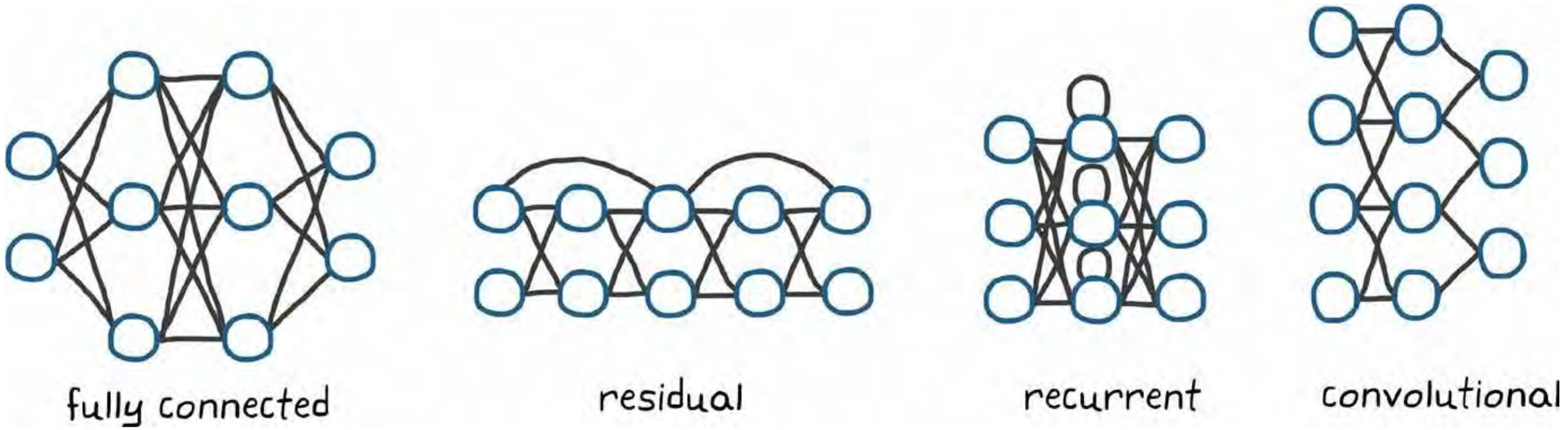
Let’s recap before moving on. You want to find a function that can take in a large number of observations and transform them into a set of actions that will control some nonlinear environment. And since the structure of this function is often too complex to solve for directly, you want to approximate it with a neural network that learns the function over time. And it’s tempting to think that you can just plug in any neural network and let loose a reinforcement learning algorithm to find the right combination of weights and biases and be done. Unfortunately, that’s not quite the case.



Neural Network Structures



You have to make a few choices about the neural network ahead of time in order to make sure it's complex enough to approximate the function you're looking for, but not so complex as to make training impossible or impossibly slow. For example, as you've already seen, you need to choose an activation function, the number of hidden layers, and the number of neurons in each layer. But beyond that you also have control over the internal structure of the network. Should it be fully connected like the network you started with, or should the connections skip layers like in a residual neural network? Should it loop back on itself to create internal memory with recurrent neural networks? Should groups of neurons work together like with a convolutional neural network?

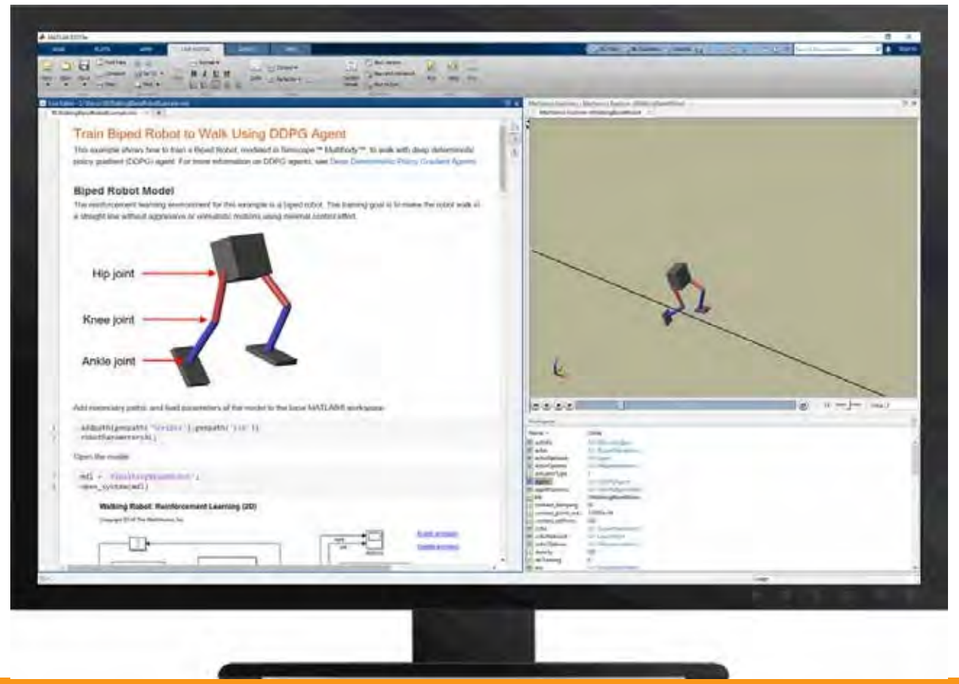


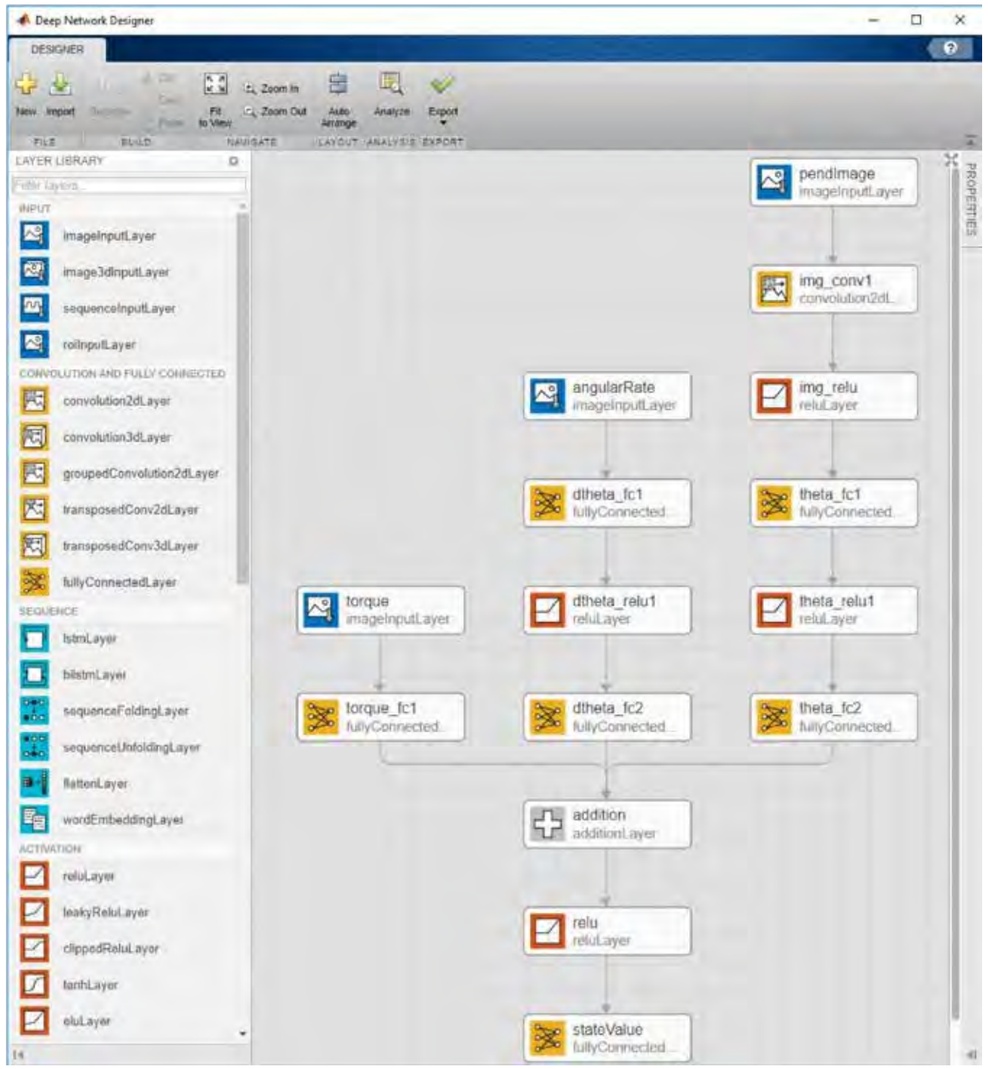
As with other control techniques, there isn't one right approach for settling on a neural network structure. A lot of it comes down to starting with a structure that has already worked for the type of problem you're trying to solve and tweaking it from there.

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.

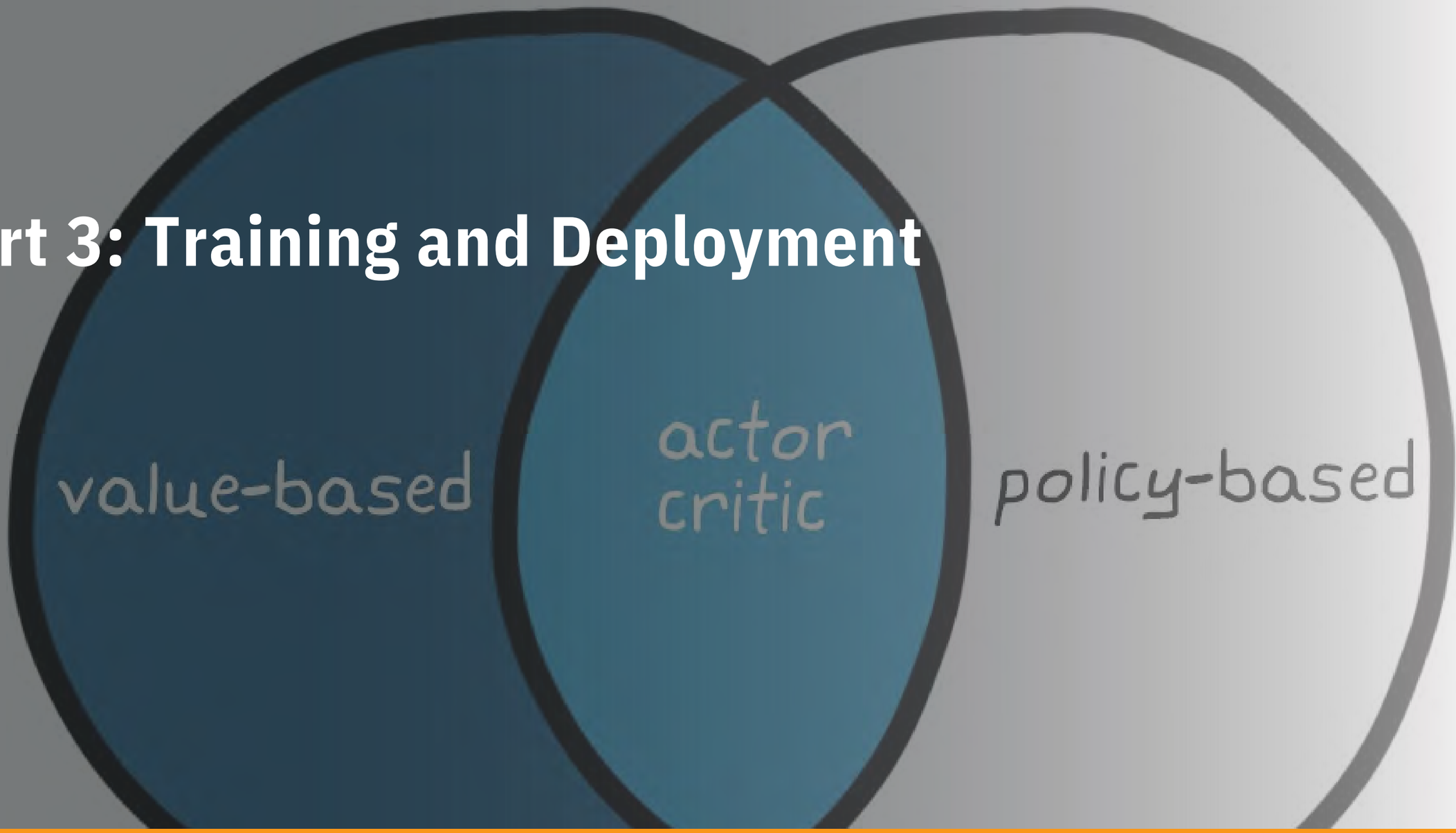
The toolbox lets you implement policies using deep neural networks, polynomials, or lookup tables. You can then train policies by enabling them to interact with environments represented by MATLAB or Simulink models.





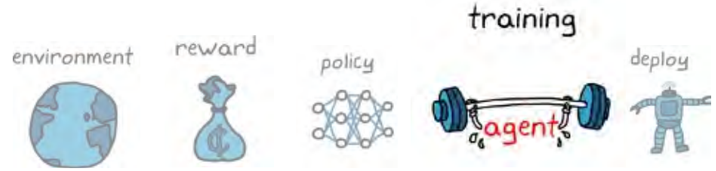
Deep Q-learning network (DQN) agent created with the Deep Network Designer app.

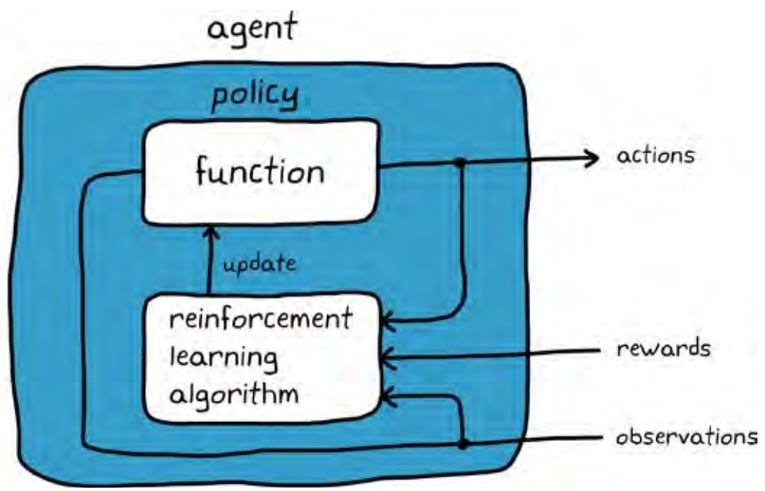
Part 3: Training and Deployment



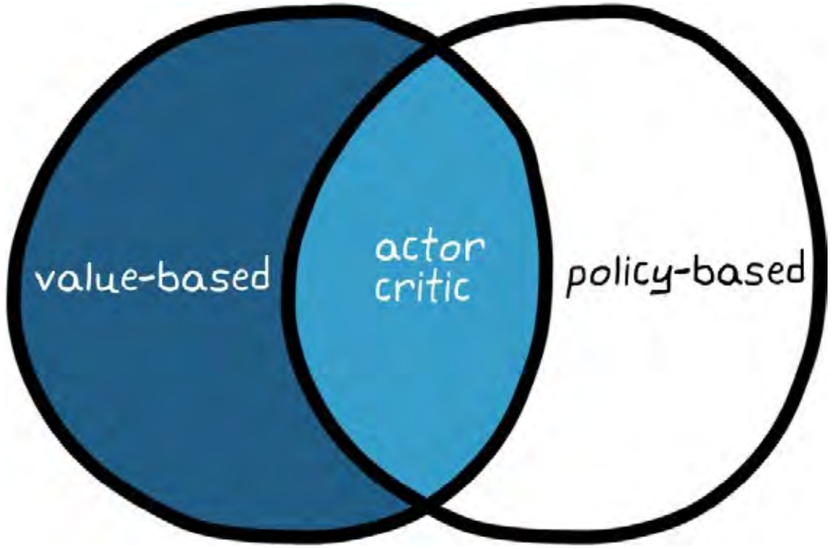
How the Policy Is Structured

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.



