Reinforcement Learning with MATLAB

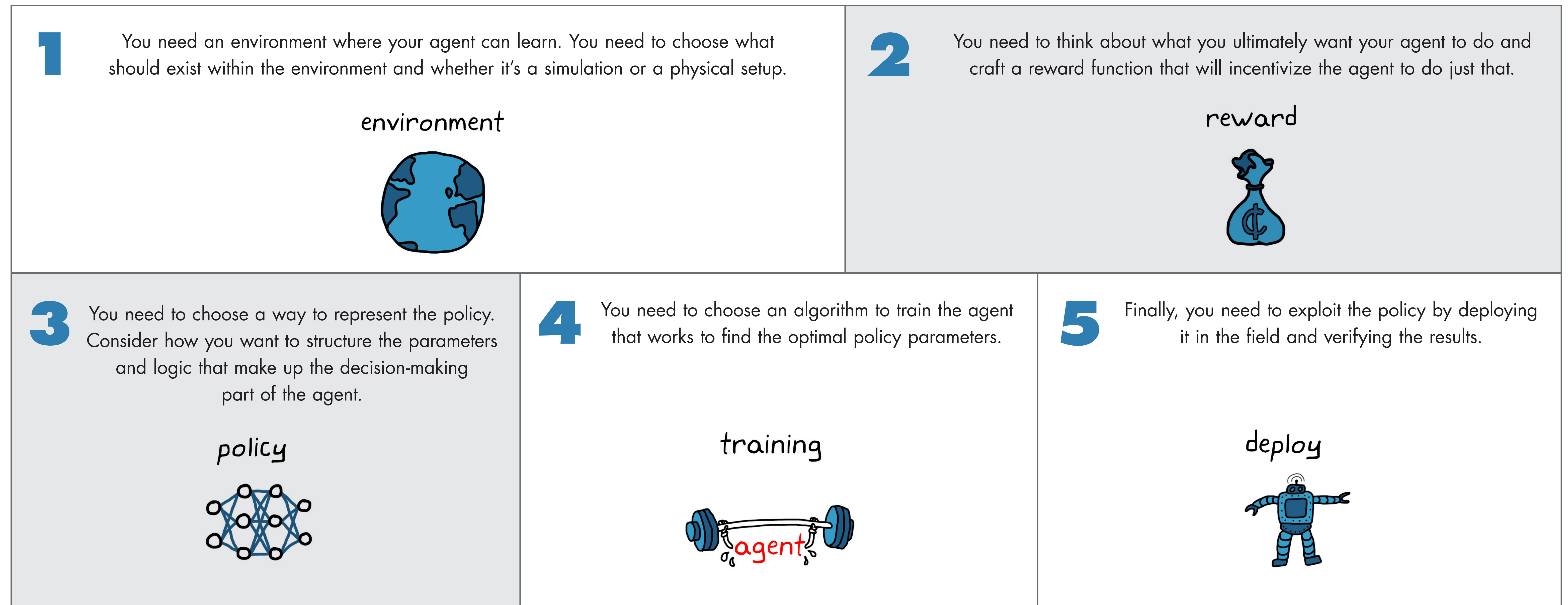
Understanding Rewards and Policy Structures

MathWorks®

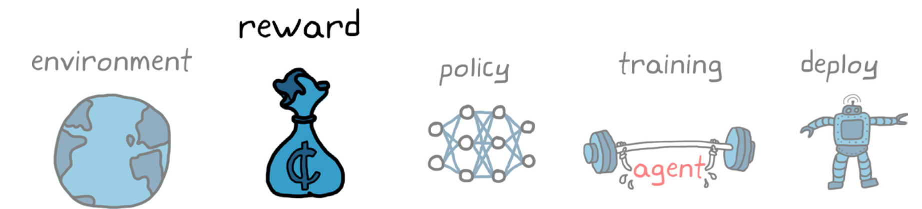
Reinforcement Learning Workflow Overview

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

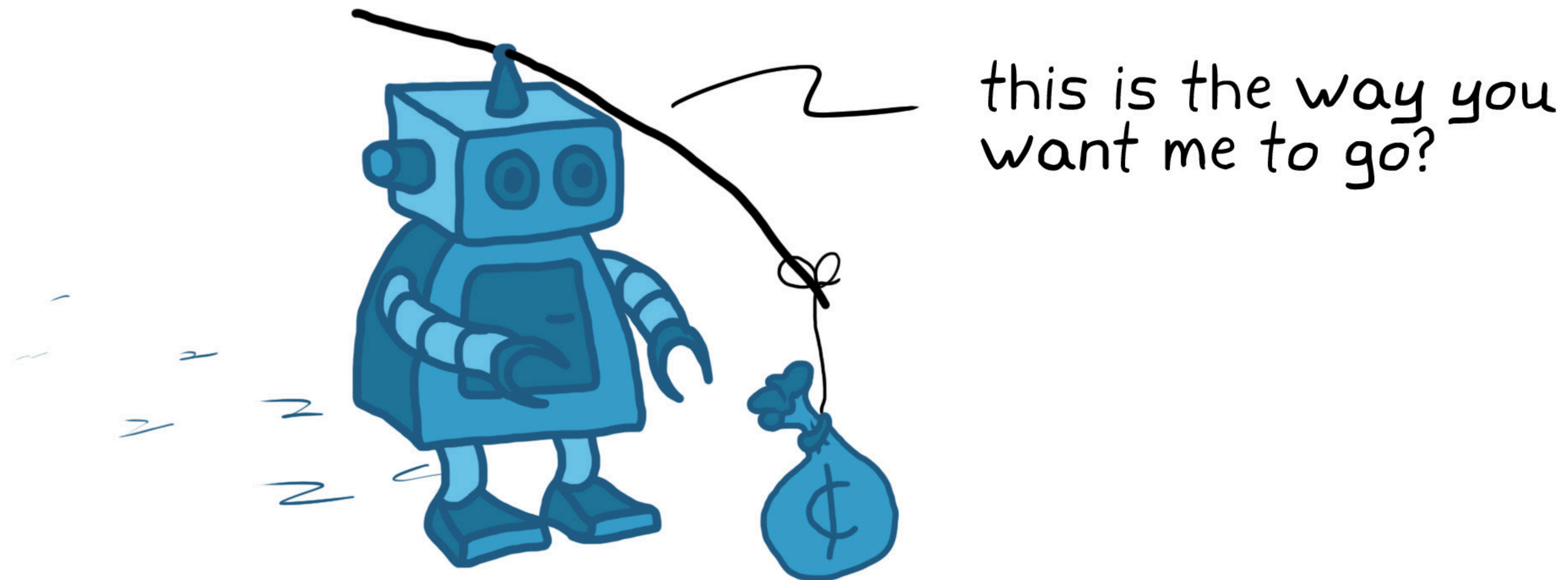
The first ebook focuses on *setting up the environment*. This ebook explores ***rewards and policy structures***. The last ebook covers *training and deployment.*



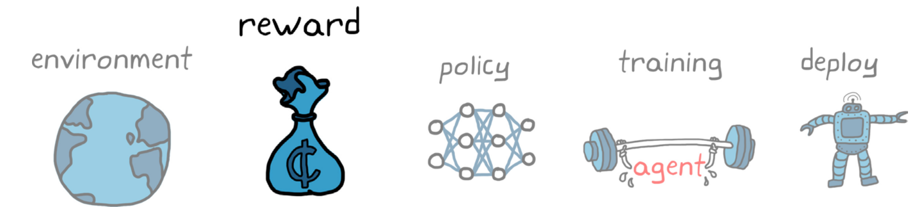
The Reward



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.

reward = function (state, action)

scalar representing “goodness”

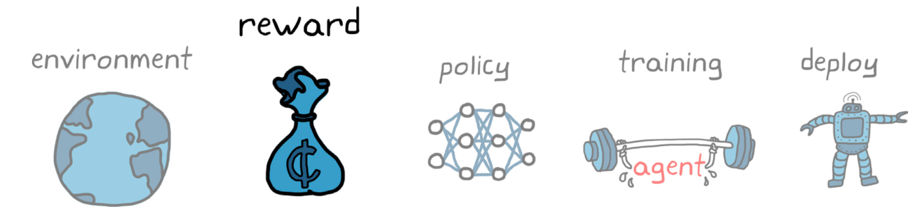
The concept is similar to the cost function in LQR, which penalizes bad system performance and increased actuator effort. The difference, of course, is that a cost function is trying to minimize the value, whereas a reward function tries to maximize the value. But this is solving the same problem since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

LQR cost function, $J = \int_0^\infty (x^T Q x + u^T R u)\,dt$

performance — effort — quadratic

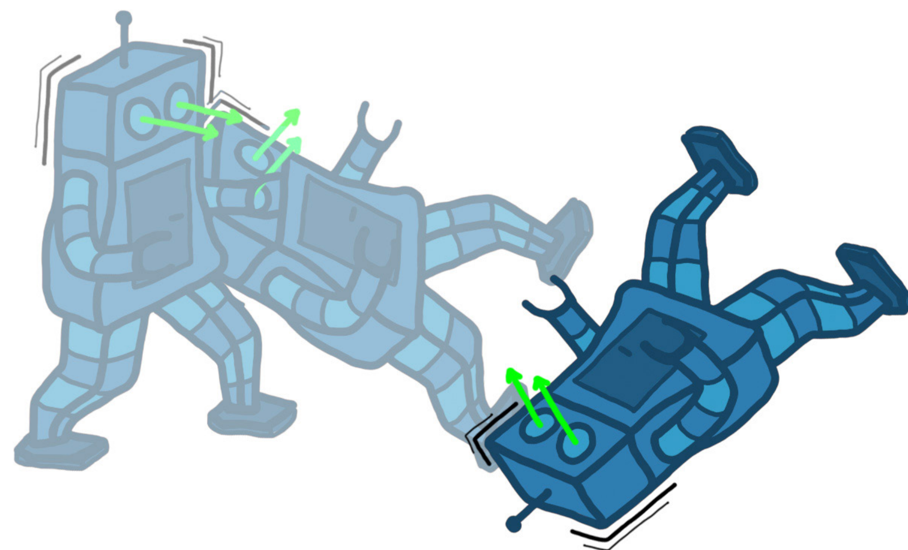
The main difference is that unlike LQR, where the cost function is quadratic, in reinforcement learning (RL) there’s really no restriction on creating a reward function. You can have sparse rewards, or rewards every time step, or rewards that only come at the very end of an episode after long periods 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



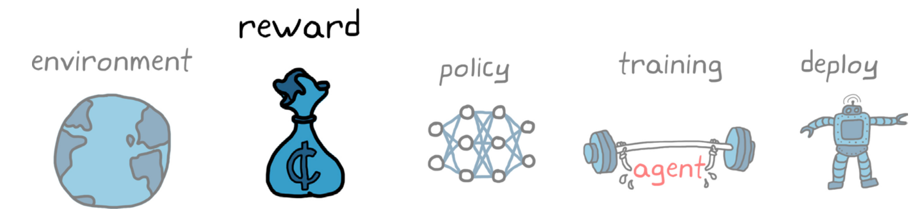
Since there are no constraints on how you create your reward function, you can get into situations where the rewards are sparse. This means that the goal you want to incentivize comes after a long sequence of actions. This would be the case for the walking robot if you set up the reward function such 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.

$$\text{reward} = \begin{cases} 1 & \text{for state} = 10 \text{ meters} \\ 0 & \text{for state} \sim= 10 \text{ meters} \end{cases}$$



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!

