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

enable policy gradients to be computed directly via backpropagation for deep networks. The key concept is the *rollout*. Let an agent act in an environment according to its current policy and observe all rewards that come in. Then backpropagate to increase the likelihood of those actions that led to more fruitful rewards. This description is accurate at a high level, but we will see more implementation details later in the chapter.

A policy is often associated with a *value function* V. This function returns the expected discounted reward for following policy π starting from the current state of the environment. V and Q are closely related functions since both provide estimates of future rewards starting from present state, but V does not specify an action to be taken and assumes rather that actions are sampled from π.

Another commonly defined function is the *advantage* A. This function defines the difference in expected reward due to taking a particular action a in a given environment state s, as opposed to following the base policy π. Mathematically, A is defined in terms of Q and V:

$$A(s, a) = Q(s, a) - V(s)$$

The advantage is useful in policy-learning algorithms, since it lets an algorithm quantify how a particular action may have been better suited than the present recommendation of the policy.

Policy Gradient Outside Reinforcement Learning

Although we have introduced policy gradient as a reinforcement learning algorithm, it can equally be viewed as a tool for learning deep networks with nondifferentiable submodules. What does this mean when we unpack the mathematical jargon?

Let's suppose we have a deep network that calls an external program within the network itself. This external program is a black box; it could be a network call or an invocation of a 1970s COBOL routine. How can we learn the rest of the deep network when this module has no gradient?

It turns out that policy gradient can be repurposed to estimate an "effective" gradient for the system. The simple intuition is that multiple "rollouts" can be run, which are used to estimate gradients. Expect to see research over the next few years extending this idea to create large networks with nondifferential modules.

Asynchronous Training

A disadvantage of the policy gradient methods presented in the previous section is that performing the rollout operations requires evaluating agent behavior in some (likely simulated) environment. Most simulators are complicated pieces of software that can't be run on the GPU. As a result, taking a single learning step will require running long CPU-bound calculations. This can lead to unreasonably slow training.

Asynchronous reinforcement learning methods seek to speed up this process by using multiple asynchronous CPU threads to perform rollouts independently. These worker threads will perform rollouts, estimate gradient updates to the policy locally, and then periodically synchronize with the global set of parameters. Empirically, asynchronous training appears to significantly speed up reinforcement learning and allows for fairly sophisticated policies to be learned on laptops. (Without GPUs! The majority of computational power is used on rollouts, so gradient update steps are often not the rate limiting aspect of reinforcement learning training.) The most popular algorithm for asynchronous reinforcement learning currently is the asynchronous actor advantage critic (A3C) algorithm.

CPU or GPU?

GPUs are necessary for most large deep learning applications, but reinforcement learning currently appears to be an exception to this general rule. The reliance of reinforcement learning algorithms to perform many rollouts seems to currently bias reinforcement learning implementations toward multicore CPU systems. It's likely that in specific applications, individual simulators can be ported to work more quickly on GPUs, but CPU-based simulations will likely continue to dominate for the near future.

Limits of Reinforcement Learning

The framework of Markov decision processes is immensely general. For example, behavioral scientists routinely use Markov decision processes to understand and model human decision making. The mathematical generality of this framework has spurred scientists to posit that solving reinforcement learning might spur the creation of artificial general intelligences (AGIs). The stunning success of AlphaGo against Lee Sedol amplified this belief, and indeed research groups such as OpenAI and DeepMind aiming to build AGIs focus much of their efforts on developing new reinforcement learning techniques.

Nonetheless, there are major weaknesses to reinforcement learning as it currently exists. Careful benchmarking work has shown that reinforcement learning techniques are very susceptible to choice of hyperparameters (even by the standards of deep learning, which is already much finickier than other techniques like random forests).