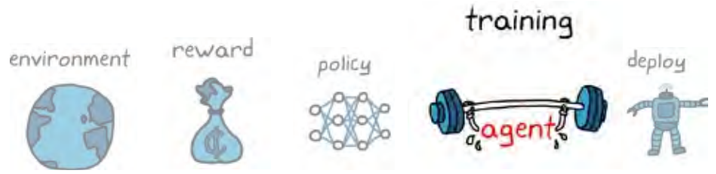


the policy structure and the RL algorithm are intimately intertwined



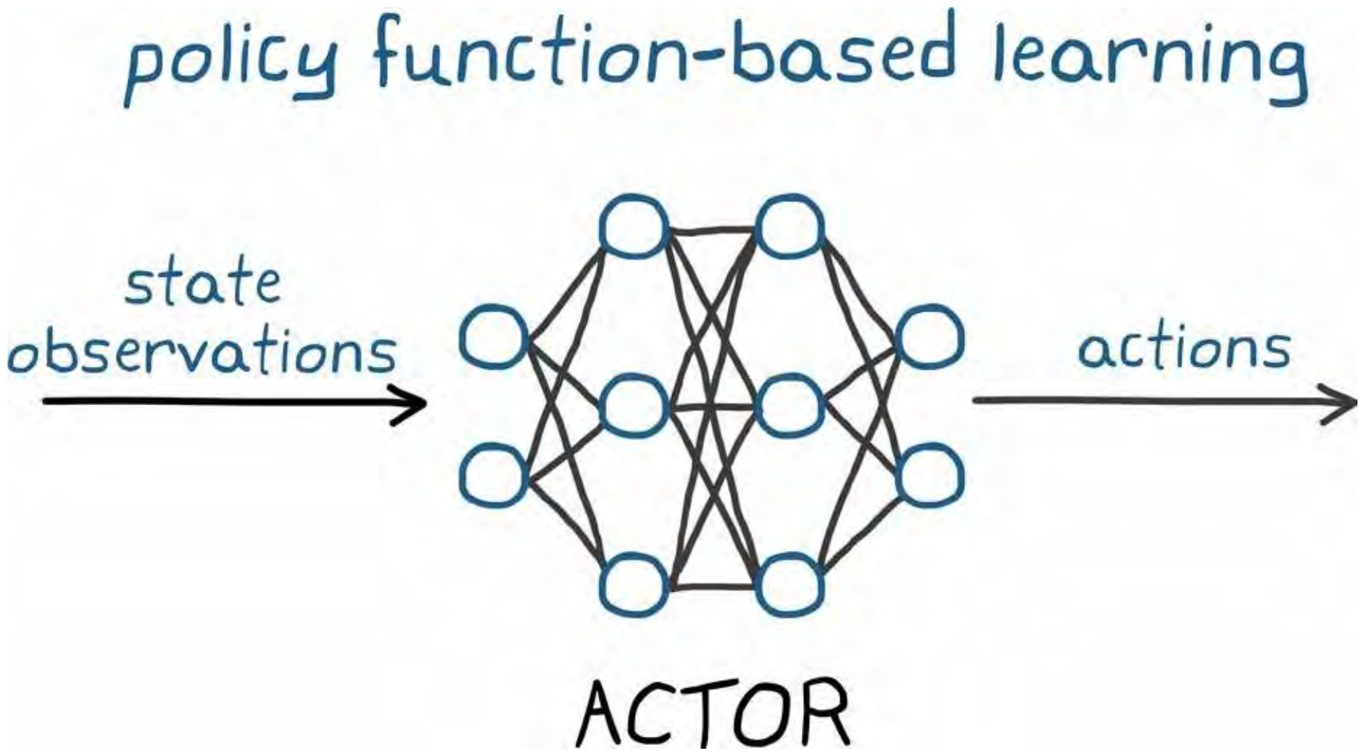
The next few pages will describe policy function–based, value function–based, and actor-critic approaches to reinforcement learning to highlight the differences in the policy structures. This will definitely be an oversimplification, but if you want a basic understanding of the ways policies can be structured,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

Policy Function–Based Learning

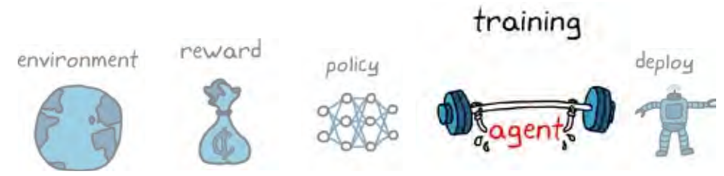


Policy function–based learning algorithms train a neural network that takes in the state observations and outputs actions. This neural network is the entire policy—hence the name *policy function–based algorithms*. The neural network is called the *actor* because it directly tells the agent which actions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

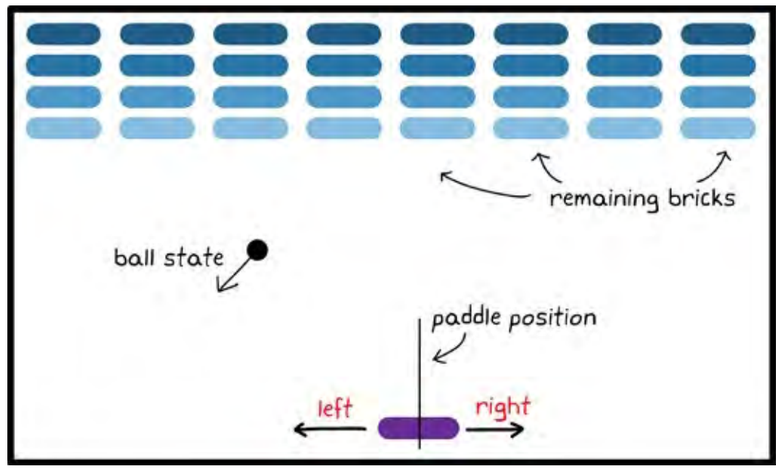
The question now is, how do we approach training this neural network? To get a general feel for how this is done, take a look at the Atari game Breakout.

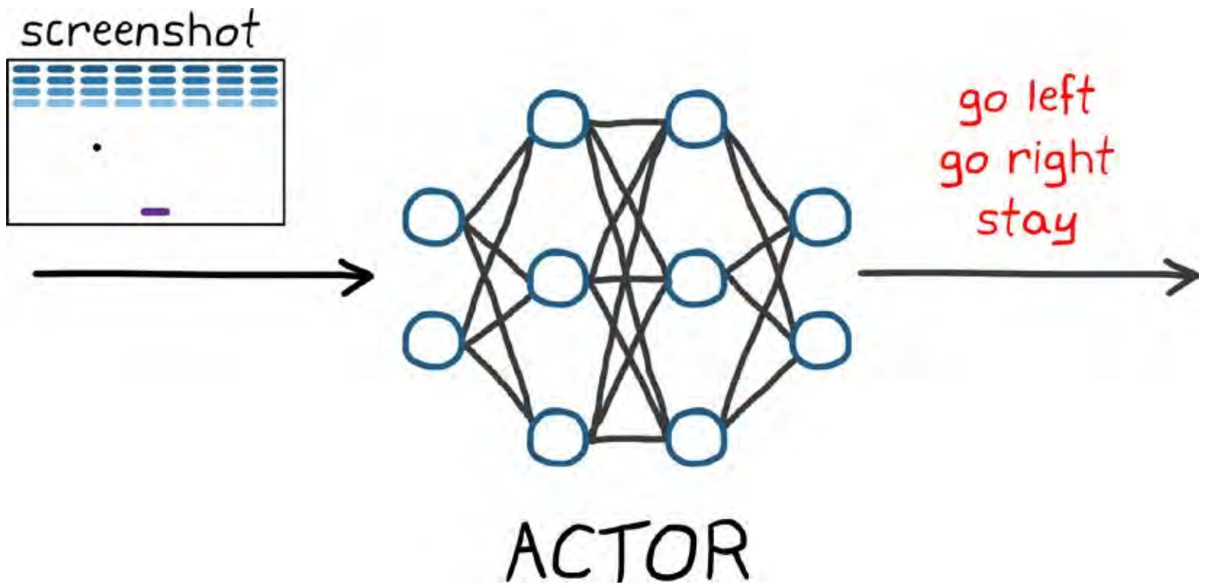


The Policy Approach to Learning Breakout



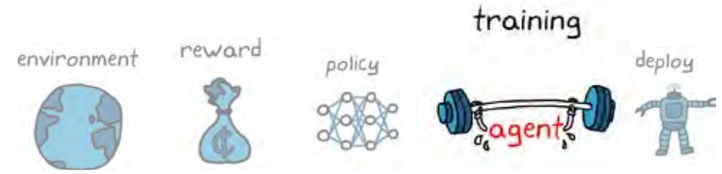
Breakout is a game in which you try to eliminate bricks using a paddle to direct a bouncing ball. The game has three actions, move the paddle left, right, or not at all, and a near-continuous state space that includes the position of the paddle, the position and velocity of the ball, and the location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



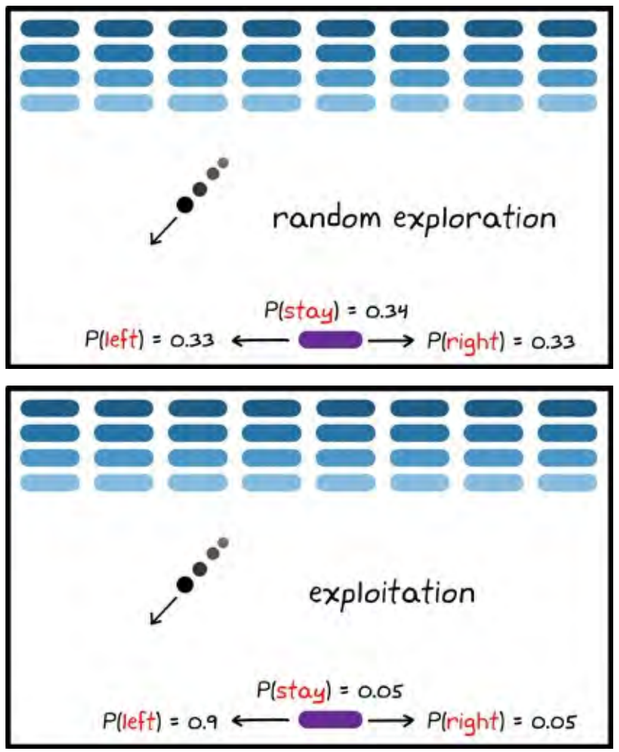


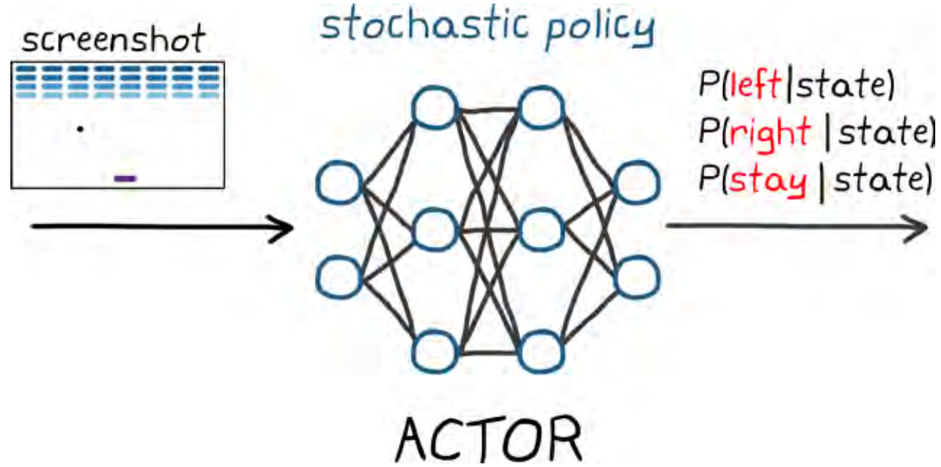
In this example, the inputs into the actor network are the states of the paddle, ball, and blocks. The outputs are nodes representing the actions: left, right, and stay. Rather than calculating the states manually and feeding them into the network, you can input a screen shot of the game and let the 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.



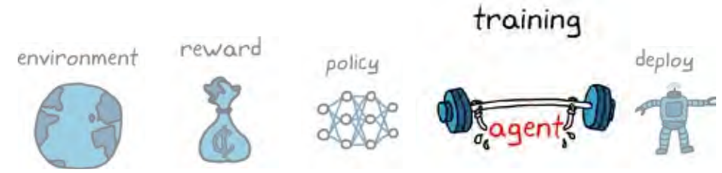


Should the agent exploit the environment by choosing the actions that collect the most rewards that it already knows about, or should it choose actions that explore parts of the environment that are still unknown?

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.

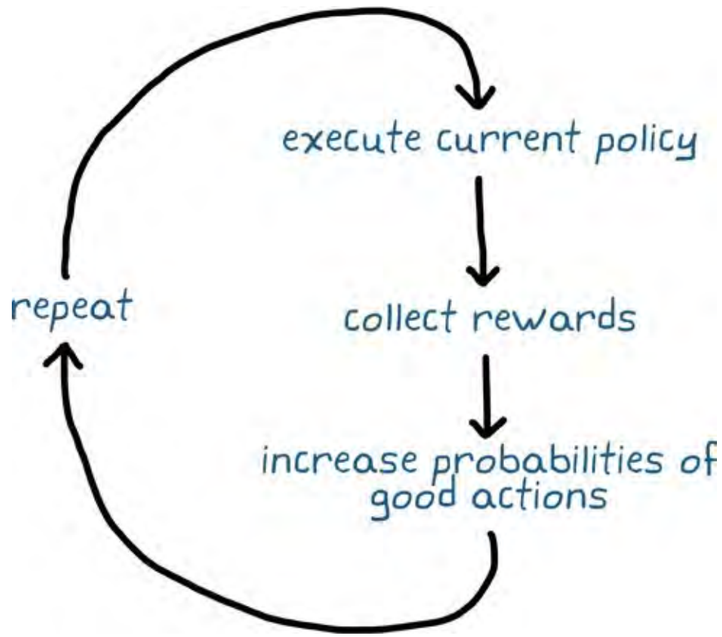
Over time, the agent will nudge these probabilities in the direction that produces the most reward. Eventually, the most advantageous action for every state will have such a high probability that the agent always takes that action.