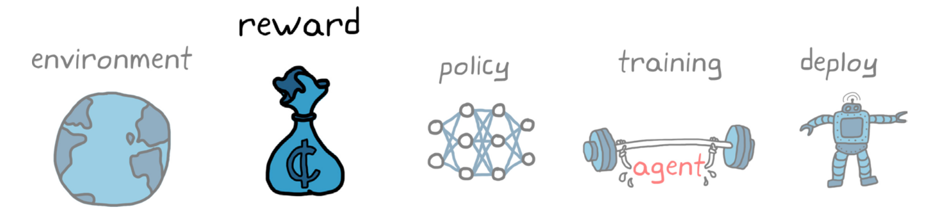
Reward Shaping



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.

$$r_t = v_x - 3y^2 - 50\hat{z}^2 + 25\frac{T_s}{T_f} - 0.02\sum_i {u_{t-1}^i}^2$$

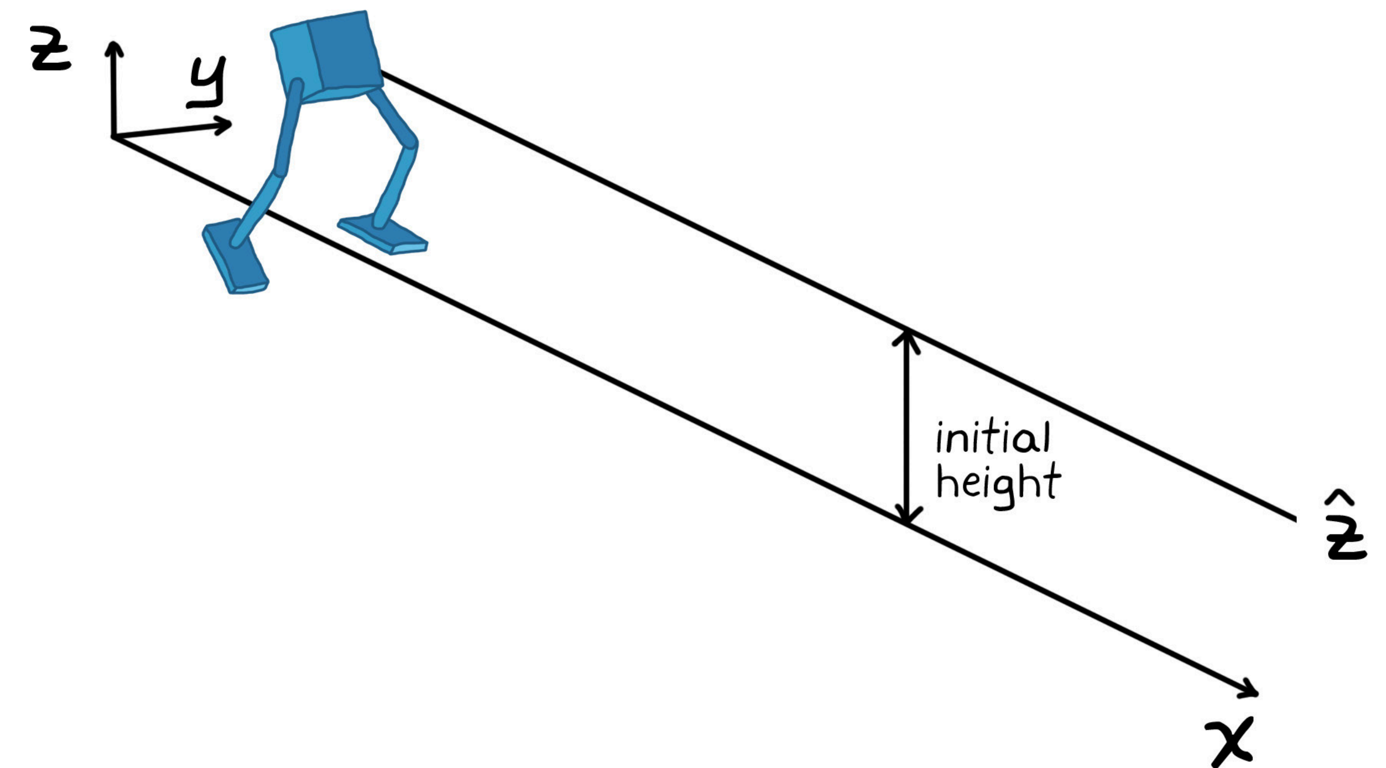
forward velocity

don't stray from path

keep trunk high

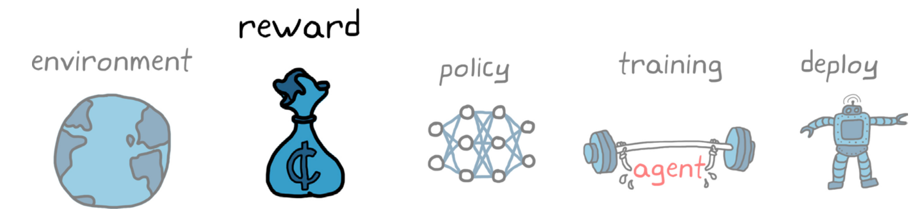
walk as long as possible

minimize actuator effort

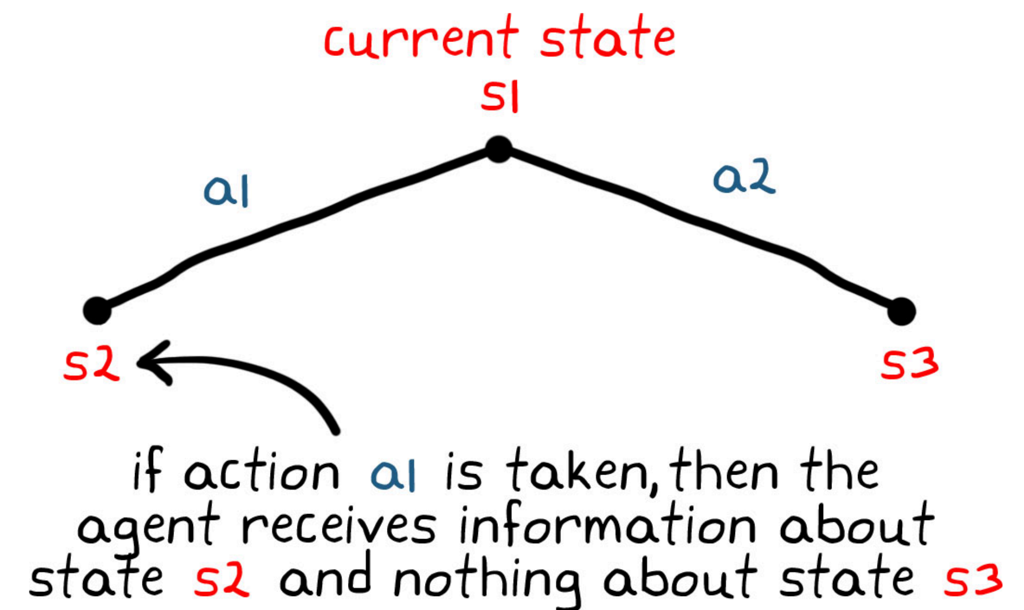


This is not intended to make crafting a reward function sound easy; getting it right is possibly one of the more difficult tasks in reinforcement learning. For example, you might not know if your reward function is poorly crafted until after you've spent a lot of time training your agent and it failed 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.

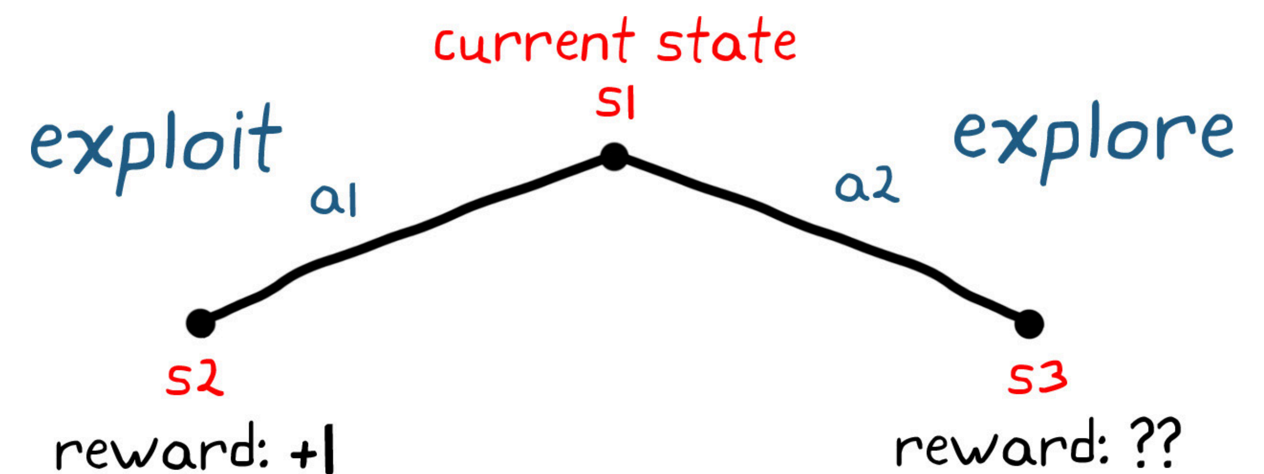
Exploration vs. Exploitation



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.

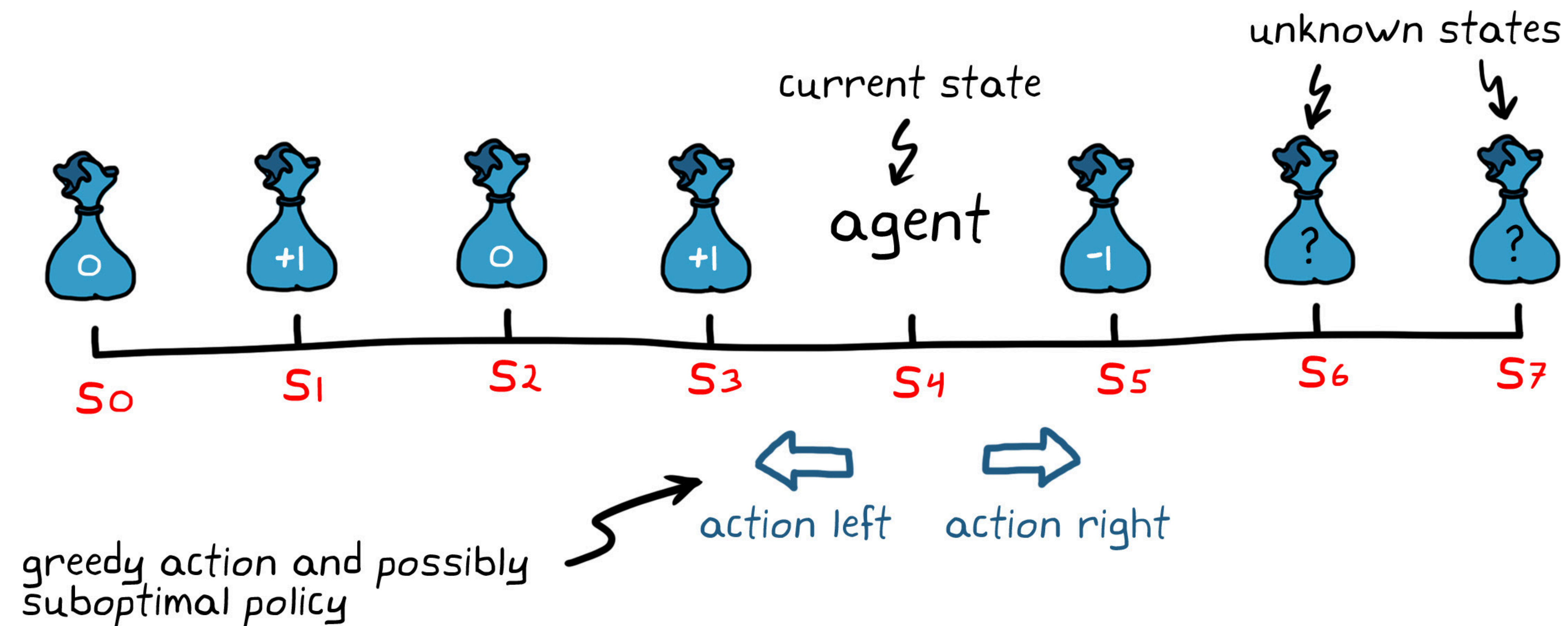


The idea is this: Should the agent exploit the environment by choosing the actions that collect the most rewards that it already knows about, or should it choose actions that explore parts of the environment that are still unknown?



