of the game upon victory or defeat, while in helicopter flight, rewards might be presented for successful flights or completion of trick moves.

Reward Function Engineering Is Hard

One of the largest challenges in reinforcement learning is designing rewards that induce agents to learn desired behaviors. For even simple win/loss games such as Go or tic-tac-toe, this can be surprisingly difficult. How much should a loss be punished and how much should a win be rewarded? There don't yet exist good answers.

For more complex behaviors, this can be extremely challenging. A number of studies have demonstrated that simple rewards can result in agents learning unexpected and even potentially damaging behaviors. These systems spur fears of future agents with greater autonomy wreaking havoc when unleashed in the real world after having been trained to optimize bad reward functions.

In general, reinforcement learning is less mature than supervised learning techniques, and we caution that decisions to deploy reinforcement learning in production systems should be taken very carefully. Given uncertainty over learned behavior, make sure to thoroughly test any deployed reinforcement learned system.

Reinforcement Learning Algorithms

Now that we've introduced you to the core mathematical structures underlying reinforcement learning, let's consider how to design algorithms that learn intelligent behaviors for reinforcement learning agents. At a high level, reinforcement learning algorithms can be separated into the buckets of *model-based* and *model-free* algorithms. The central difference is whether the algorithm seeks to learn an internal model of how its environment acts. For simpler environments, such as tic-tac-toe, the model dynamics are trivial. For more complex environments, such as helicopter flight or even ATARI games, the underlying environment is likely extraordinarily complex. Avoiding the construction of an explicit model of the environment in favor of an implicit model that advises the agent on how to act may well be more pragmatic.

Simulations and Reinforcement Learning

Any reinforcement learning algorithm requires iteratively improving the performance of the current agent by evaluating the agent's current behavior and changing it to improve received rewards. These updates to the agent structure often include some gradient descent update, as we will see in the following sections. However, as you know intimately from previous chapters, gradient descent is a slow training algorithm! Millions or even billions of gradient descent steps may be required to learn an effective model.

This poses a problem if the learning environment is in the real world; how can an agent interact millions of times with the real world? In most cases it can't. As a result, most sophisticated reinforcement learning systems depend critically on simulators that allow interaction with a simulation computational version of the environment. For the helicopter flight environment, one of the hardest challenges researchers faced was building an accurate helicopter physics simulator that allowed learning of effective flight policies computationally.

Q-Learning

In the framework of Markov decision processes, agents take actions in an environment and obtain rewards that are (presumably) tied to agent actions. The *Q* function predicts the expected reward for taking a given action in a particular environment state. This concept seems very straightforward, but the trickiness arises when this expected reward includes discounted rewards from future actions.

Discounting Rewards

The notion of a discounted reward is widespread, and is often introduced in the context of finances. Suppose a friend says he'll pay you $10 next week. That future 10 dollars is worth less to you than 10 dollars in your hand right now (what if the payment never happens, for one?). So mathematically, it's common practice to introduce a discounting factor γ (typically between 0 and 1) that lowers the "present-value" of future payments. For example, say your friend is somewhat untrustworthy. You might decide to set $\gamma = 0.5$ and value your friend's promise as worth $10\gamma = 5$ dollars today to account for uncertainty in rewards.

However, these future rewards depend on actions taken by the agent in the future. As a result, the *Q* function must be formulated recursively in terms of itself, since expected rewards for one state depend on those for another state. This recursive definition makes learning the *Q* function tricky. This recursive relationship can be