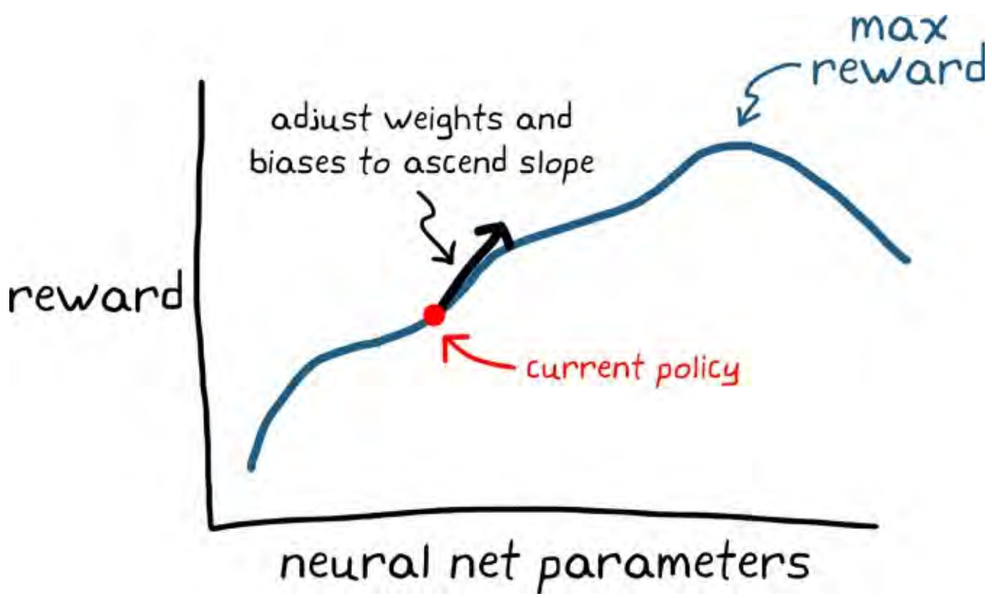
Policy Gradient Methods



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.

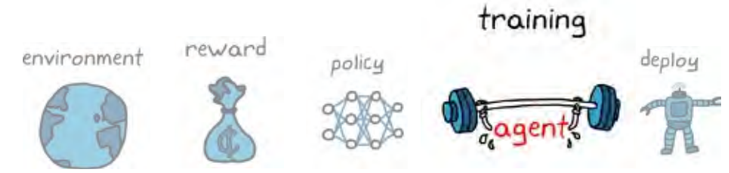
If the paddle went left, missing the ball and causing a negative reward, then change the neural network to increase the probability of moving the paddle right next time the agent is in that state.



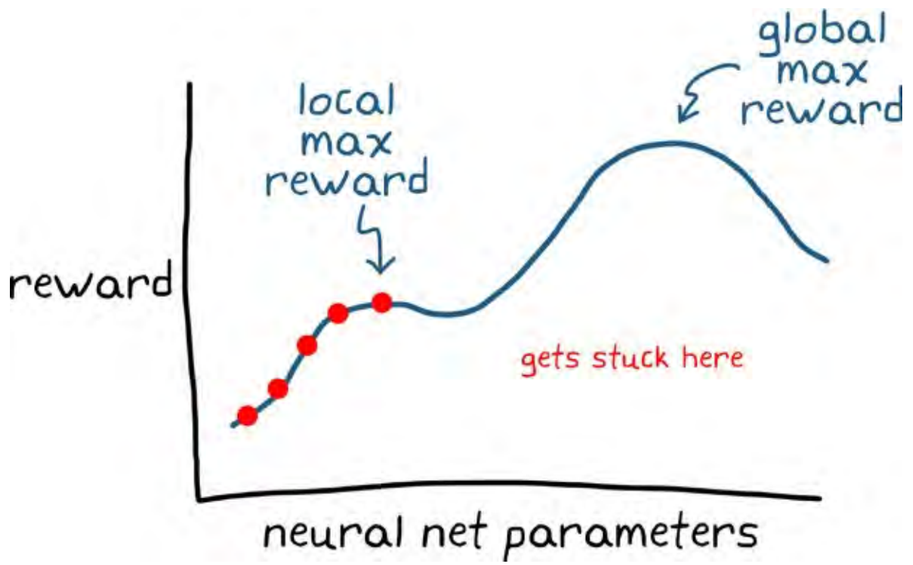


You take the derivative of each weight and bias in the network with respect to reward, and adjust them in the direction of a positive reward increase. In this way, the learning algorithm is moving the weights and biases of the network to ascend up the reward slope. This is why the term *gradient* is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

The Downside of Policy Gradient Methods

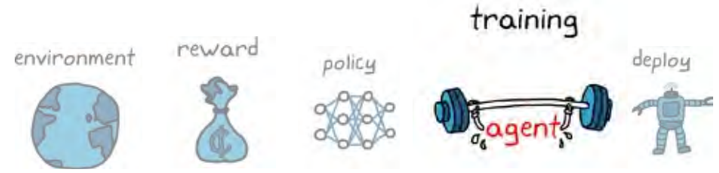


One of the downsides of policy gradient methods is that the naive approach of just following the direction of steepest ascent can converge on a local maxima rather than global. Policy gradient methods can also converge slowly due to their sensitivity to noisy measurements, which happens, for example,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

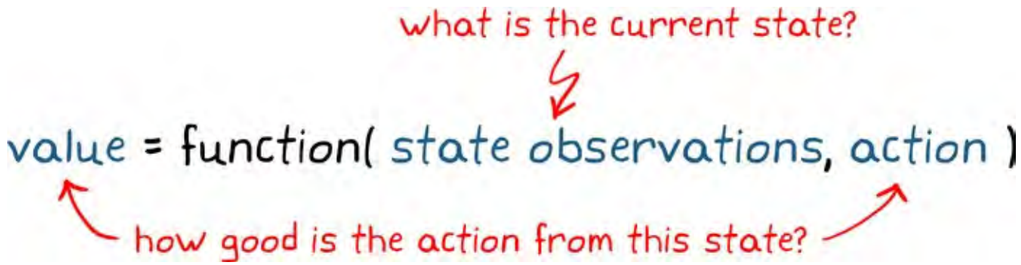


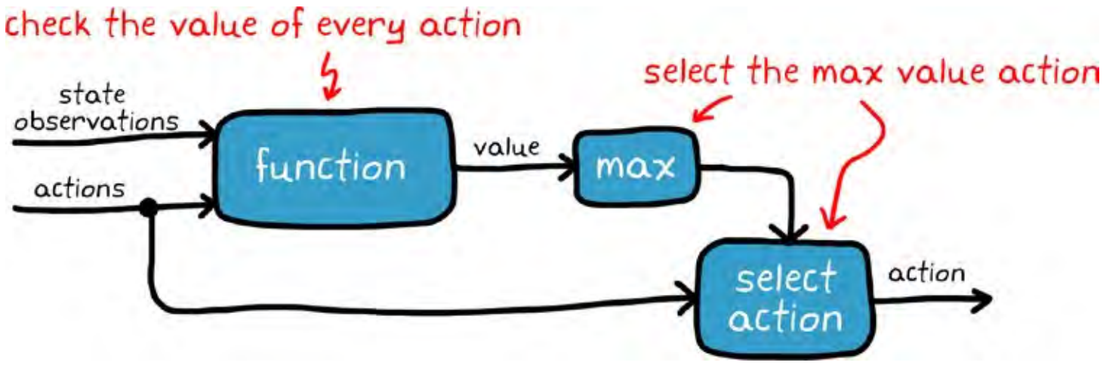
For example, in Breakout the agent might make a lot of quick left and right paddle movements as the paddle ultimately works its way across the field to strike the ball and receive a reward. The agent wouldn't know if every single one of those actions was actually required to get that reward, so the policy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

Value Function–Based Learning



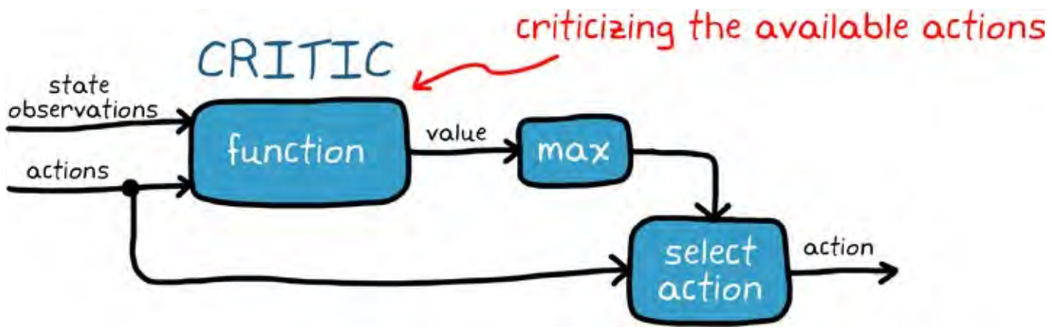
With a value function–based agent, a function would take in the state and one of the possible actions from that state, and output the value of taking that action.



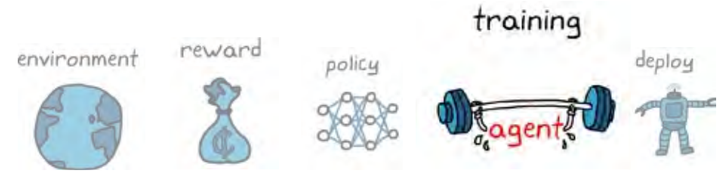


This function alone is not enough to represent the policy since it outputs a value and the policy needs to output an action. Therefore, the policy would be to use this function to check the value of every possible action from a given state and choose the action with the highest value.

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

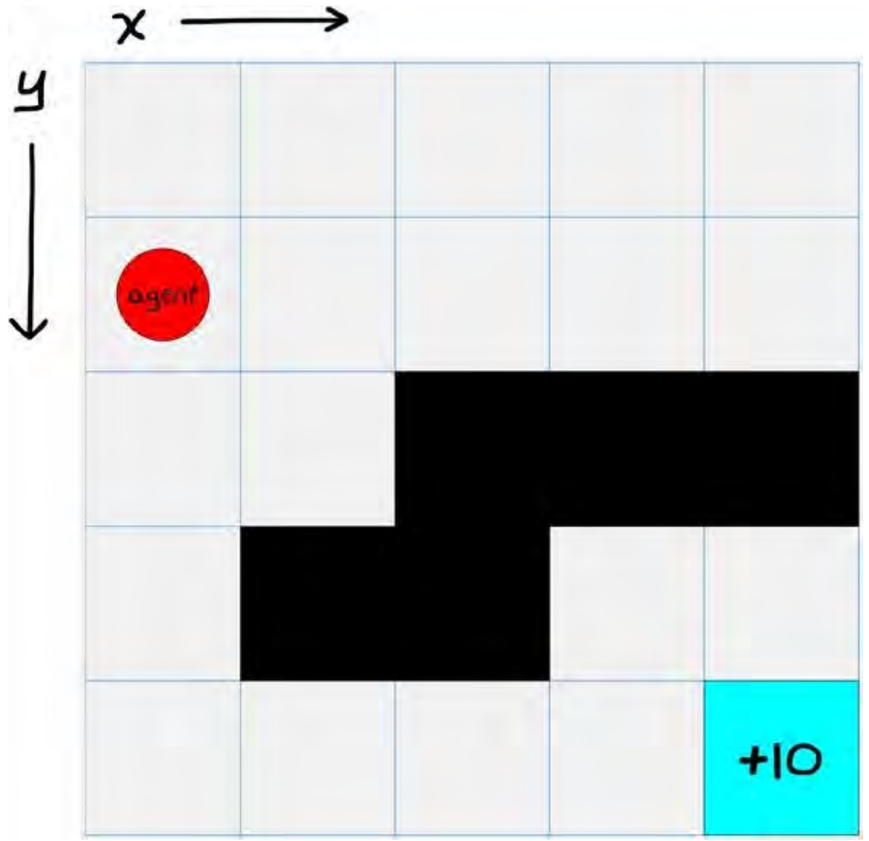


To see how this would work in practice, here's an example using the Grid World environment.

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.

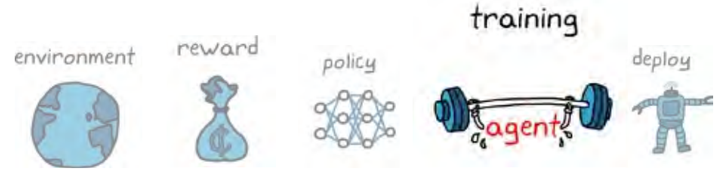
If the agent tries to move off the grid or into one of the black obstacles, then the agent doesn't move into the new state but the -1 reward is still received. In this way, the agent is penalized for essentially running into a wall and it doesn't make any physical progress for that effort.

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.



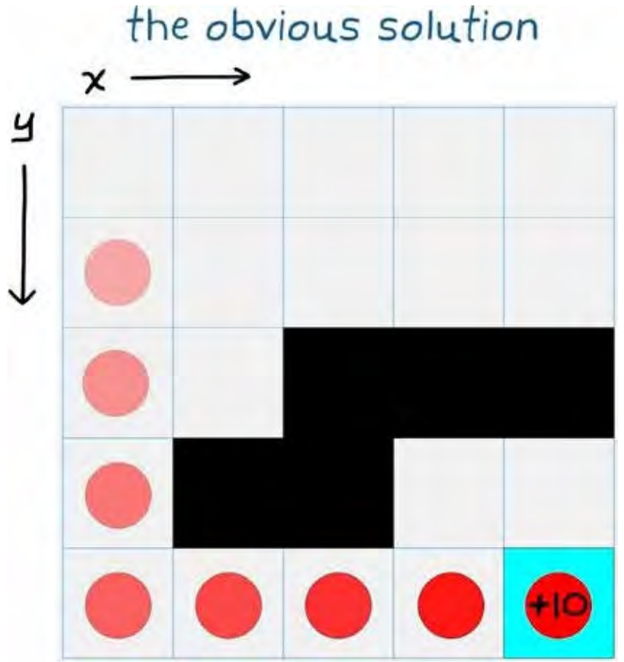
» *See how to solve a Grid World environment in MATLAB*

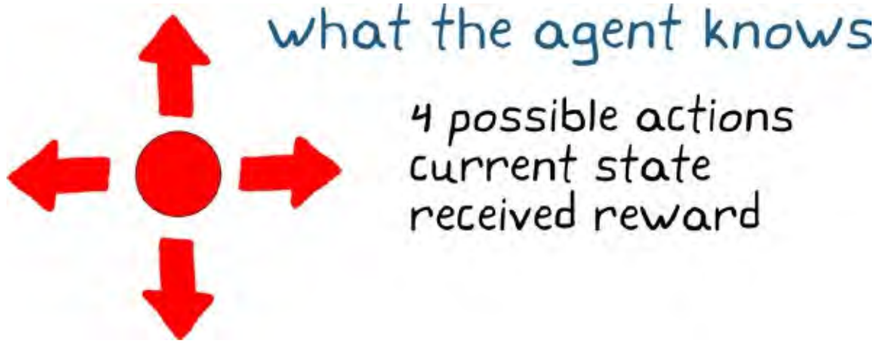
Value Functions and Grid World *Continued*



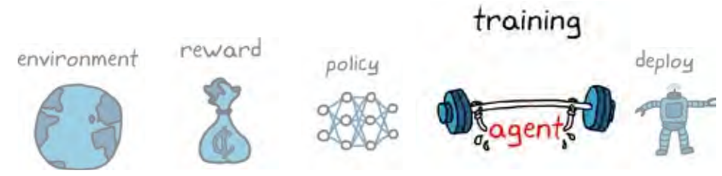
It might seem easy to determine exactly which route to take to get to the reward.

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.