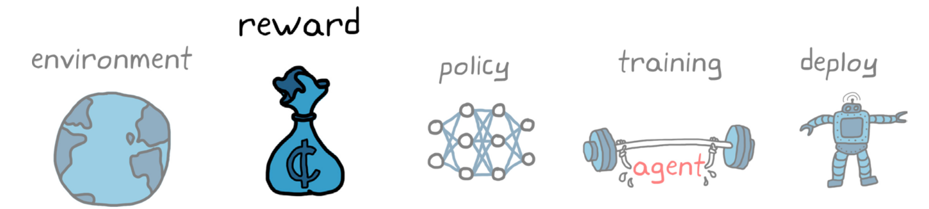
The Problem with Pure Exploitation

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.

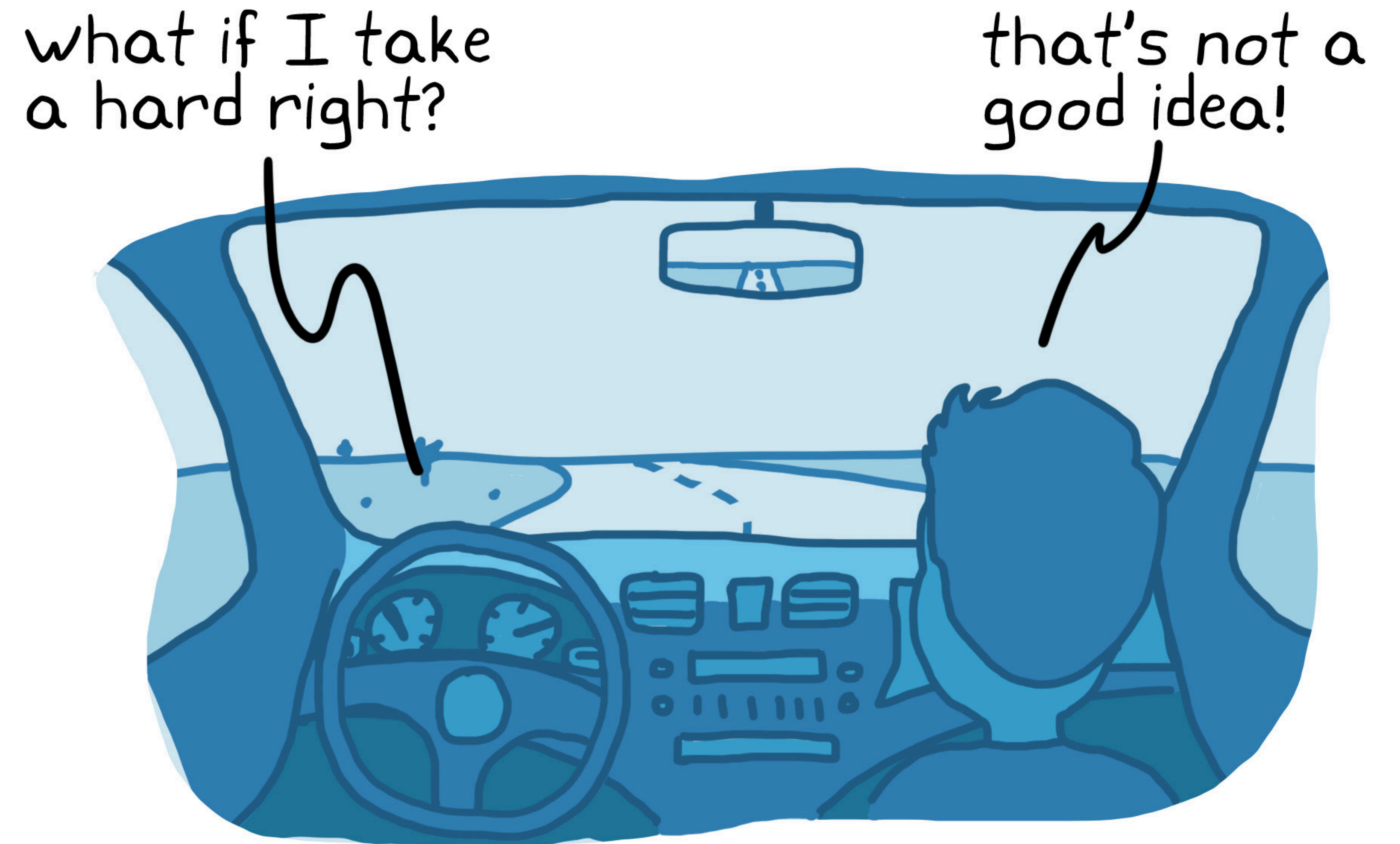


So you can see that if the agent is always exploiting what it thinks is the best action at any given time, it may never receive additional information about the states that exist beyond a low-reward action. This pure exploitation can increase the amount of time it takes to find the optimal policy or 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

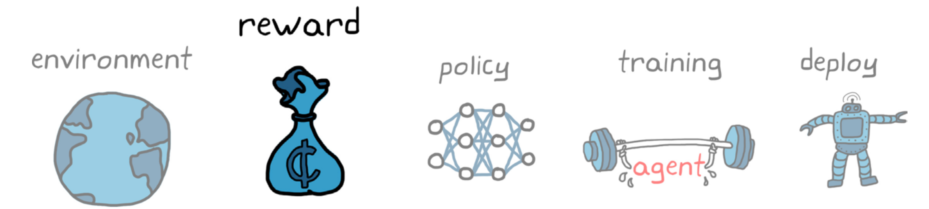


Instead, if you occasionally let the agent explore, even at the risk of collecting fewer rewards, it can expand its policy for the new states. This opens up the possibility of finding higher rewards it didn't know about and increases the chances of converging on the global solution. But you don't want 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.

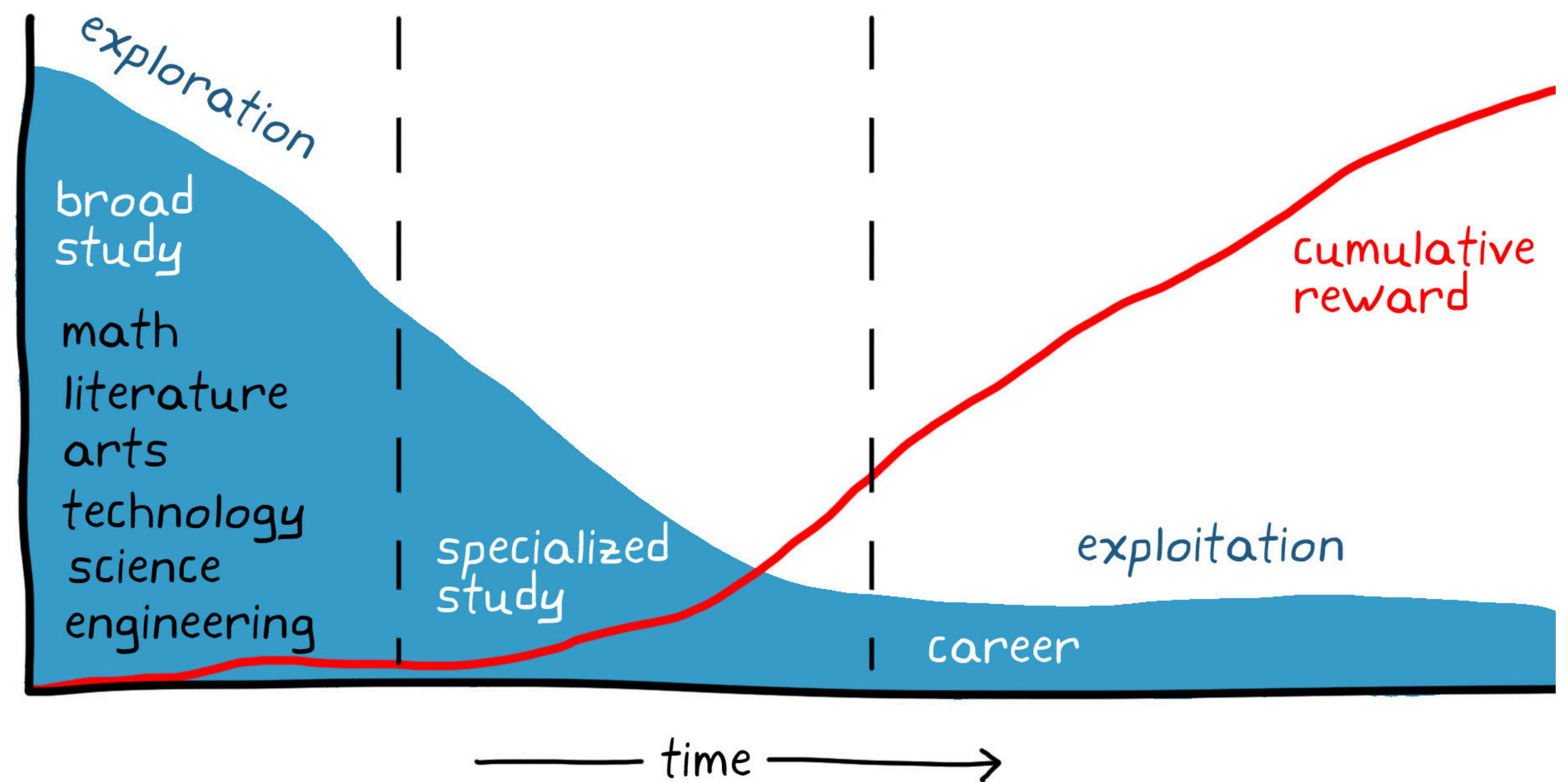
Balancing Exploration and Exploitation



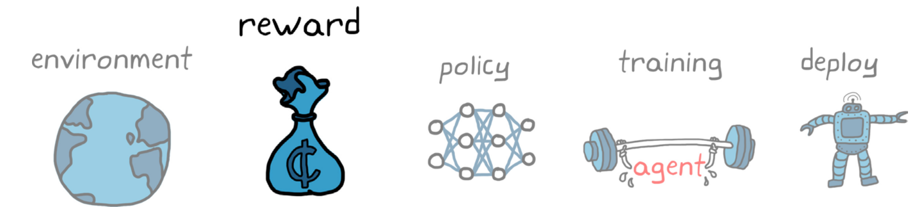
Consider how student might approach choosing a career path. When students are young, they explore different subjects and classes and are generally open to new experiences. After a certain amount of exploration, they are then likely to converge on learning more about a specialized subject and then finally 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



