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

formulated explicitly for simple environments with discrete state spaces and solved with dynamic programming methods. For more general environments, *Q*-learning methods were not very useful until recently.

Recently, Deep *Q*-networks (DQN) were introduced by DeepMind and used to solve ATARI games as mentioned earlier. The key insight underlying DQN is once again the universal approximation theorem; since *Q* may be arbitrarily complex, we should model it with a universal approximator such as a deep network. While using neural networks to model *Q* had been done before, DeepMind also introduced the notion of experience replay for these networks, which let them train DQN models effectively at scale. Experience replay stores observed game outcomes and transitions from past games, and resamples them while training (in addition to training on new games) to ensure that lessons from the past are not forgotten by the network.

Catastrophic Forgetting

Neural networks quickly forget the past. In fact, this phenomenon, termed *catastrophic forgetting*, can occur very rapidly; a few mini-batch updates can be sufficient for the network to forget a complex behavior it previously knew. As a result, without techniques like experience replay that ensure the network always trains on episodes from past matches, it wouldn't be possible to learn complex behaviors.

Designing a training algorithm for deep networks that doesn't suffer from catastrophic forgetting is still a major open problem today. Humans notably don't suffer from catastrophic forgetting; even if you haven't ridden a bike in years, it's likely you still remember how to do so. Creating a neural network that has similar resilience might involve the addition of long-term external memory, along the lines of the Neural Turing machine. Unfortunately, none of the attempts thus far at designing resilient architectures has really worked well.

Policy Learning

In the previous section, you learned about *Q*-learning, which seeks to understand the expected rewards for taking given actions in given environment states. Policy learning is an alternative mathematical framework for learning agent behavior. It introduces the policy function π that assigns a probability to each action that an agent can take in a given state.

Note that a policy is sufficient for defining agent behavior entirely. Given a policy, an agent can act just by sampling a suitable action for the current environment state. Policy learning is convenient, since policies can be learned directly through an algorithm called policy gradient. This algorithm uses a couple mathematical tricks to