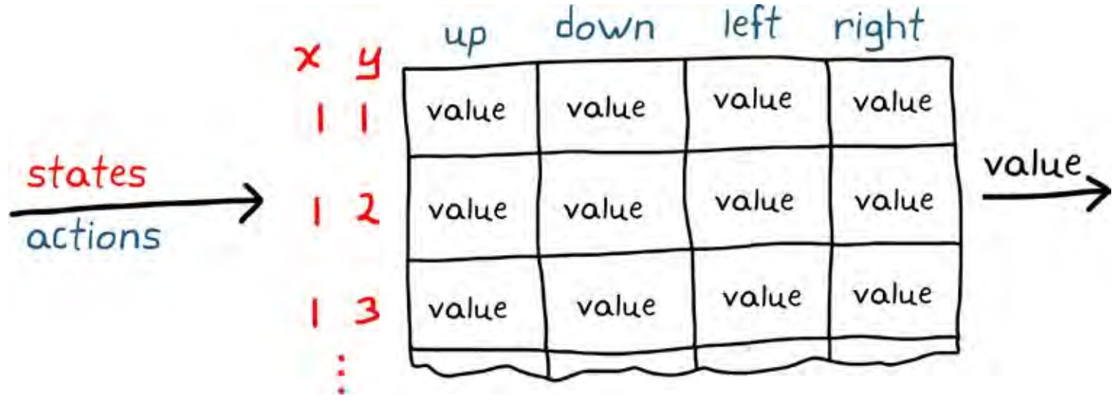


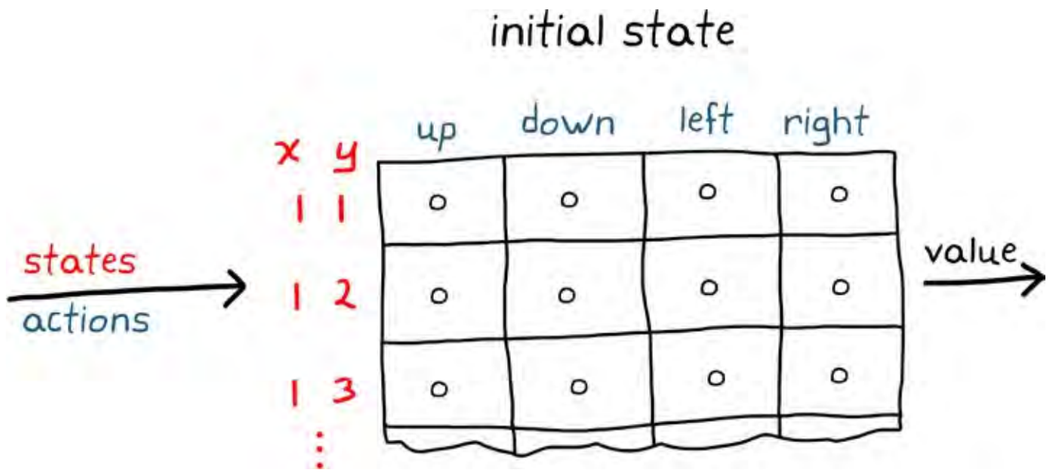
Solving Grid World with Q-Tables



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.

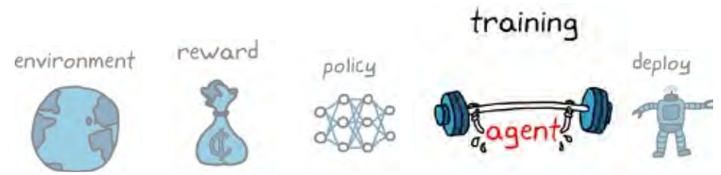




With Q-learning, you can start by initializing the table to zeros, so all actions look the same to the agent. After the agent takes a random action, it gets to a new state and collects the reward from the environment.

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.

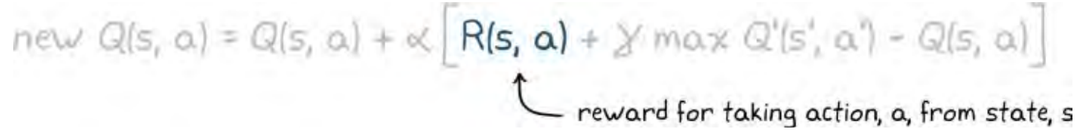
The Bellman Equation



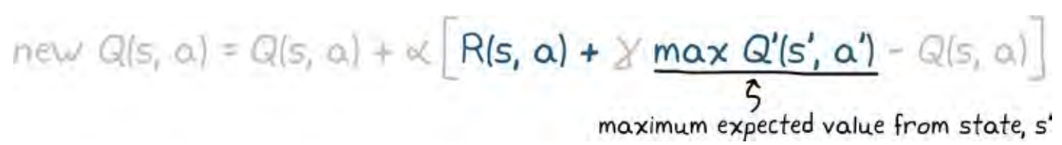
$$\text{new } Q(s, a) = Q(s, a) + \alpha\left[R(s, a) + \gamma \cdot \max Q'(s', a') - Q(s, a)\right]$$

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.



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.



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).

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).

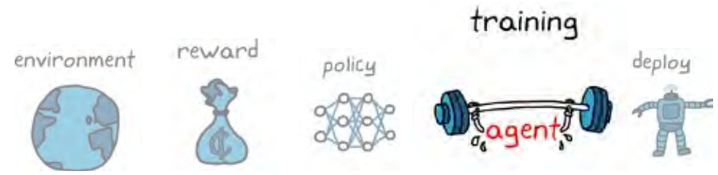
$$\text{new } Q(s,a) = Q(s,a) + \underbrace{\alpha}_{\text{error is multiplied by learning rate}}\left[R(s,a) + \gamma \cdot \max Q'(s',a') - Q(s,a)\right]$$

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

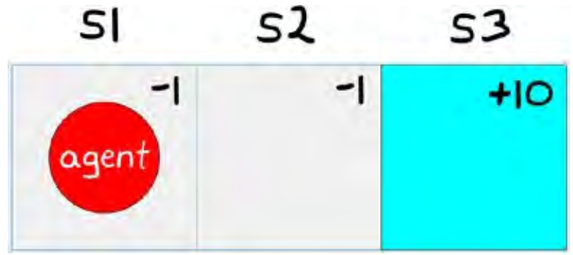
$$\text{new } Q(s,a) = \underbrace{Q(s,a)}_{\text{delta value is added to old estimate}} + \alpha\left[R(s,a) + \gamma \cdot \max Q'(s',a') - Q(s,a)\right]$$

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: 0, 0; S3: 0, 0	right (random)	-1	S1: Left 0, Right -1; S2: 0, 0; S3: 0, 0
1	2	S2	S1: Left 0, Right 0; S2: 0, 0; S3: 0, 0	right (random)	+10	S1: Left 0, Right -1; S2: 0, 10; S3: 0, 0

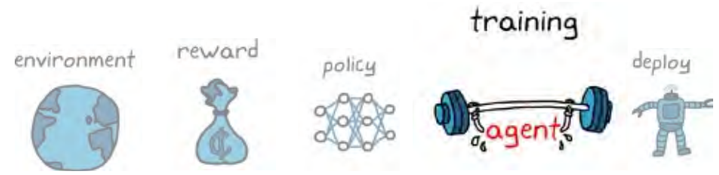
Bellman equation: $0 + 1 \cdot [-1 + 0.9 \cdot 0 - 0] = -1$

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

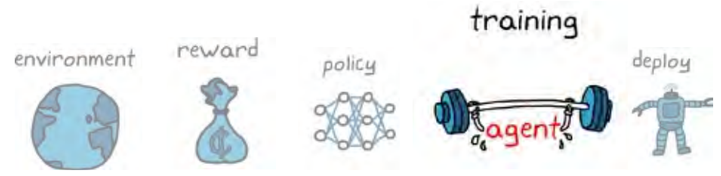
	S1	S2	S3
Left	-1	0	0
Right	8	10	0

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

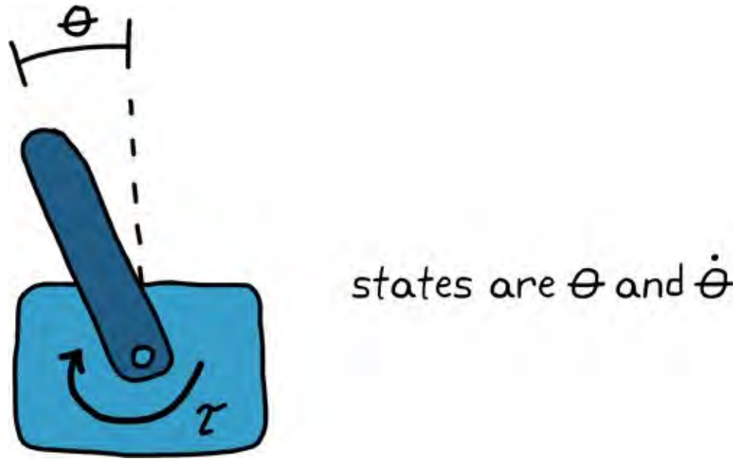
End of episode

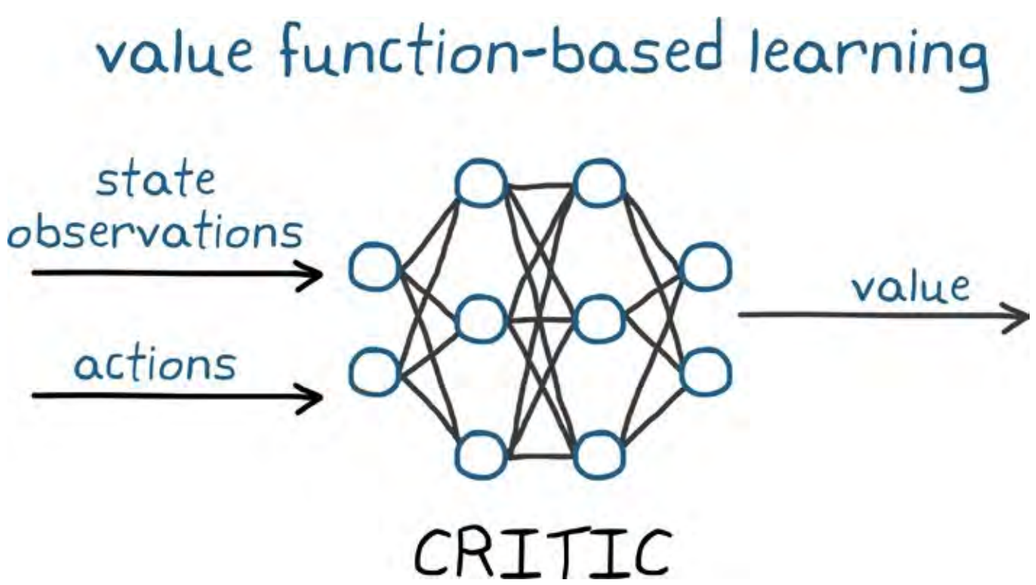
Within just four actions, the agent has already settled on a Q-table that produces the optimal policy; in state S1, it will take a right since the value 8 is higher than -1, and in state S2, it will take a right again since the value 10 is higher than 0. What's interesting about this result is that the 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.

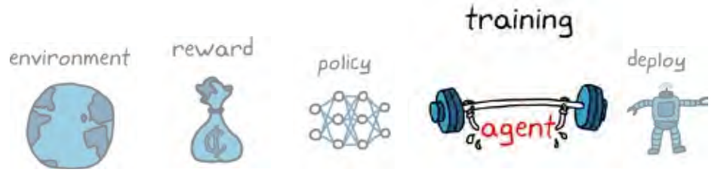




The value function (the critic) is represented with a neural network. The idea is the same as with a table: You input the state observations and an action, the neural network returns the value of that state/action pair, and the policy is to choose the action with the highest value.

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



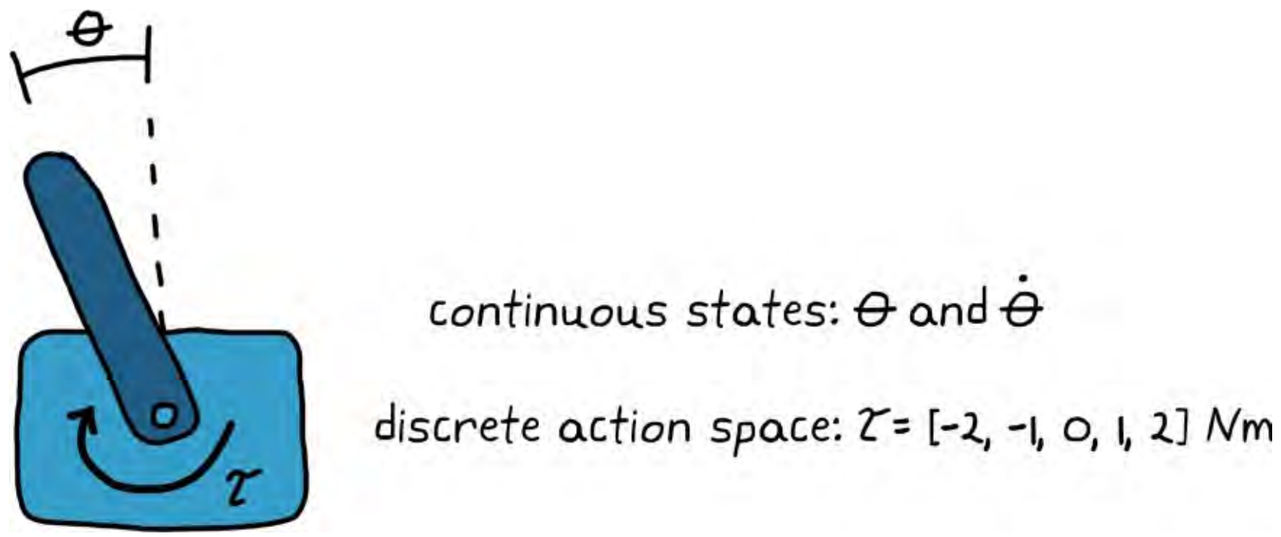
You can use a neural network to define the value function for continuous state spaces. If the inverted pendulum has a discrete action space, you can feed discrete actions into your critic network one at a time.

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.

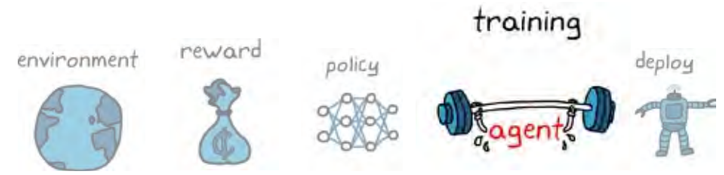
So what can you do?

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*.