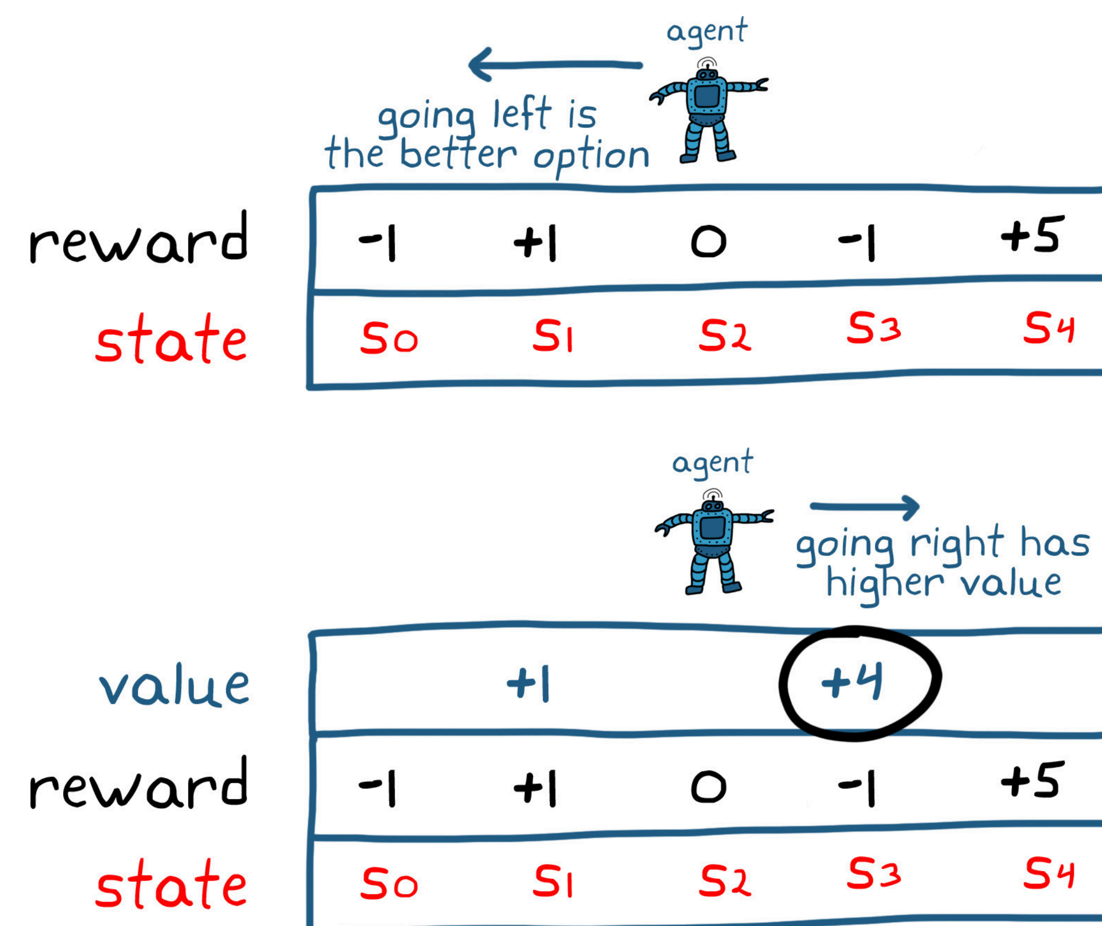
A second critical aspect of reinforcement learning is the concept of *value*. Assessing the value of a state or an action, rather than reward, helps the agent choose the action that will collect the most rewards over time rather than a short-term benefit.

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

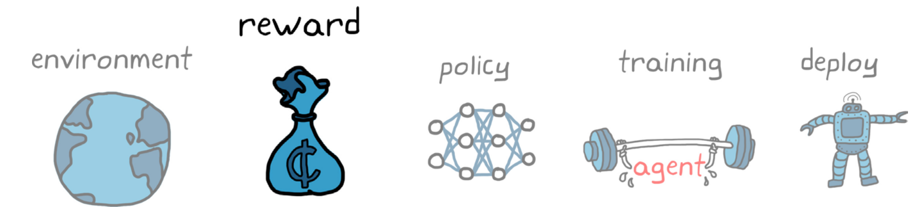
value: the total rewards an agent expects to receive from a state and onwards into the future

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.

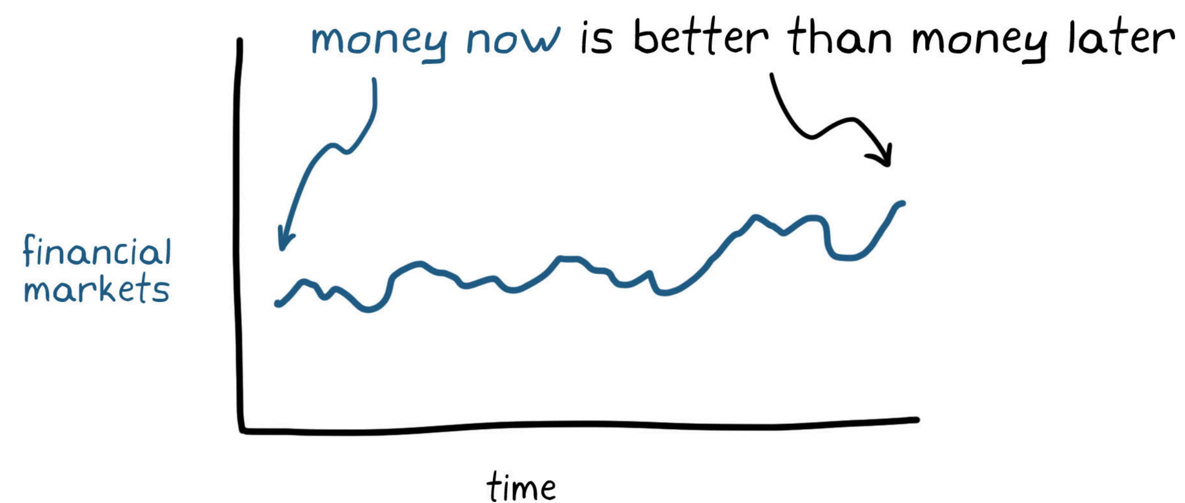


The Benefit of Being Short-Sighted

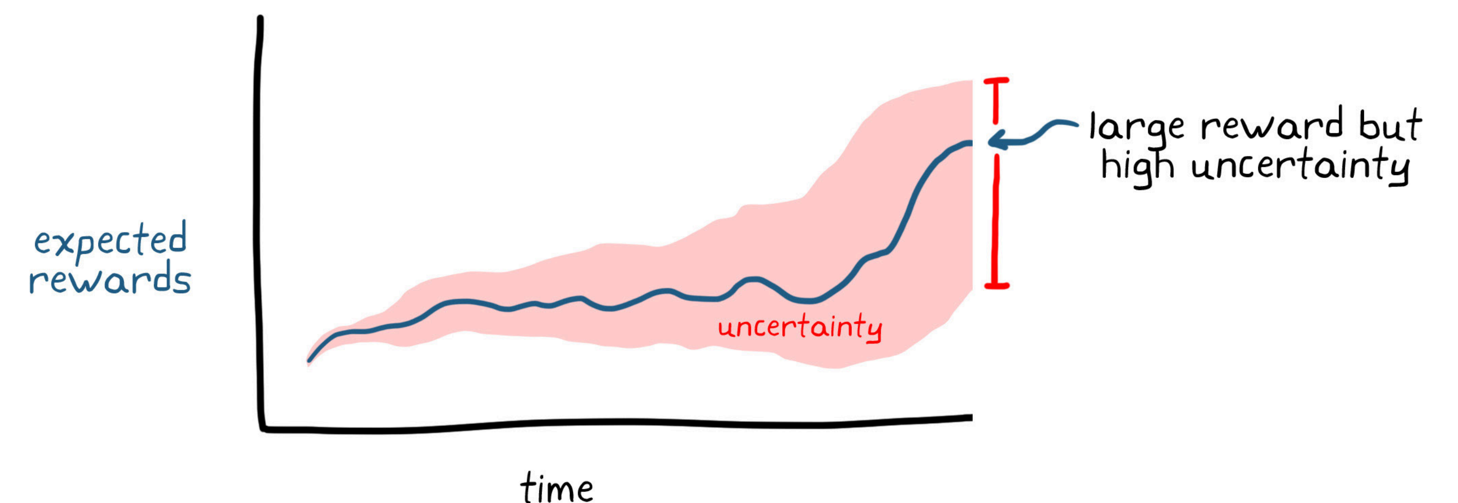


Of course, the promise of receiving a high reward after many sequential actions doesn't mean that the first action is necessarily the best; there are at least two good reasons for this.

First, like with the financial market, money in your pocket now can be better than a little more money in your pocket a year from now.



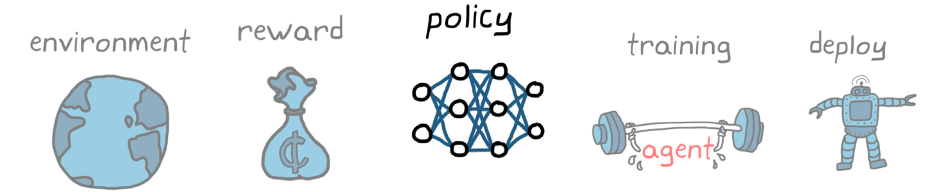
And second, your prediction of rewards further into the future becomes less reliable; therefore, that high reward might not be there by the time the agent reaches it.