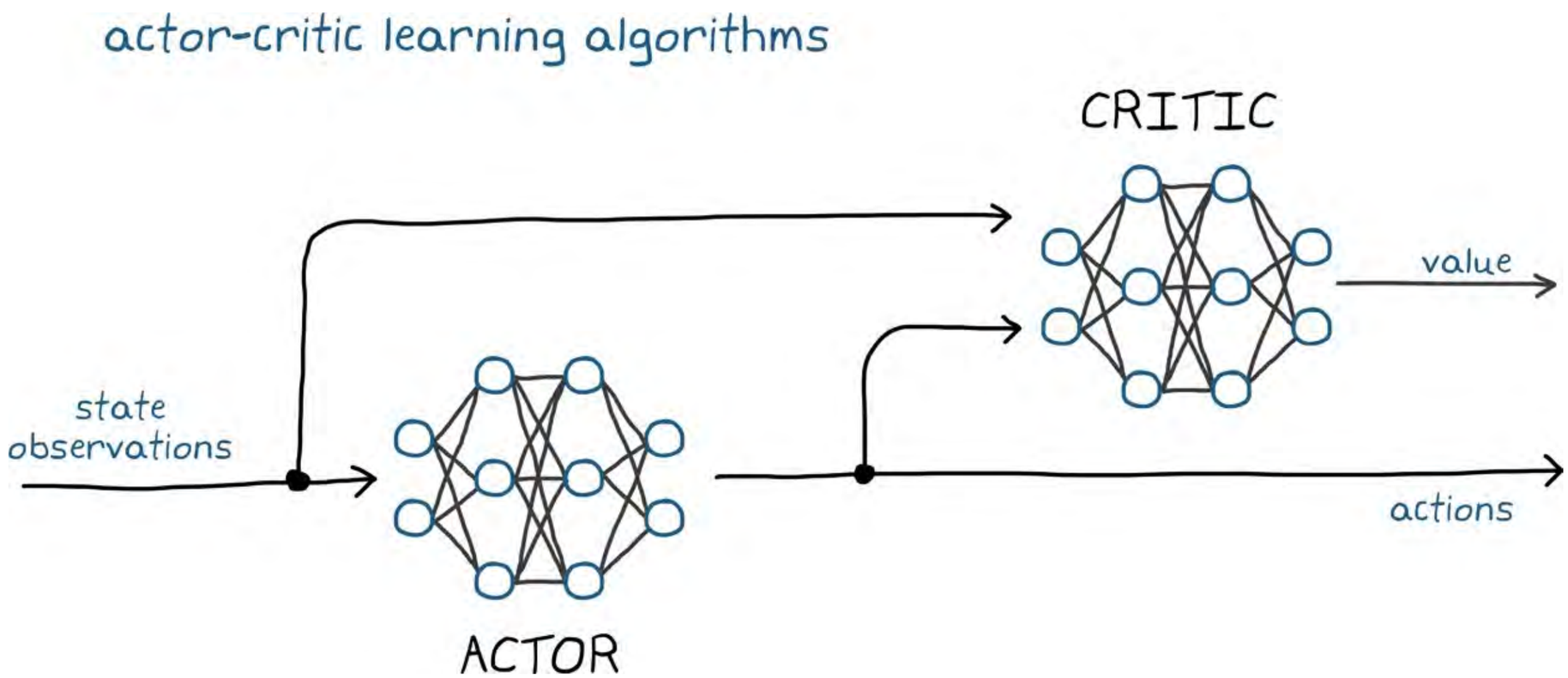
» *See how to train an actor-critic agent to balance an inverted pendulum in MATLAB*



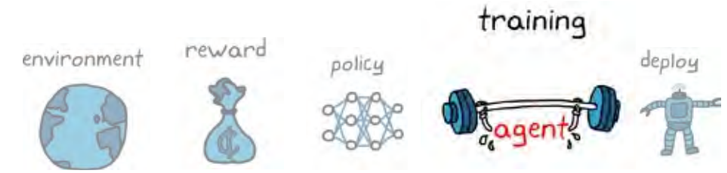
Actor-Critic Methods



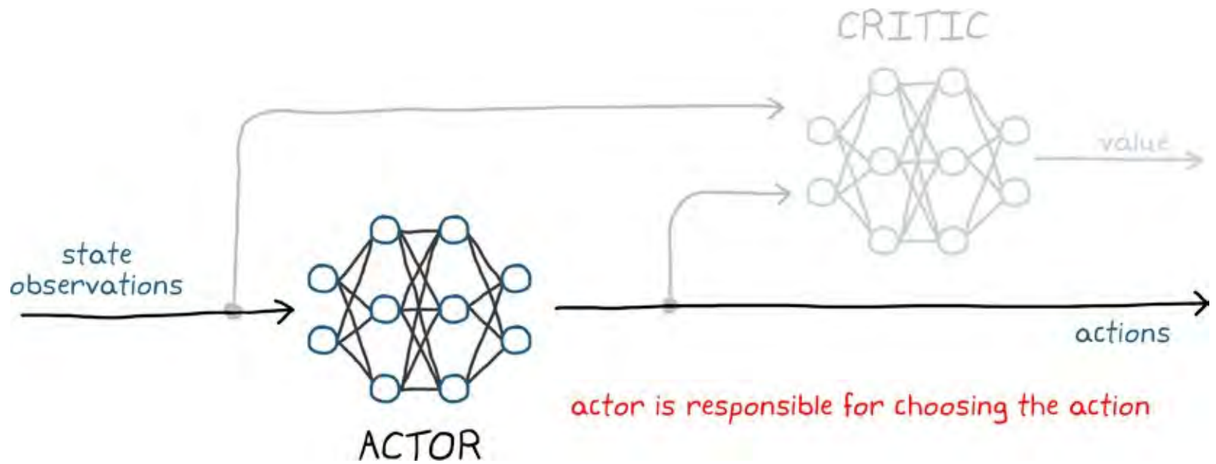
The *actor* is a network that is trying to take what it thinks is the best action given the current state, as seen with the policy function method. The *critic* is a second network that is trying to estimate the value of the state and the action that the actor took, as seen with the value function method. 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.

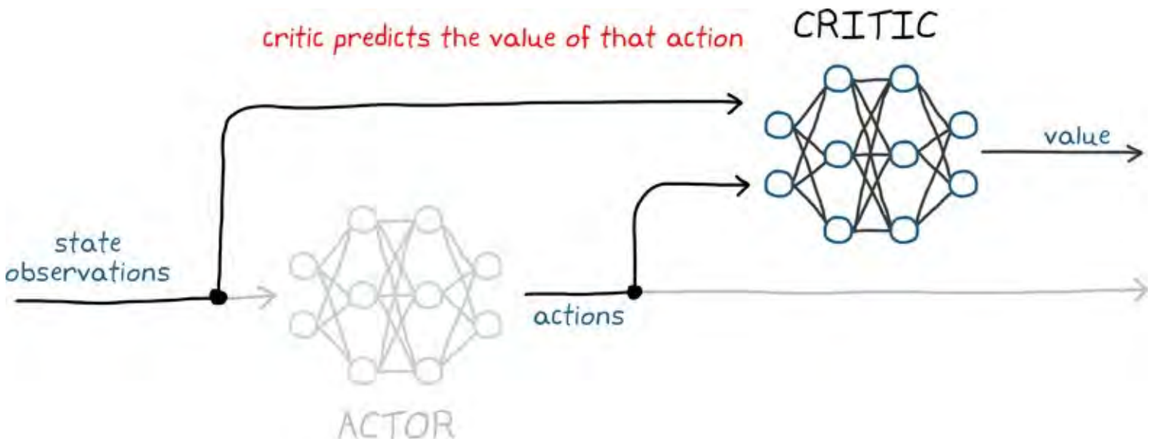


The Actor-Critic Learning Cycle



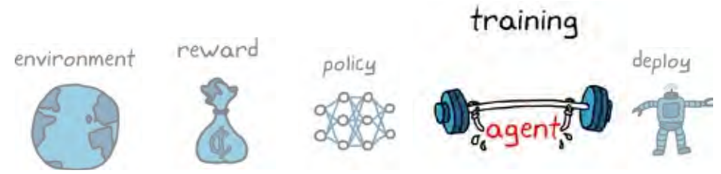
The actor chooses an action in the same way that a policy function algorithm would, and it is applied to the environment.



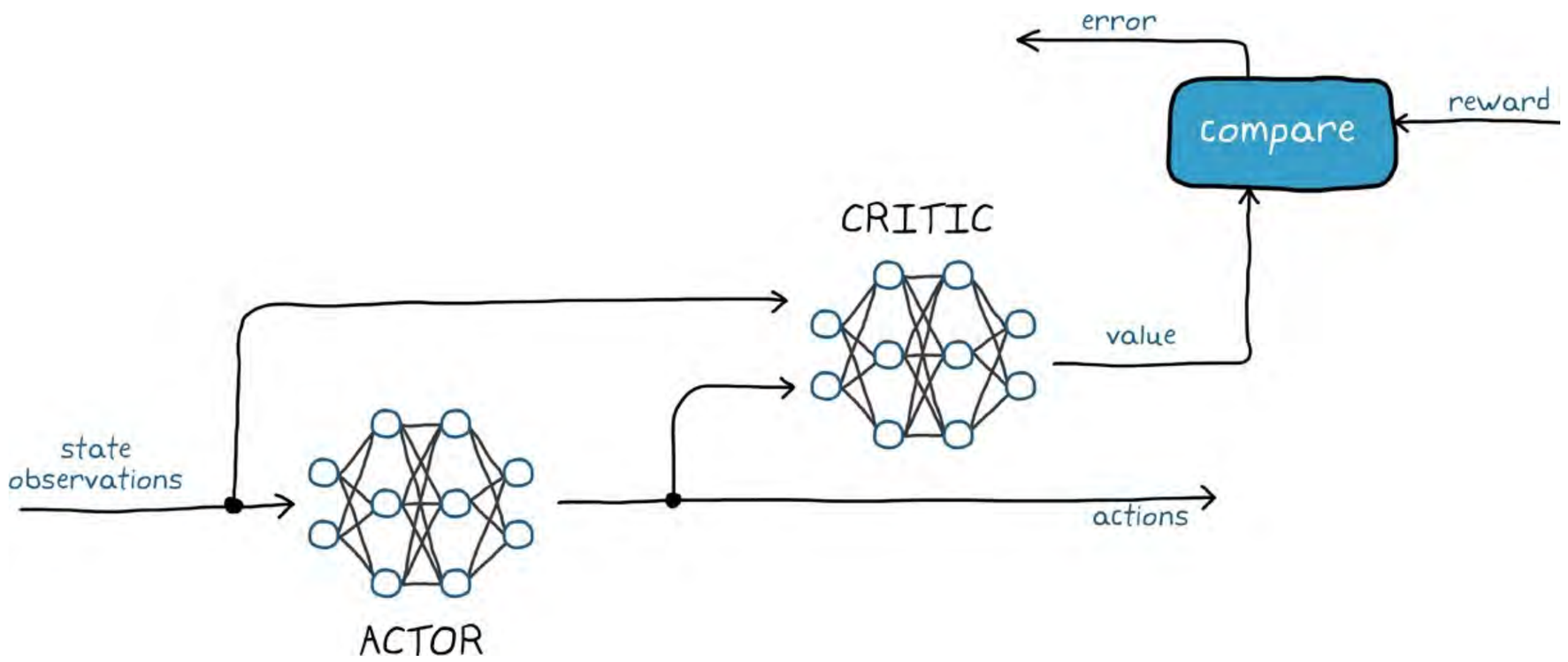


The critic makes a prediction of what the value of that action is for the current state and action pair.

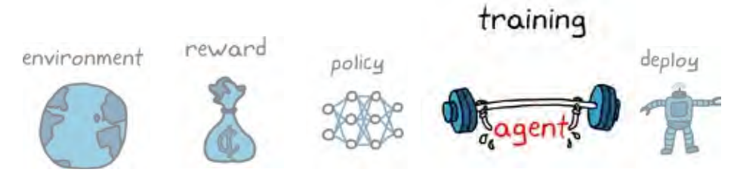
The Actor-Critic Learning Cycle *Continued*



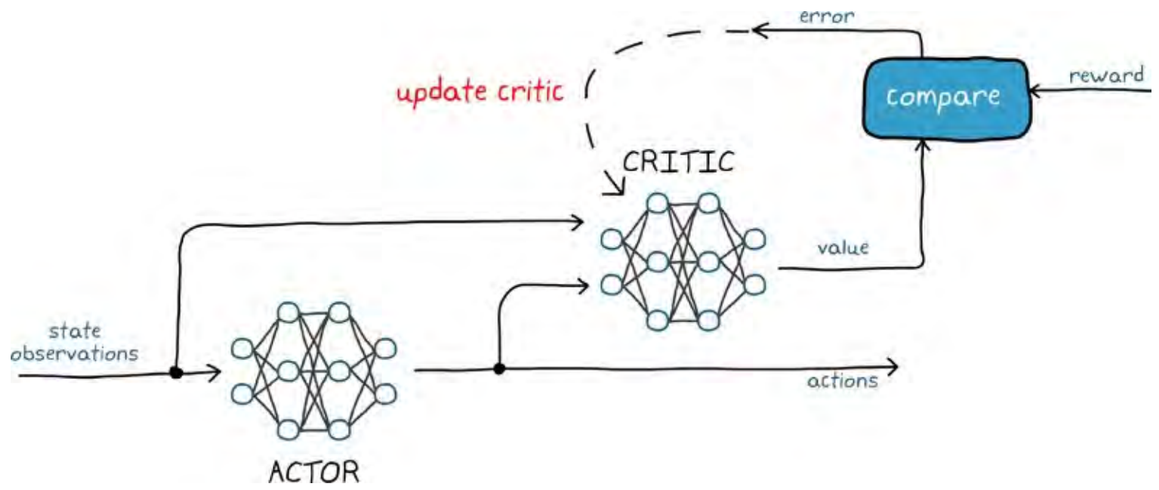
The critic then uses the reward from the environment to determine the accuracy of its value prediction. The error is the difference between the new estimated value of the previous state and the old value of the previous state from the critic network. The new estimated value is based on the received reward 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.

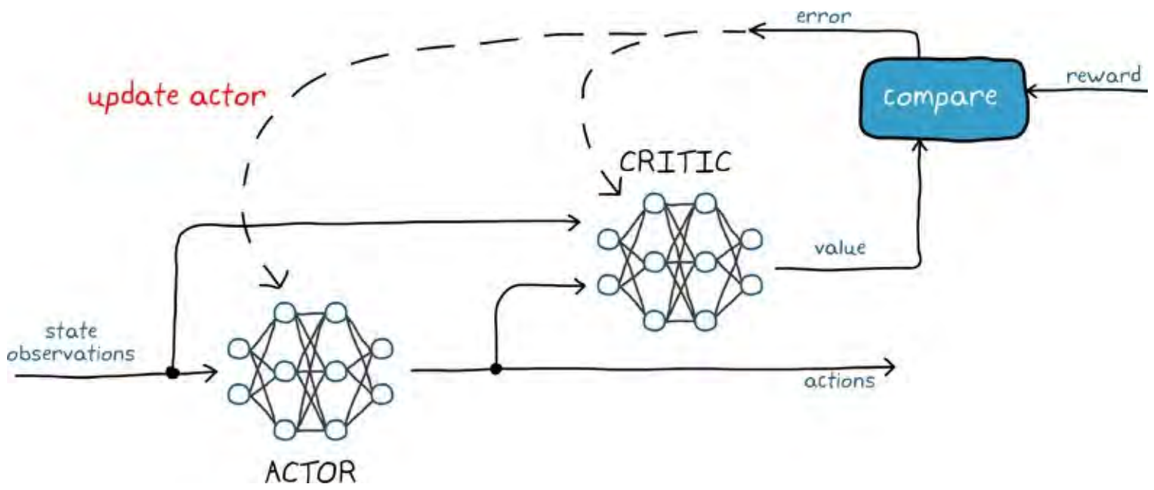


The Actor-Critic Learning Cycle *Continued*



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.

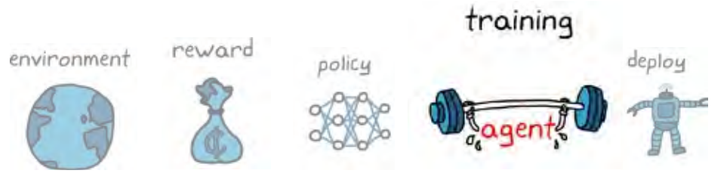




The actor also updates itself with the response from the critic so that it can adjust its probabilities of taking that action again in the future.

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

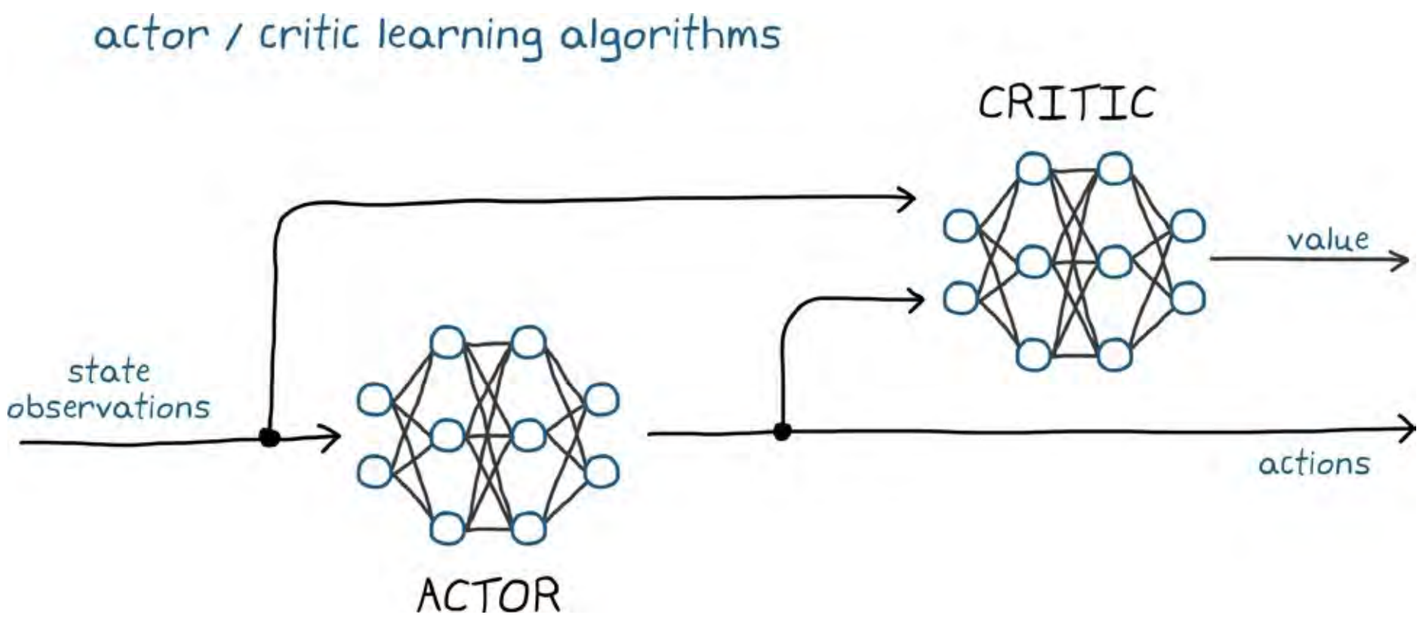


A lot of different types of learning algorithms use an actor-critic policy; this ebook generalizes these concepts to remain algorithm agnostic.

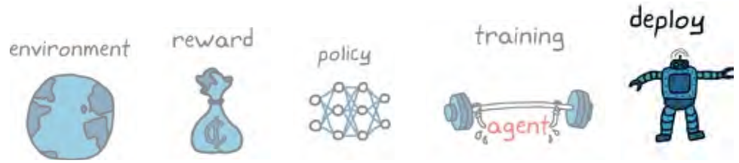
The actor and critic are neural networks that try to learn the optimal behavior. The actor is learning the correct actions to take using feedback from the critic to know which actions are good and bad, and the critic is learning the value function from the received rewards so that it can properly criticize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

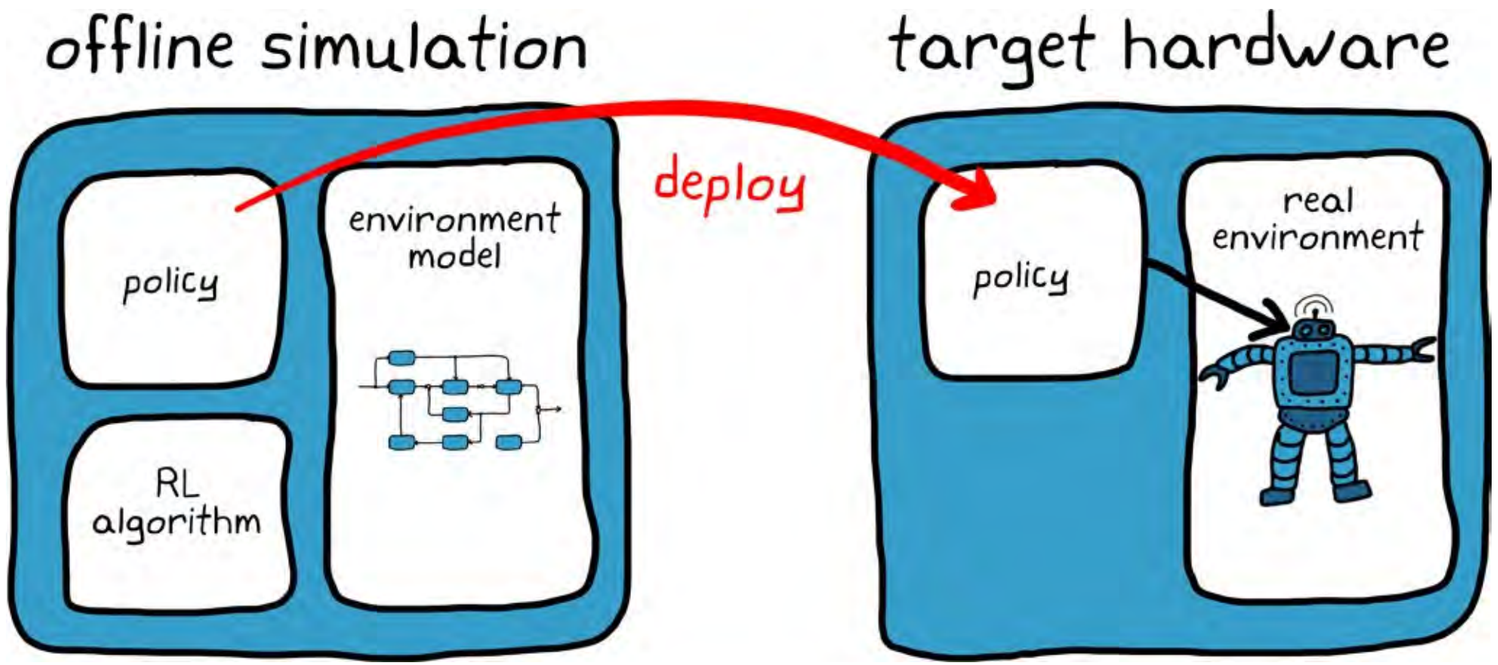


Policy Deployment

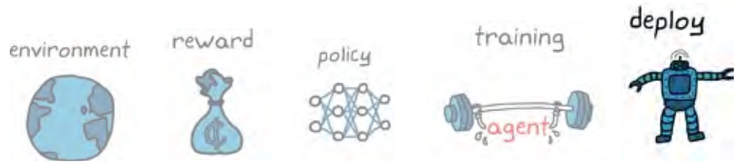


The last step in the reinforcement learning workflow is to deploy the policy.

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.



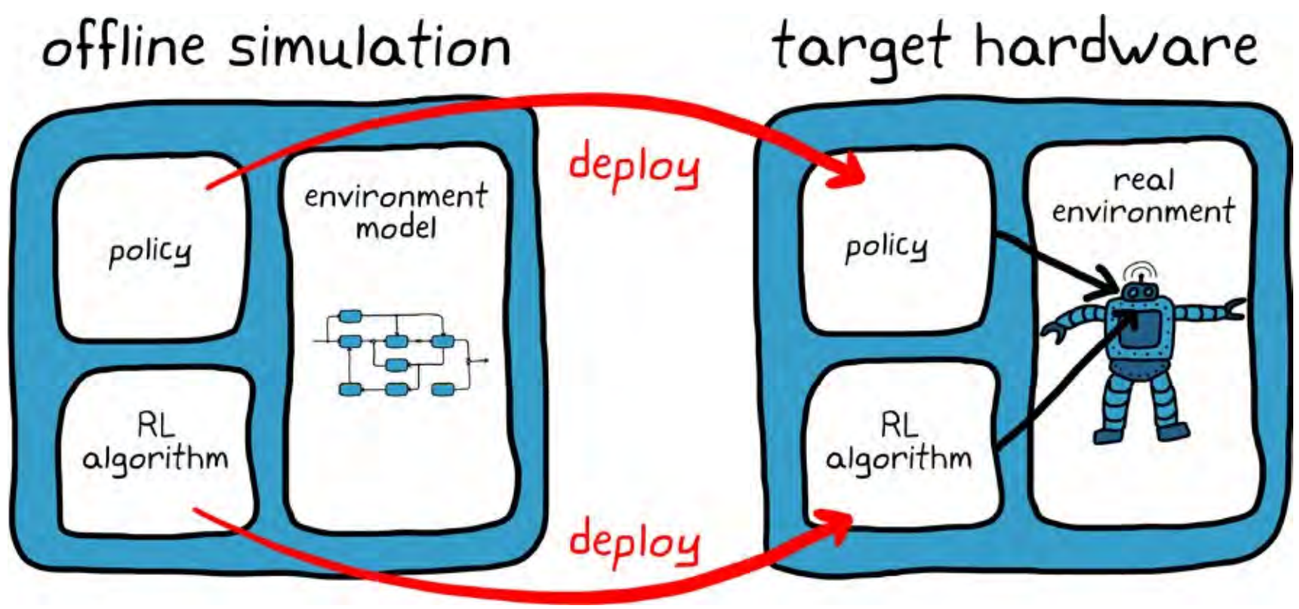
Deploying the Learning Algorithm



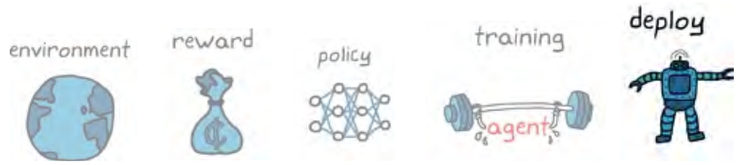
Even if the majority of learning is done offline with a simulated environment, it may be necessary to continue learning with the real physical hardware after deployment.

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.

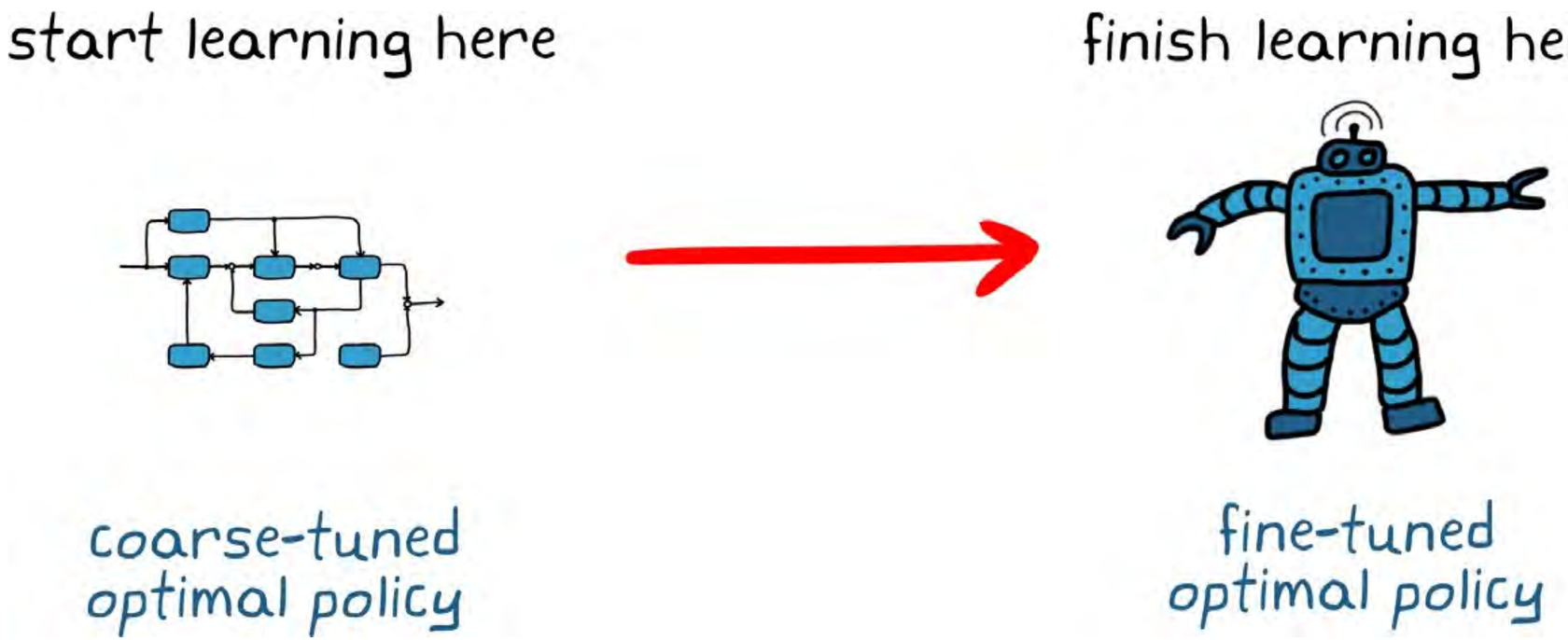
For these reasons, you deploy both the static policy and the learning algorithms to the target. With this setup, you have the option of executing the static policy (turn learning off) or continuing to update the policy (turn learning on).



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.