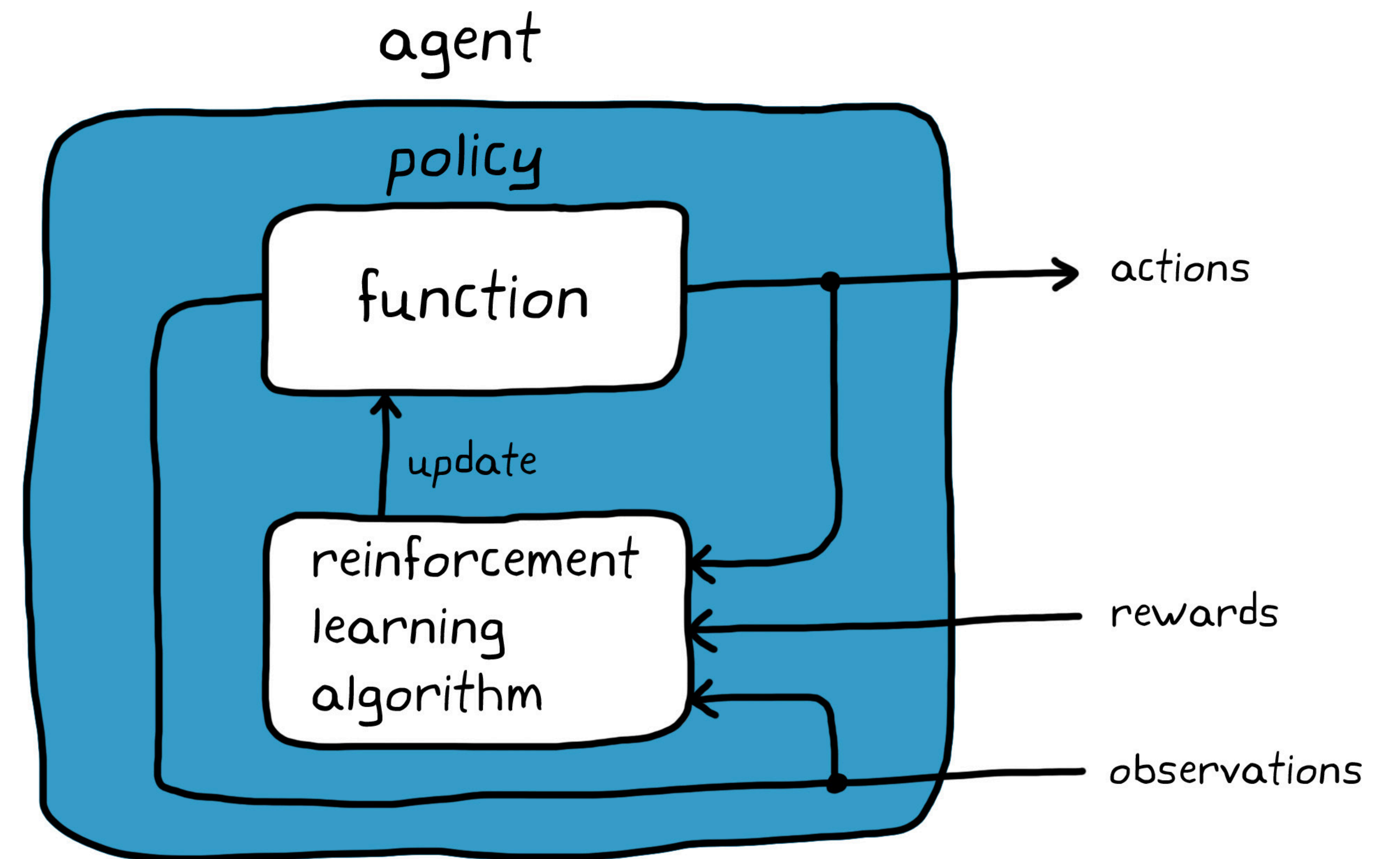
In both of these cases, it's advantageous to be a little more short-sighted when estimating value. In reinforcement learning, you can set how short-sighted you want your agent to be by discounting rewards by a larger amount the further they are in the future. This is done by setting the discount factor,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

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

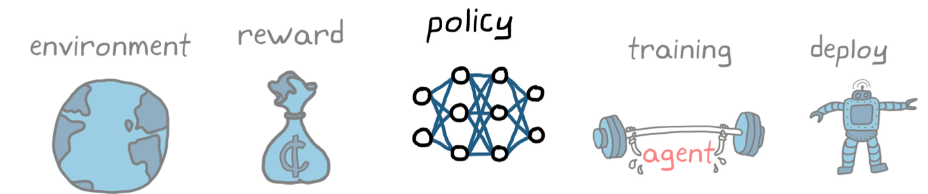
What Is the Policy?



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)

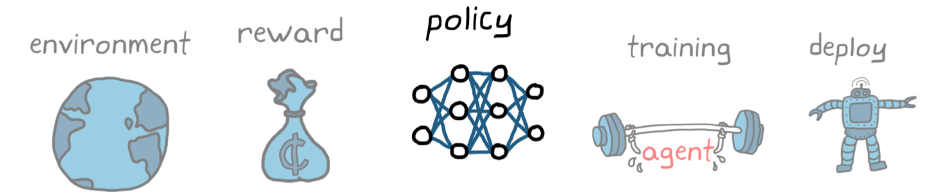
In general, there are two approaches for structuring the policy function:

- 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.

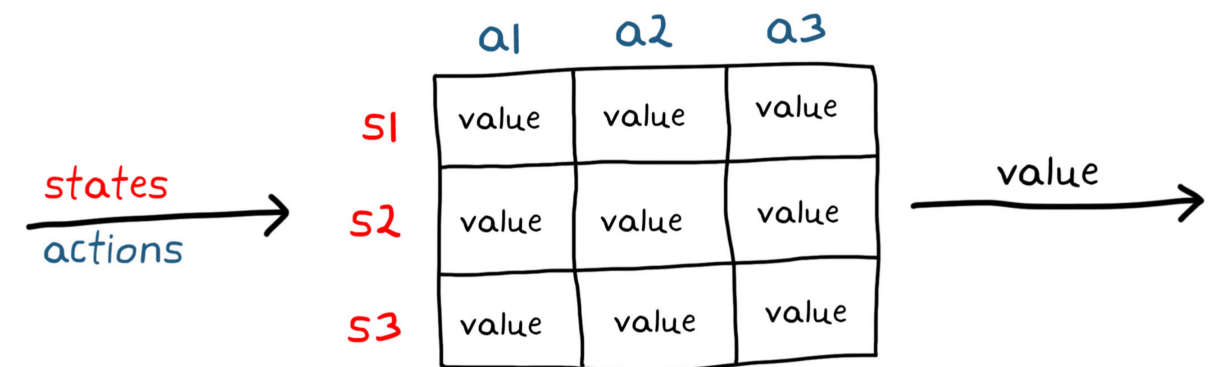
** Spoiler alert! You can combine the benefits of direct policy mapping and value-based mapping in a third method called actor-critic, which will be covered a bit later.*

Representing a Policy with a Table

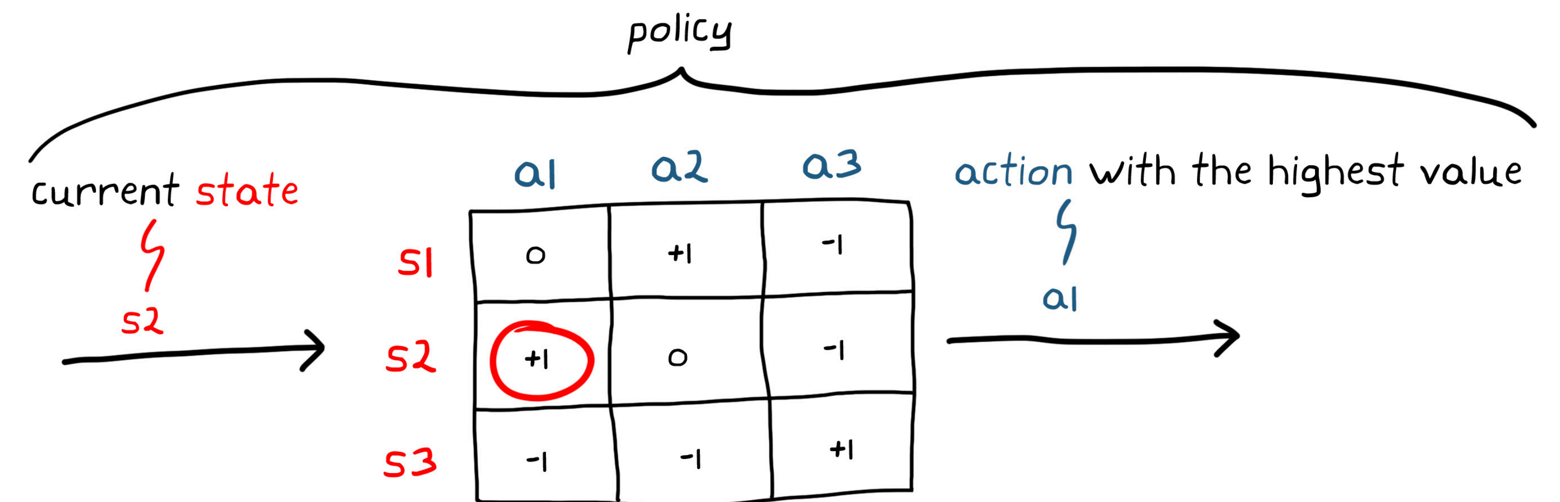


If the state and action spaces for the environment are discrete and few in number, you could use a simple table to represent policies.

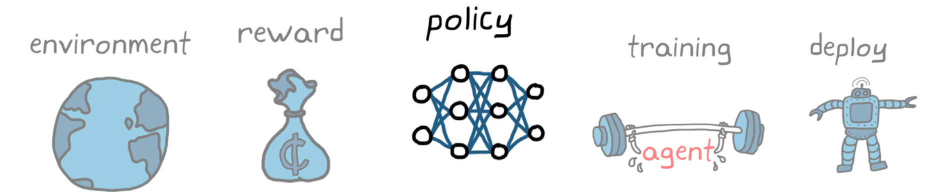
Tables are exactly what you'd expect: an array of numbers where an input acts as a lookup address and the output is the corresponding number in the table. One type of table-based function is the Q-table, which maps states and actions to value.



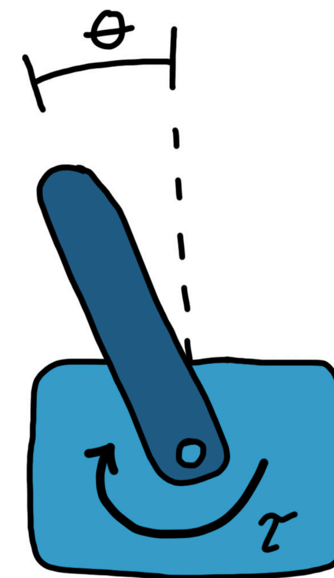
With a Q-table, the policy is to check the value of every possible action given the current state and then choose the action with the highest value. Training an agent with a Q-table would consist of determining the correct values for each state/action pair in the table. Once the table has been fully 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.



		torque				
angle rate	angle	-0.01	-0.001	0.02	0.021	...
-0.3	0.124	value	value	value	value	
0.17	0.137	value	value	value	value	
0.175	0.139	value	value	value	value	
0.223	0.204	value	value	value	value	
⋮	⋮					

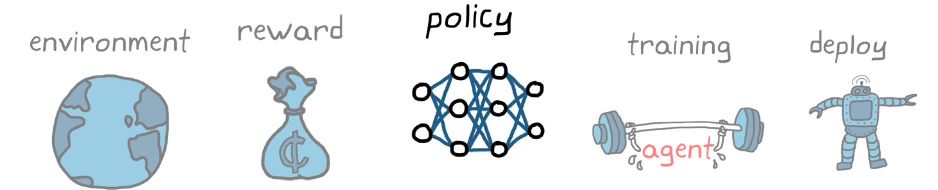
You could represent the continuous nature of the inverted pendulum with a continuous function—something that takes in states and outputs actions. However, before you could start learning the right parameters in this function, you would need to define the logical structure. This might be difficult to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

$$\text{value} = -(\dot{\theta}^2 + \theta^2) + \tau \,? \qquad \text{value} = \dot{\theta}\sin(\theta) + \tau \,?$$