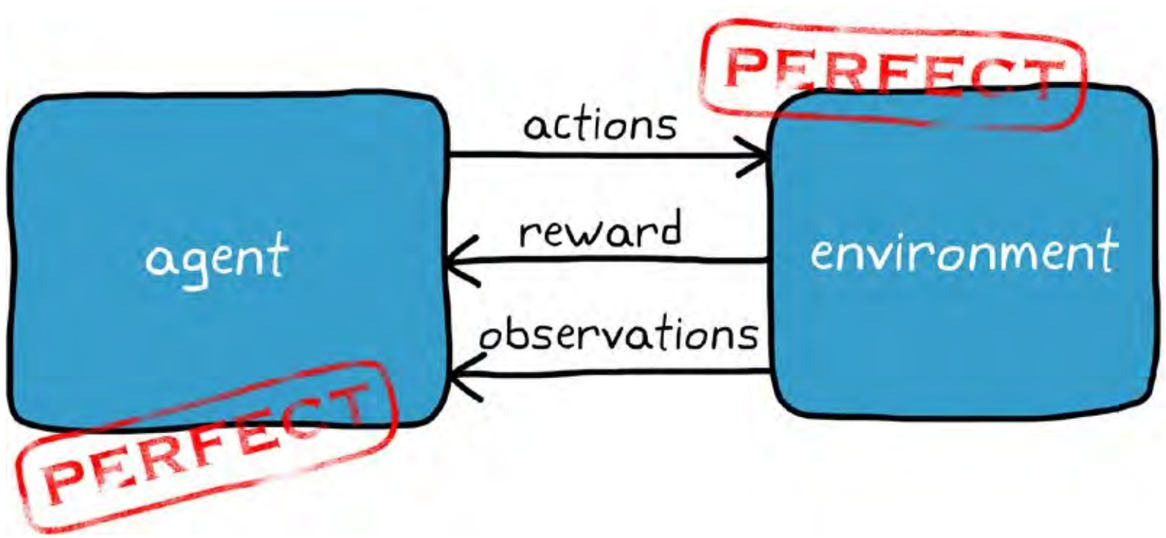


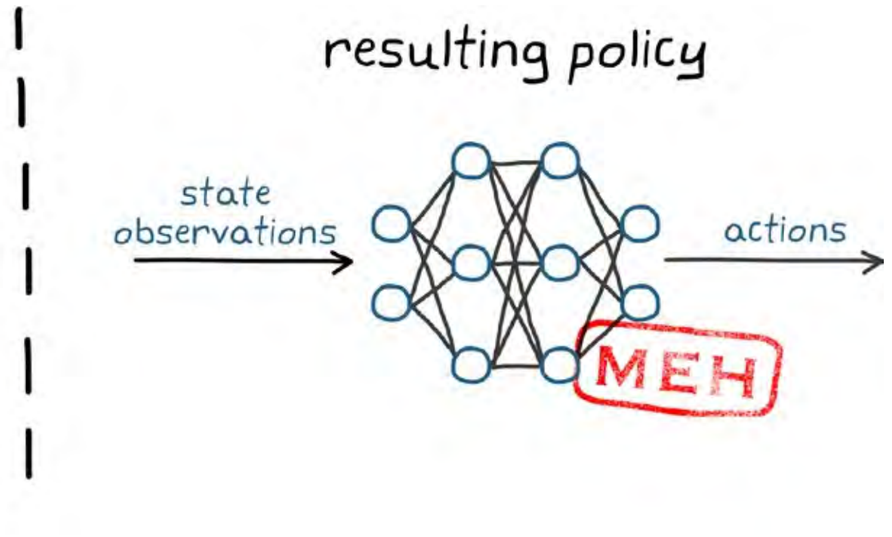
The Drawbacks of RL

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:

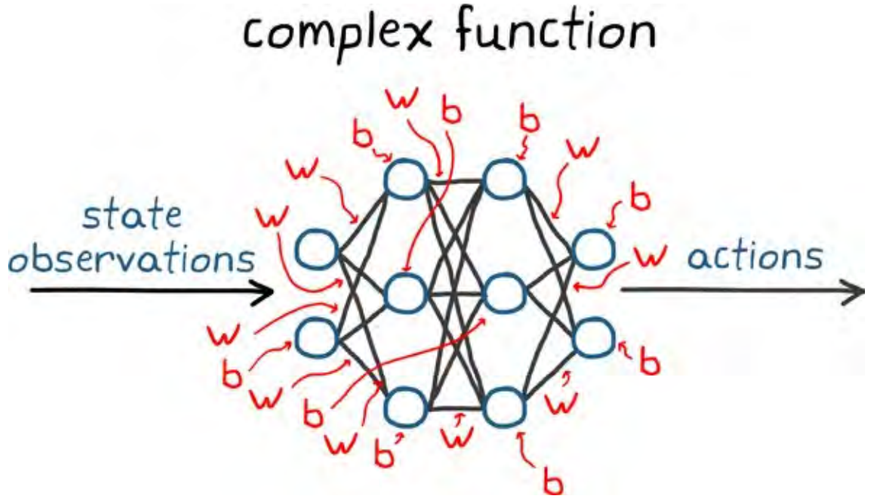
- How do you know the solution is going to work?
- Is there a way to manually adjust it if it's not quite perfect?

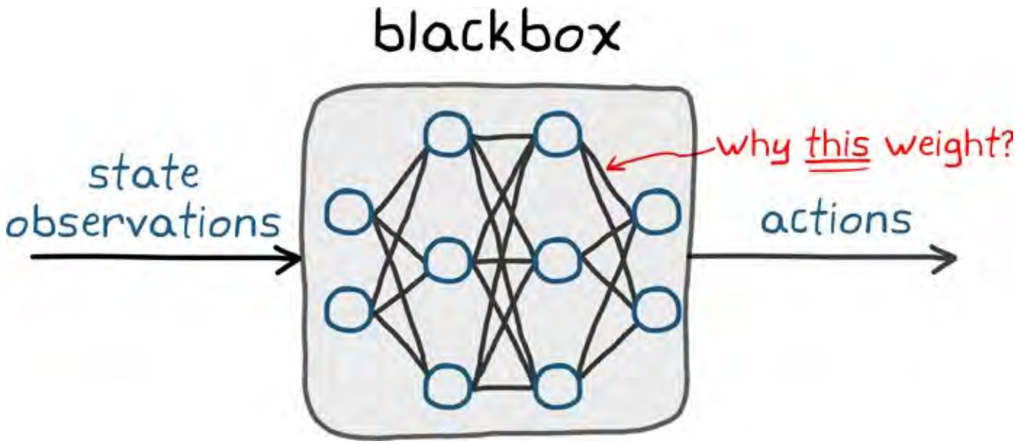




The Unexplainable Neural Network

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.





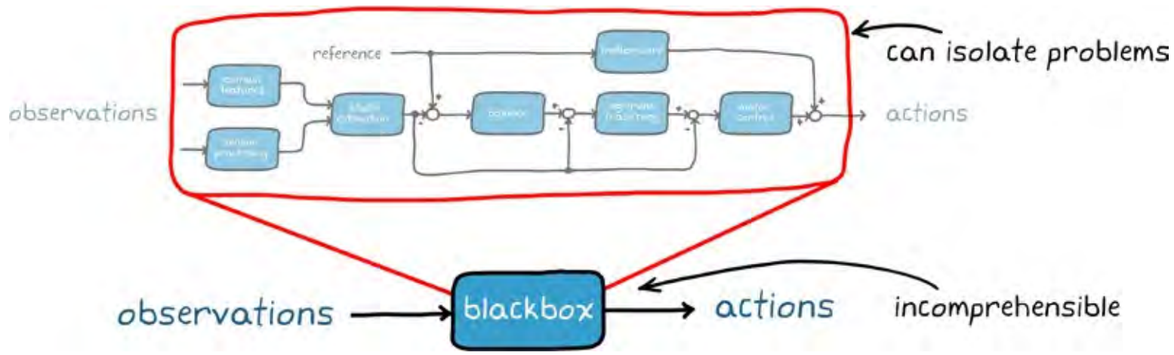
This function is a black box to the designer. You may have an intuitive sense of how this function operates and the hidden features that this network has identified, but you don't know the reason behind the value of any given weight or bias. So if the policy doesn't meet a specification or if the operating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

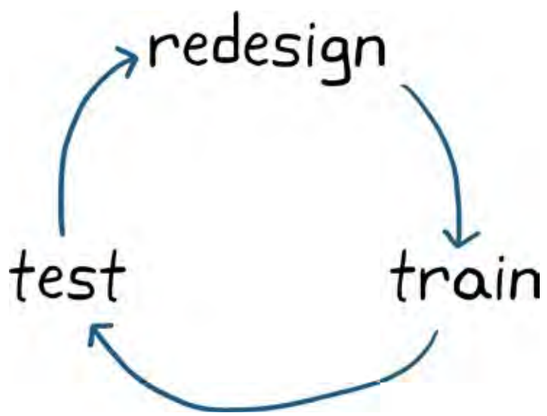
There is active research that is trying to push the concept of *explainable artificial intelligence*. This is the idea that you can set up your network so that it can be easily understood and audited by humans. At the moment, the majority of RL-generated policies are still categorized as a black box,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



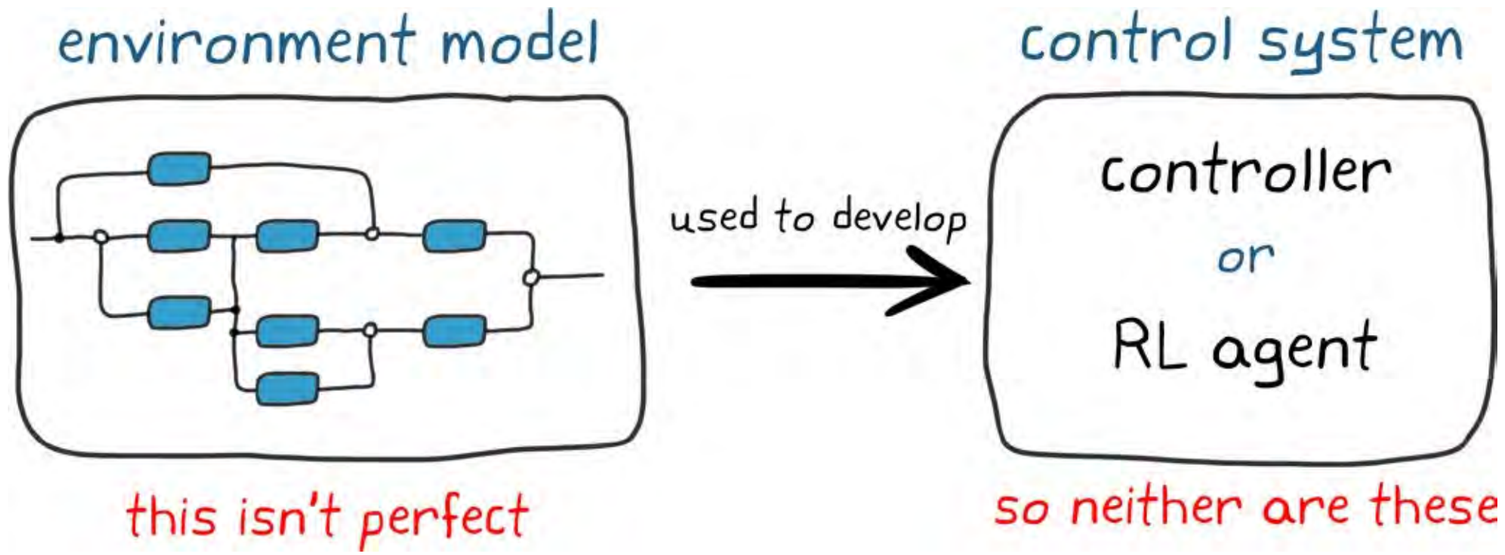


Isolating issues is difficult to do when the solution is a monolithic collection of neurons and weights and biases. So, if you end up with a policy that isn't quite right, rather than being able to fix the offending part of the policy, we have to redesign the agent or the environment model and then train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



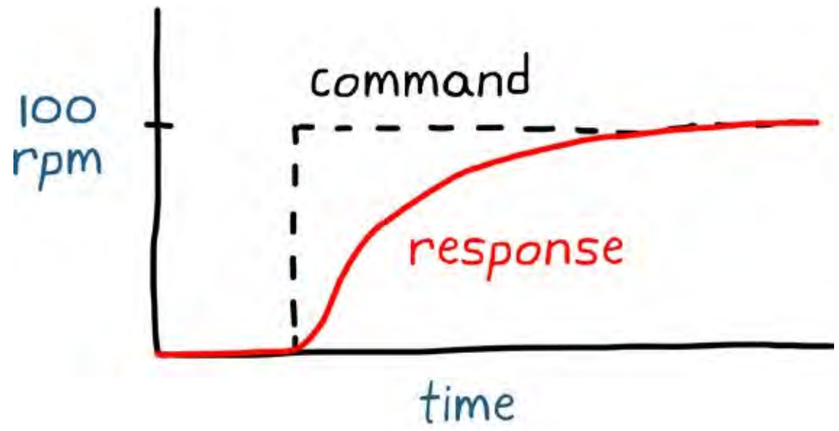
This is less of an issue if you use the model to design a traditional control system, since you can understand the functions and can tune the controllers. However, with a neural network policy, you don't have that luxury. As you can never build an absolutely realistic model, any agent you train with 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.

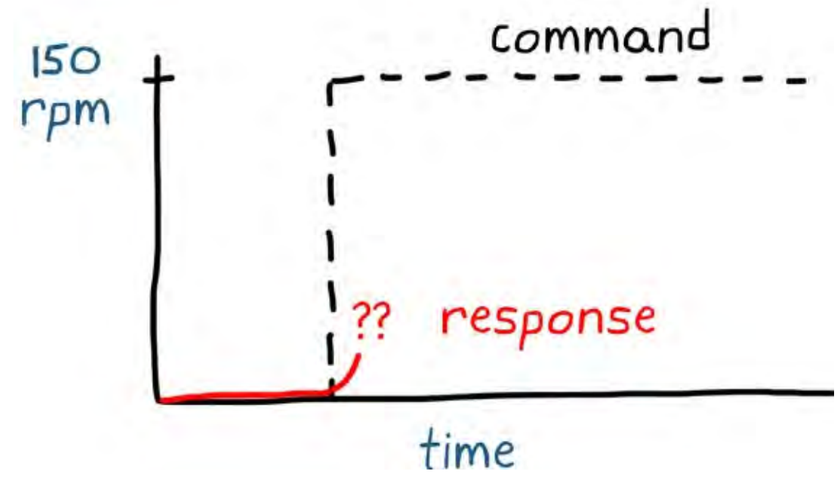
Verifying the Learned Policy

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.

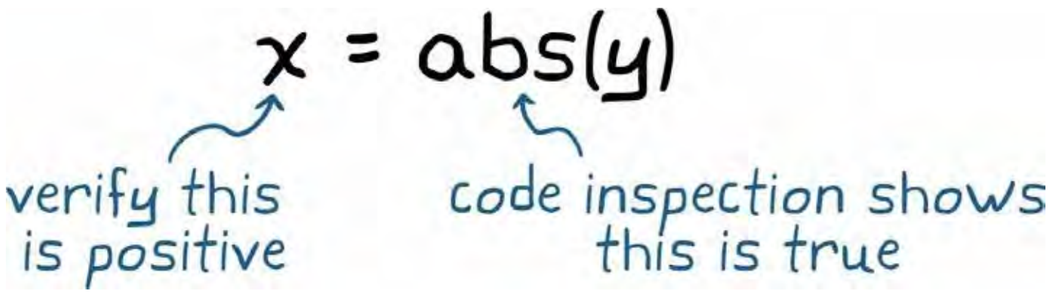
Having to run a few extra tests might not seem like a big deal, but you have to remember that one of the benefits of deep neural networks is that they can handle data from rich sensors, like images from a camera that have extremely large input spaces; think thousands of pixels that each can have a value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

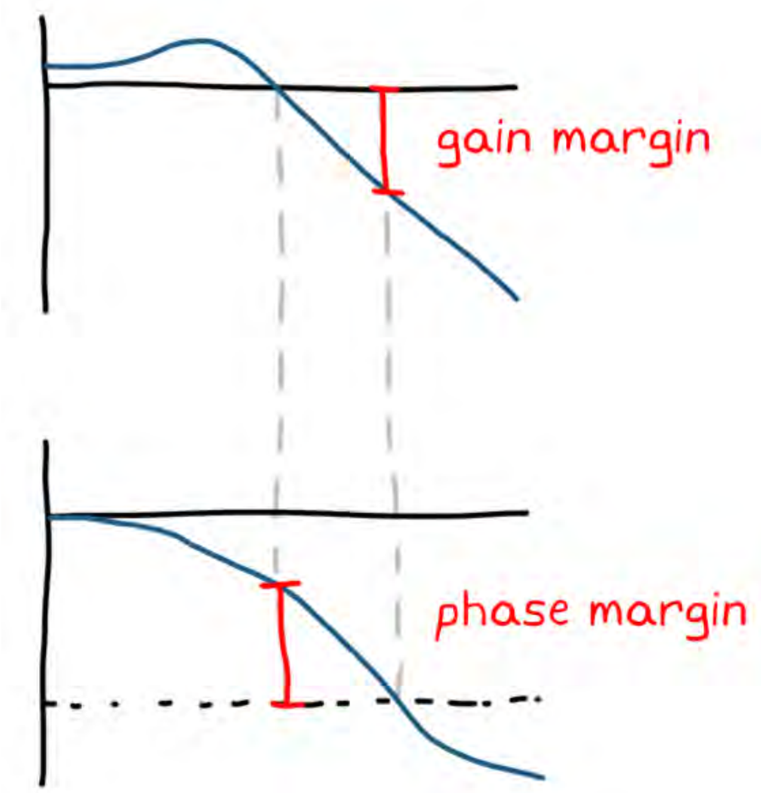




Formal Verification Methods

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.