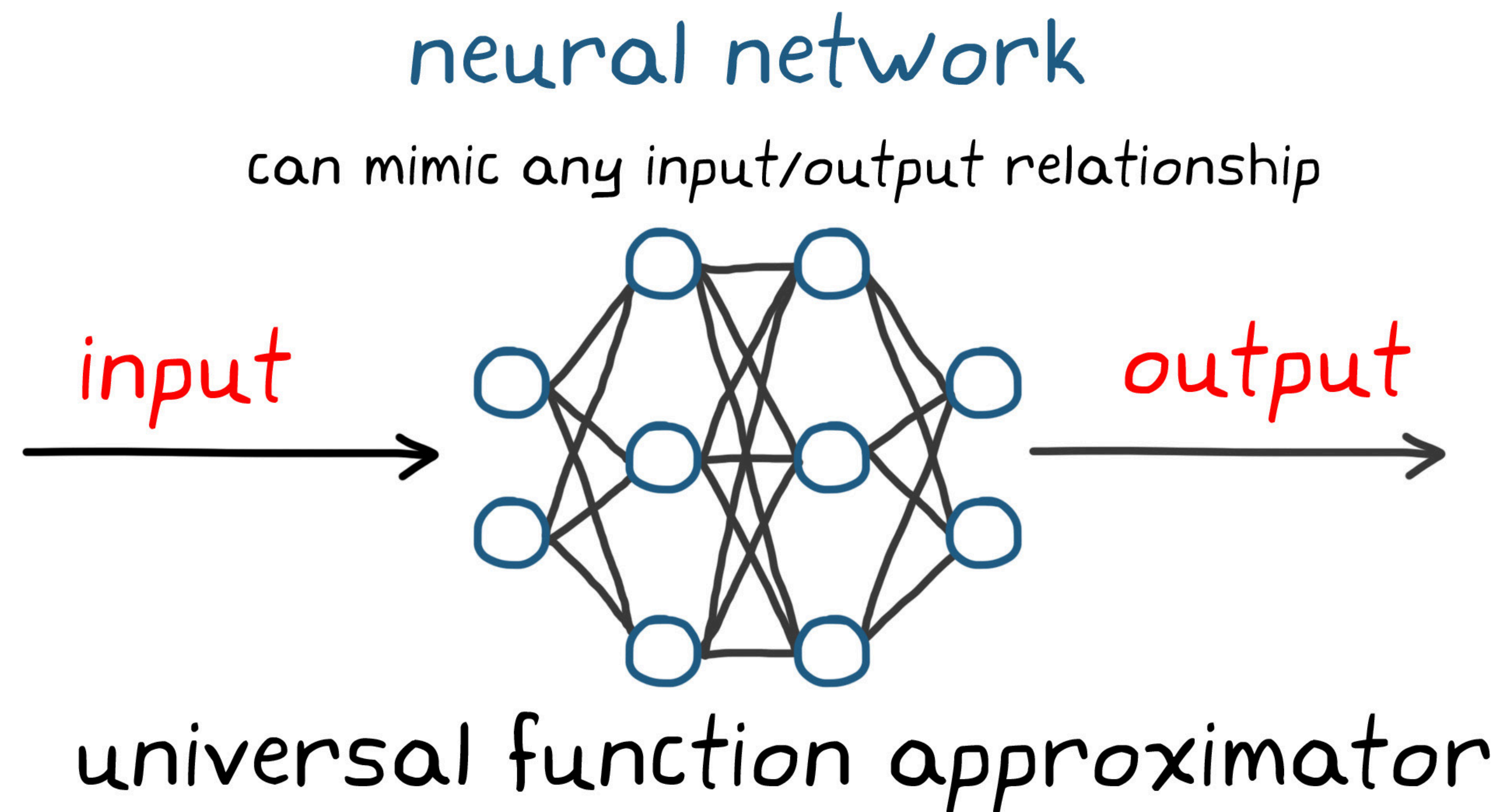
we need to define the logical structure

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

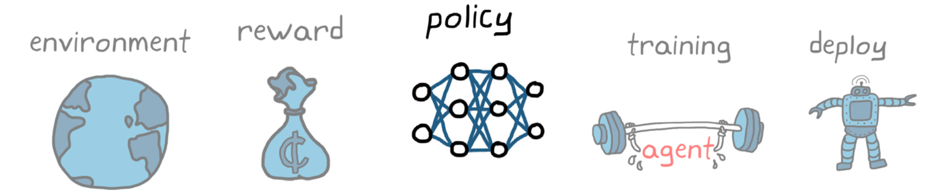


A *neural network* is a group of nodes, or artificial neurons, that are connected in a way that allows them to be a universal function approximator. This means that given the right combination of nodes and connections, you can set up the network to mimic any input and output relationship. Even though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



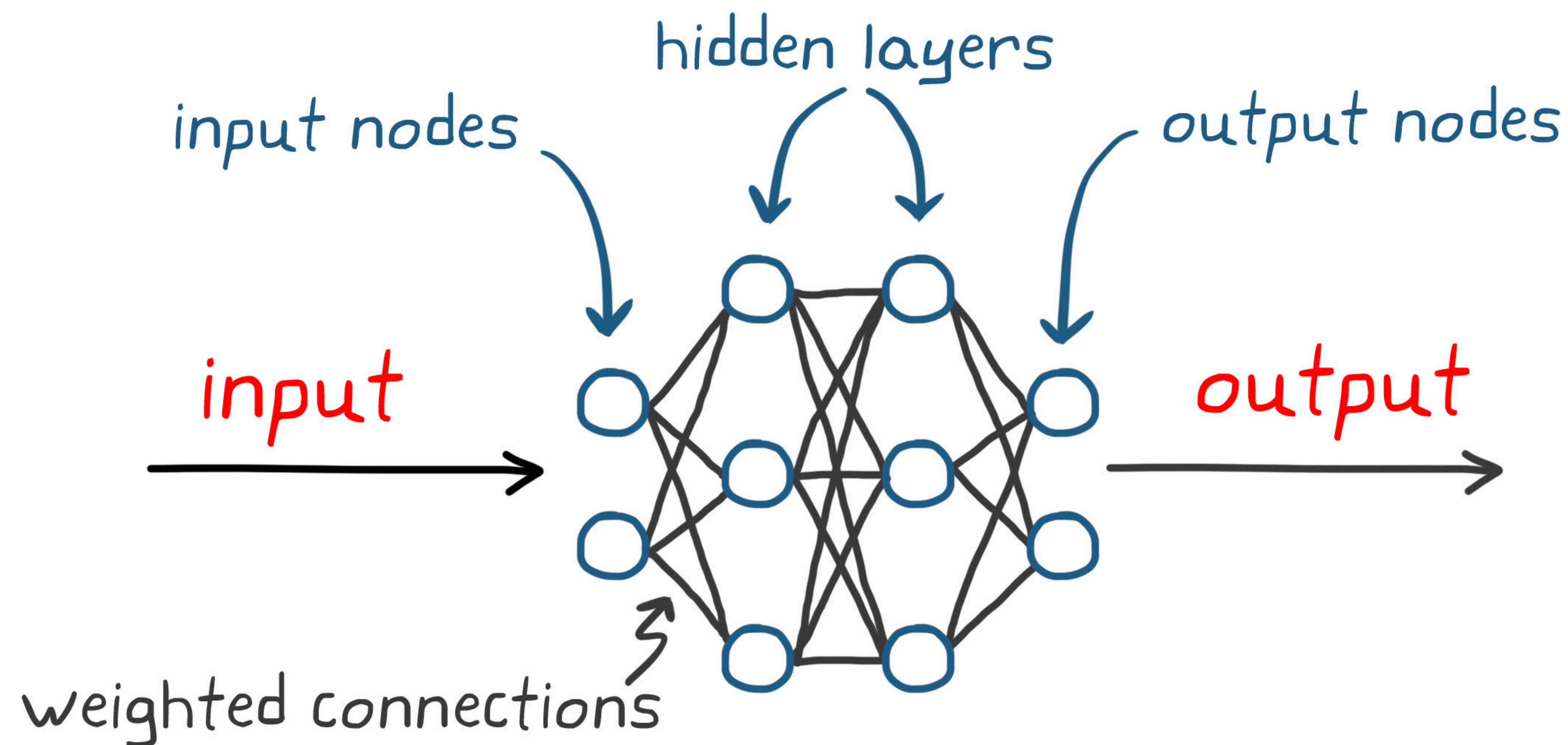
So instead of trying to find the perfect nonlinear function structure that works with a specific environment, with a neural network you can use the same combination of nodes and connections in many different environments. The only difference is in the parameters themselves. The learning process would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

What Is a Neural Network?

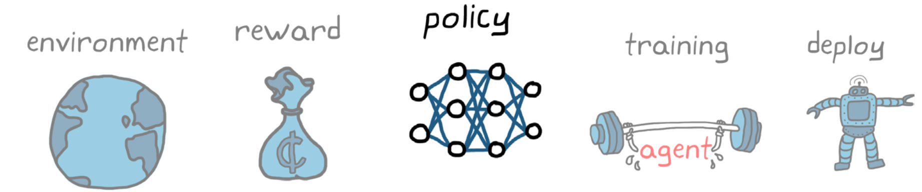


The mathematics of neural networks are not covered in depth here. But it's important to highlight a few things to help explain some of the decisions later on when setting up the policies.

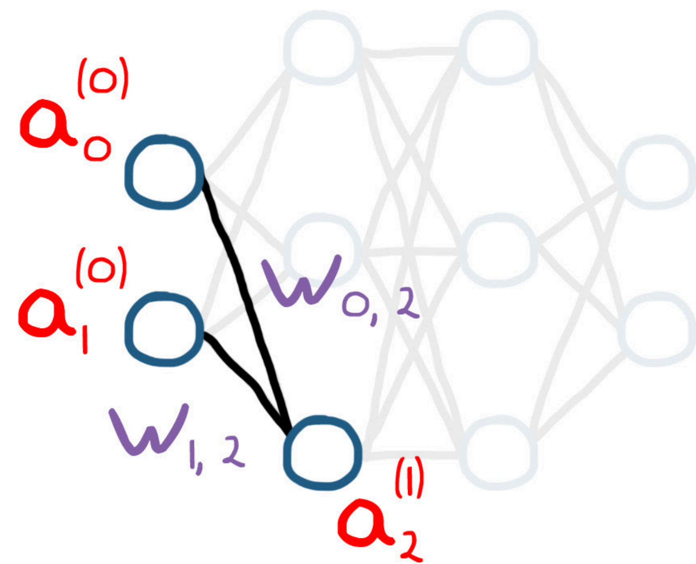
On the left are the input nodes, one for each input to the function, and on the right are the output nodes. In between are columns of nodes called hidden layers. This network has 2 inputs, 2 outputs, and 2 hidden layers of 3 nodes each. With a fully connected network, there is a weighted connection from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



$$\text{value of } a_2^{(1)} = w_{0,2} \cdot a_0^{(0)} + w_{1,2} \cdot a_1^{(0)} + b_2^{(1)}$$

(superscript: layer; subscript: node position)

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.

transform from layer 0 to layer 1

$$a^{(1)} = W_0 \, a^{(0)} + b^{(1)}$$

matrices

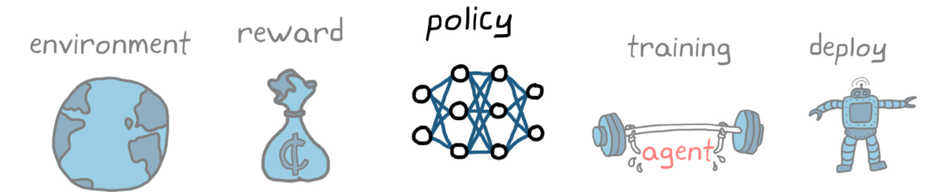
transform from layer 1 to layer 2

$$a^{(2)} = W_1 \, a^{(1)} + b^{(2)}$$

transform from layer 2 to layer 3

$$a^{(3)} = W_2 \, a^{(2)} + b^{(3)}$$

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.

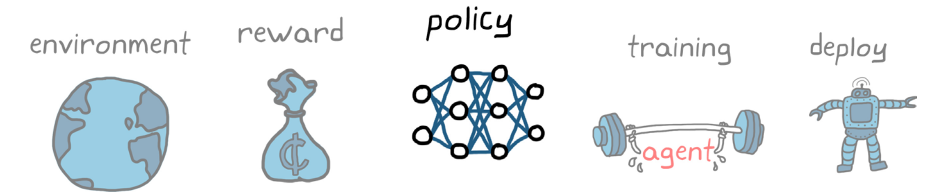
$$a^{(1)} = act\,[\,W_0\, a^{(0)} + b^{(1)}\,]$$

activation function is applied after linear operations

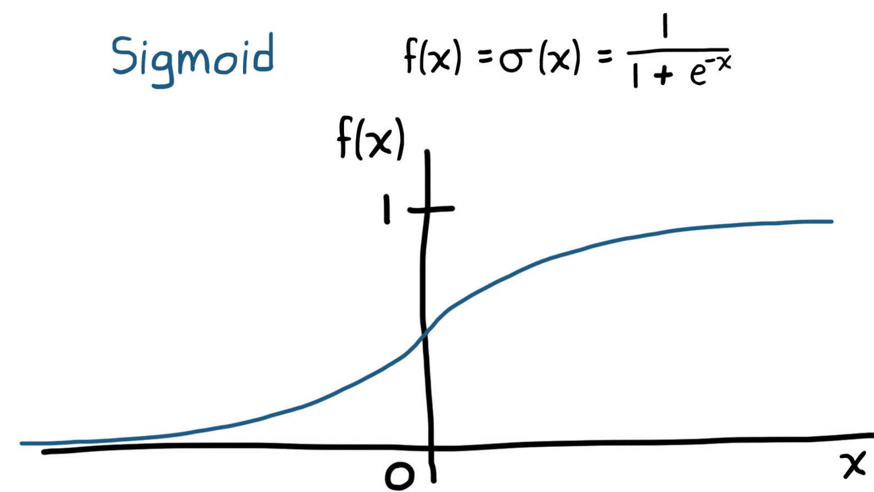
There are a number of different activation functions. What they all have in common is that they are nonlinear, which is critical to making a network that can approximate any function. Why is this the case? Because many nonlinear functions can be broken down into a weighted combination of activation function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

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

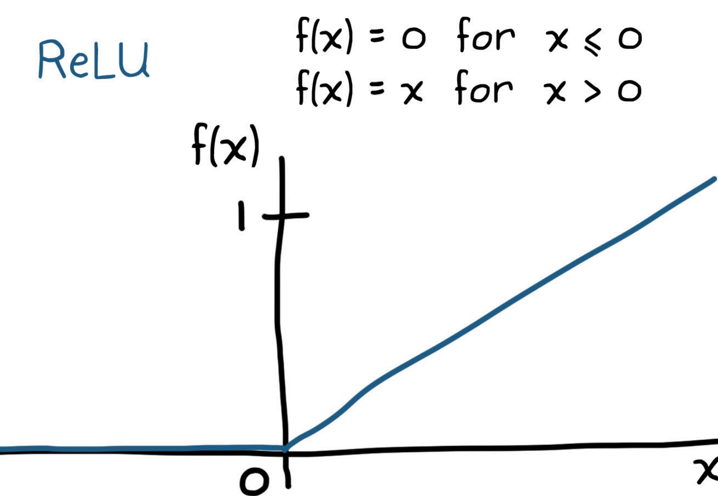
ReLU and Sigmoid Activations



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.



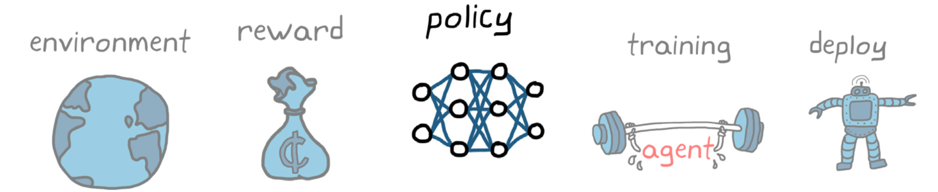
The rectified linear unit (ReLU) function zeroes out any negative node values and leaves the positive values unmodified.



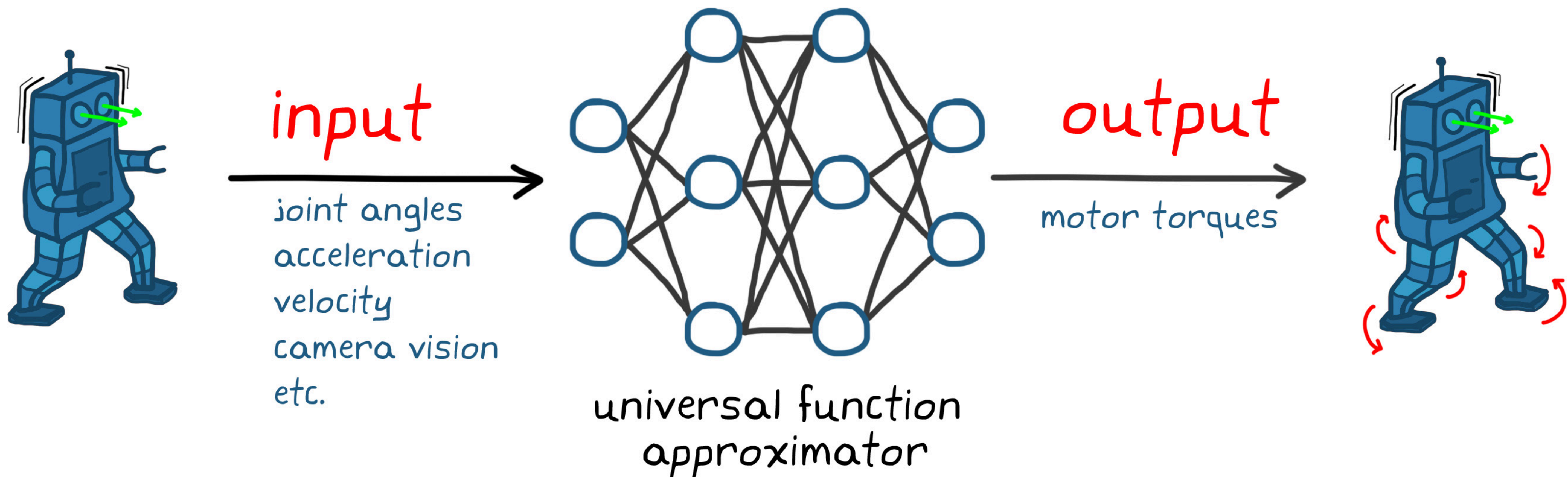
As an example, a pre-activation node value of -2 would become 0.12 with a sigmoid activation and 0 with a ReLU activation.

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

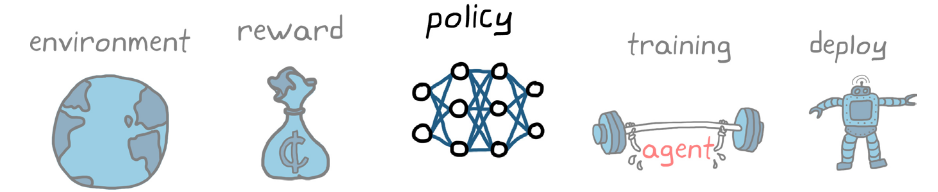
Representing a Policy with a Neural Network



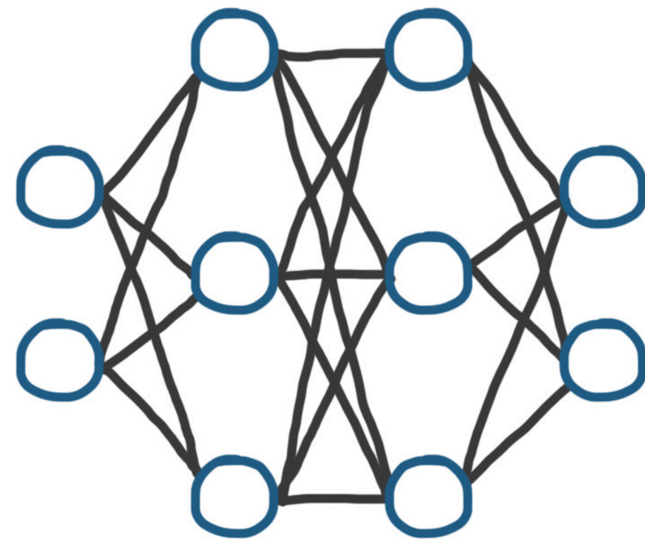
Let's recap before moving on. You want to find a function that can take in a large number of observations and transform them into a set of actions that will control some nonlinear environment. And since the structure of this function is often too complex to solve for directly, you want to approximate 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.

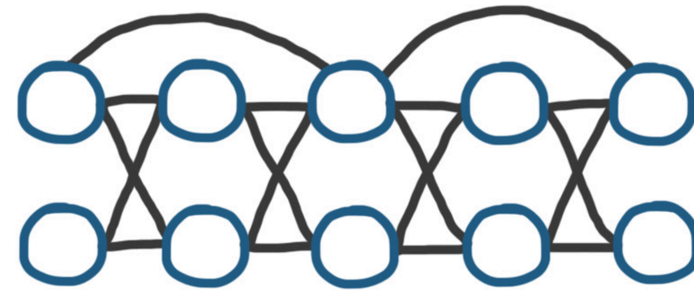


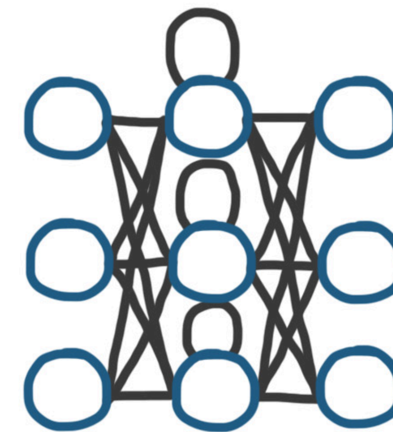
Neural Network Structures

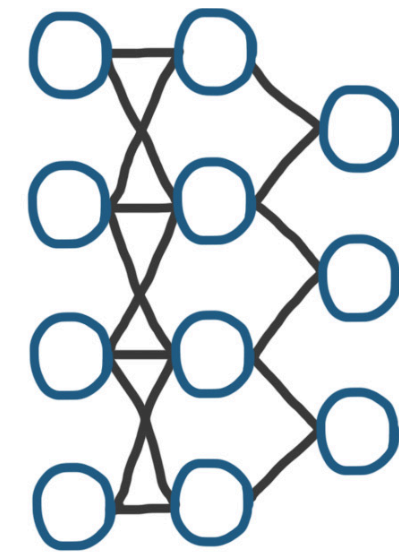


You have to make a few choices about the neural network ahead of time in order to make sure it's complex enough to approximate the function you're looking for, but not so complex as to make training impossible or impossibly slow. For example, as you've already seen, you need to choose an activation function, 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?