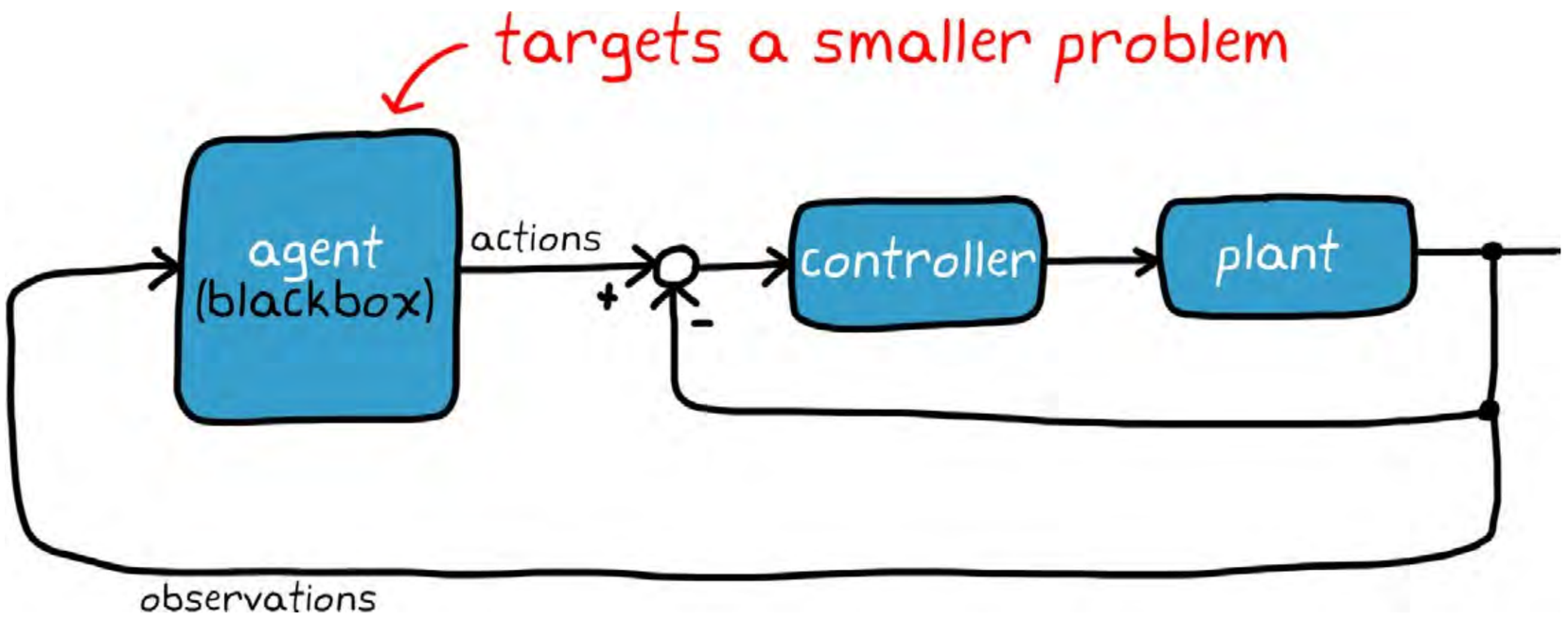


For neural networks, this type of formal verification is more difficult. As we've discussed, it's hard to inspect the code and make any guarantees about how it will behave. You also don't have methods to determine its robustness or its stability. It all comes back to the fact that you can't explain what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

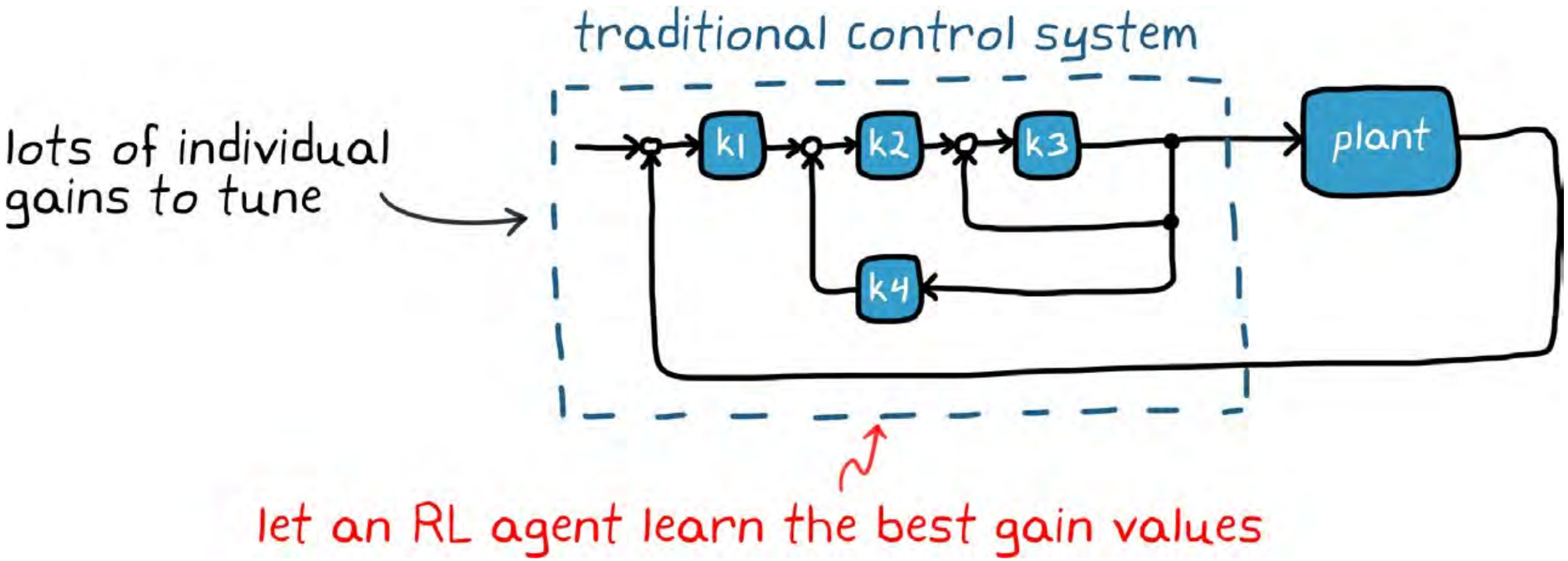
```
% software monitor
if abs(body_angle) > 45;    % monitor for falling
    mode = "safe";          % set safe mode
    extend_arms();          % prepare for impact
end
```

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

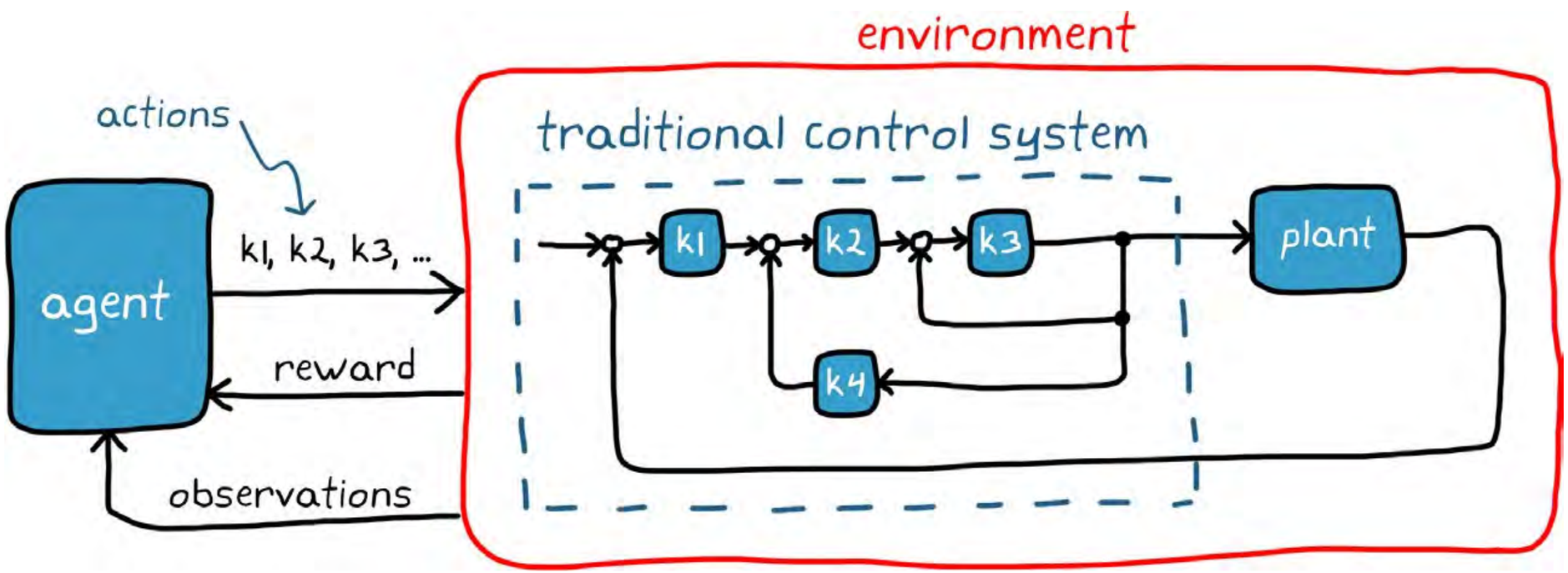
Workarounds are nice, but you can fix the issues directly by solving a different problem altogether. You can use reinforcement learning as a tool to optimize the controller gains in a traditionally architected control system. Imagine designing an architecture with dozens of nested loops and controllers,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

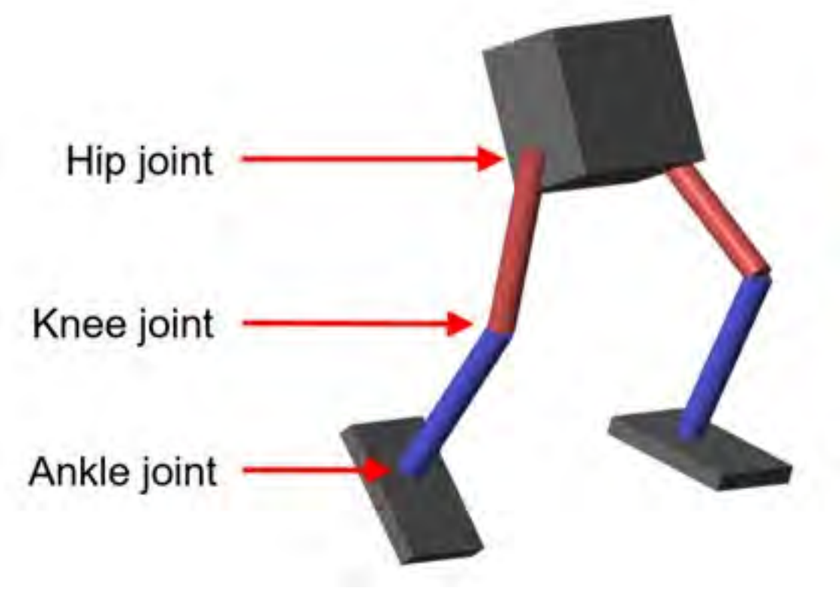
You get the best of both worlds with this method. You don't have to deploy any neural networks, verify them, or worry about having to change them; you just need to code the final static gain values into the controller. This way, you still have a traditionally architected system, one that can be verified 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.

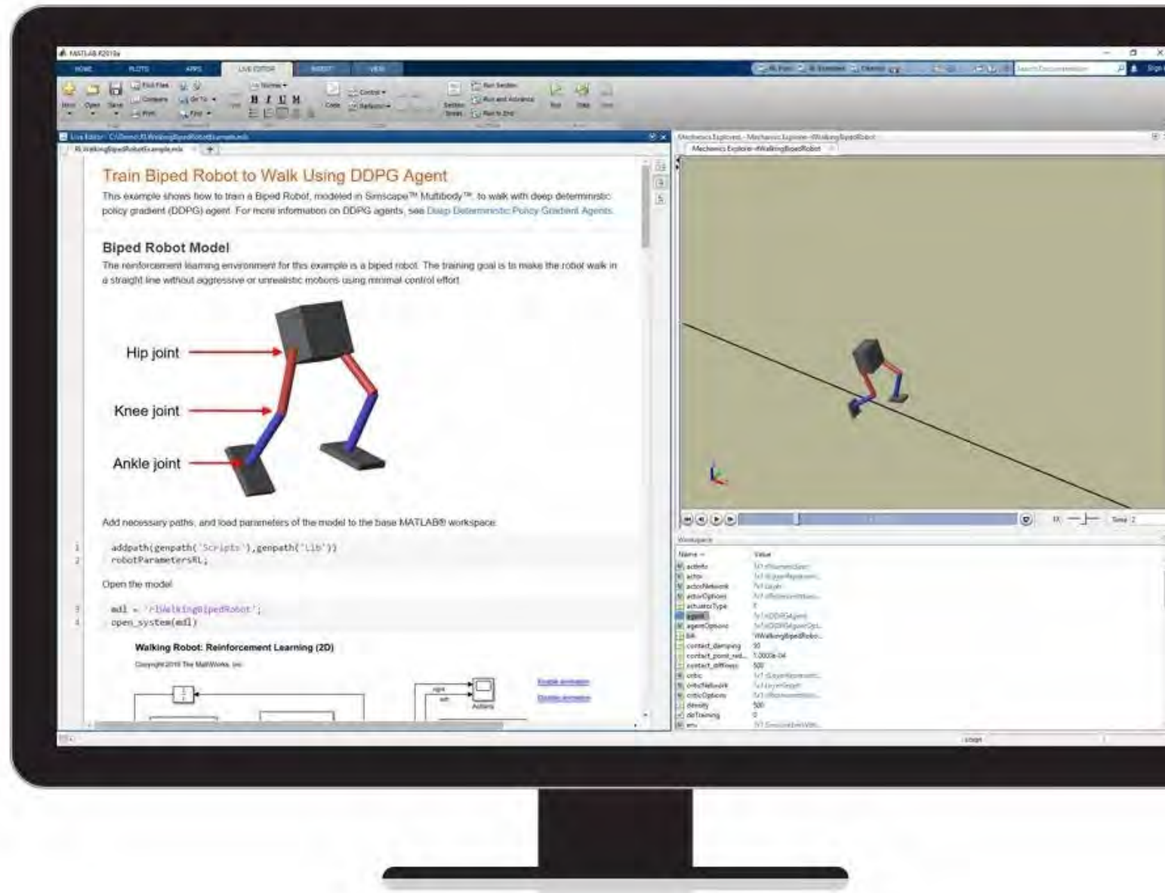
Learning algorithms, reinforcement learning design tools such as MATLAB and Reinforcement Learning Toolbox™, and verification methods are advancing all the time.



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

Develop Reinforcment Learning project with MATLAB, Simulink, and a full set of products for Reinforcement Learning

- Download Trial Now

Contact our technical experts for a FREE Demo.

- FREE Demo