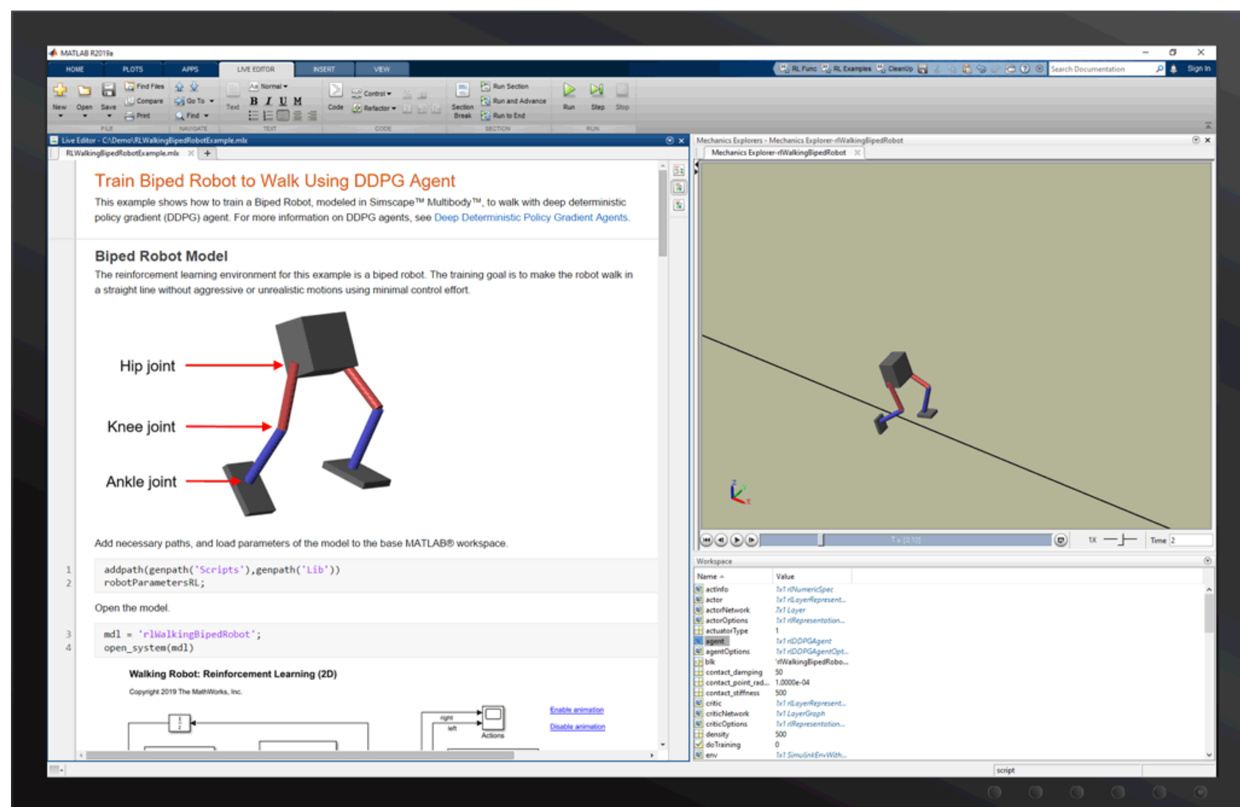


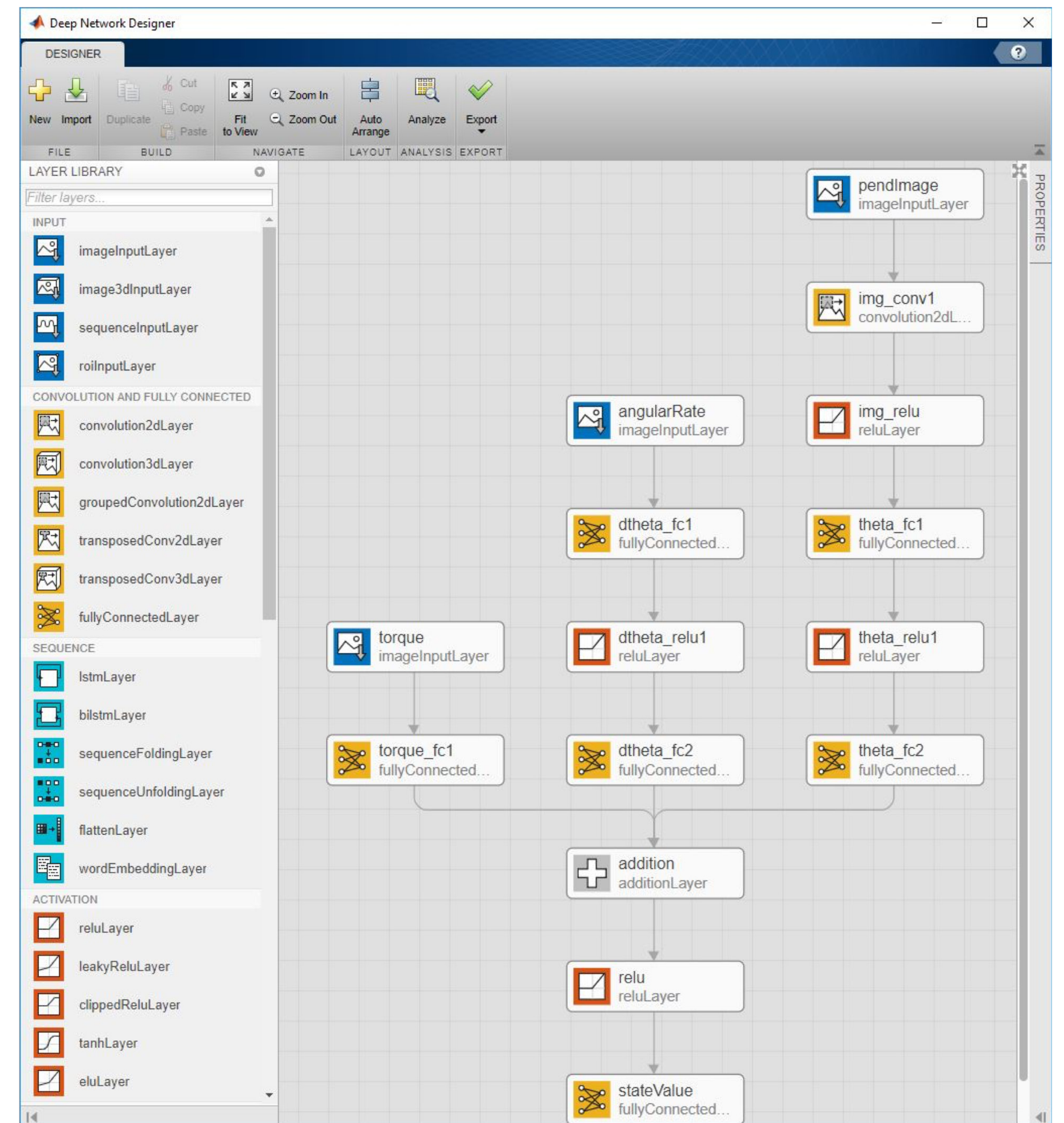
As with other control techniques, there isn't one right approach for settling on a neural network structure. A lot of it comes down to starting with a structure that has already worked for the type of problem you're trying to solve and tweaking it from there.

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.

Learn More

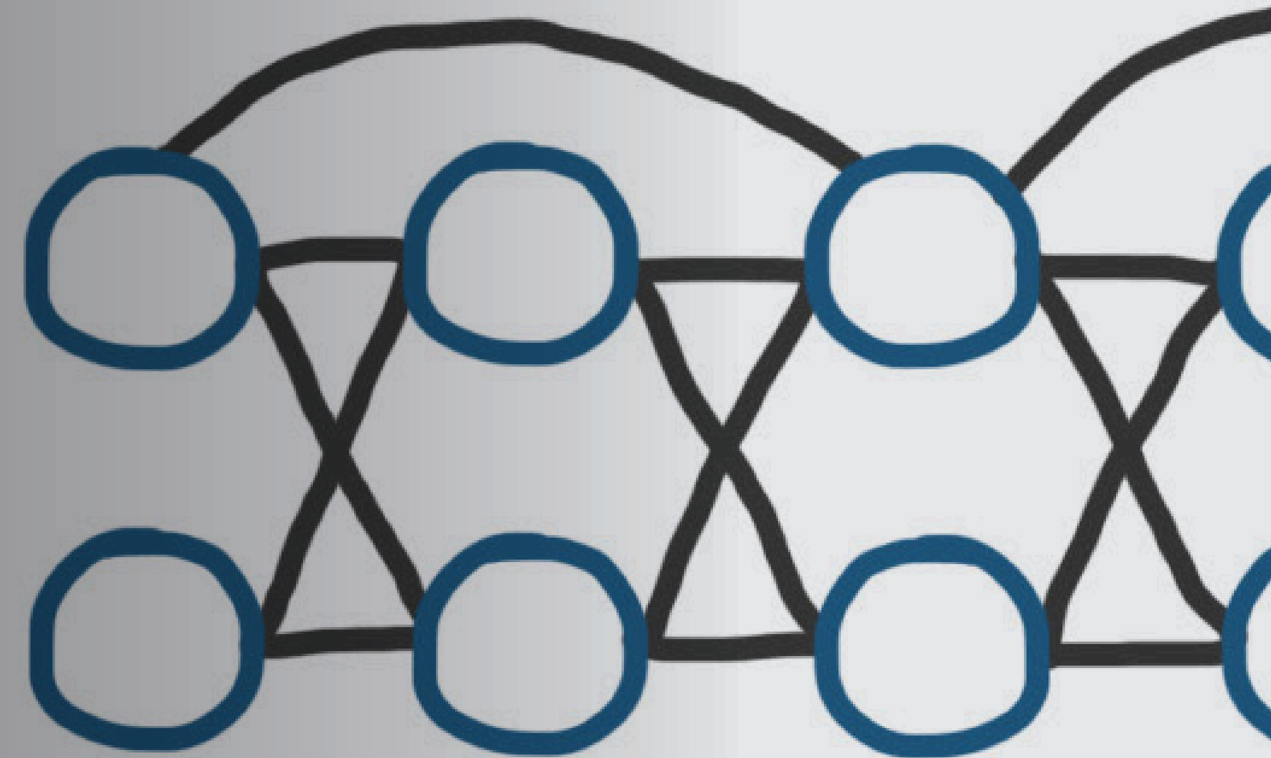
Reinforcement Learning Toolbox - Overview

Understanding Policies and Learning Algorithms (17:50) - Video

Defining Reward Signals in MATLAB and Simulink - Documentation

Policy and Value Function Representations - Documentation

Reference Examples for Getting Started - Examples



© 2019 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.