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).

As we have mentioned, reward function engineering is very immature. Humans are capable of internally designing their own reward functions or effectively learning to copy reward functions from observation. Although "inverse reinforcement learning" algorithms that learn reward functions directly have been proposed, these algorithms have many limitations in practice.

In addition to these fundamental limitations, there are still many practical scaling issues. Humans are capable of playing games that combine high-level strategizing with thousands of "micro" moves. For example, master-level play of the strategy game StarCraft (see Figure 8-5) requires sophisticated strategic ploys combined with careful control of hundreds of units. Games can require thousands of local moves to be played to completion. In addition, unlike Go or chess, StarCraft has a "fog of war" where players cannot see the entire game state. This combination of large game state and uncertainty has foiled reinforcement learning attempts on StarCraft. As a result, teams of AI researchers at DeepMind and other groups are focusing serious effort on solving StarCraft with deep reinforcement learning methods. Despite some serious effort, though, the best StarCraft bots remain at amateur level.

Figure 8-5. A collection of subtasks required for playing the real-time strategy game StarCraft. In this game, players must build an army that they can use to defeat the opposing force. Successful StarCraft play requires mastery of resource planning, exploration, and complex strategy. The best computer StarCraft agents remain at amateur level.

In general, there's wide consensus that reinforcement learning is a useful technique that's likely to be deeply influential over the next few decades, but it's also clear that the many practical limitations of reinforcement learning methods will mean that most work will continue to be done in research labs for the near future.

Playing Tic-Tac-Toe

Tic-tac-toe is a simple two-player game. Players place Xs and Os on a 3 × 3 game board until one player succeeds in placing three of her pieces in a row. The first player to do so wins. If neither player succeeds in obtaining three in a row before the board is filled up, the game ends in a draw. Figure 8-6 illustrates a tic-tac-toe game board.

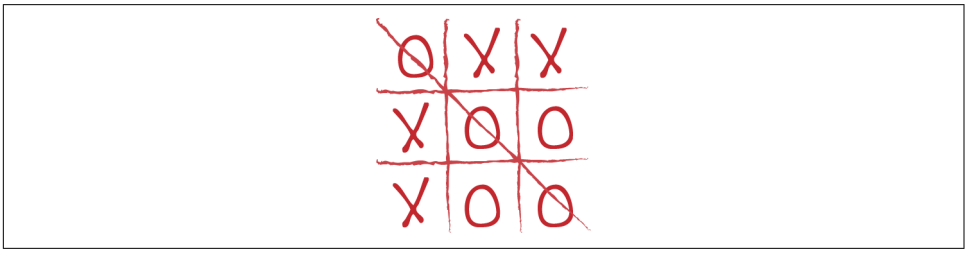

Figure 8-6. A tic-tac-toe game board.

Tic-tac-toe is a nice testbed for reinforcement learning techniques. The game is simple enough that exorbitant amounts of computational power aren't required to train effective agents. At the same time, despite tic-tac-toe's simplicity, learning an effective agent requires considerable sophistication. The TensorFlow code for this section is arguably the most sophisticated example found in this book. We will walk you through the design of a TensorFlow tic-tac-toe asynchronous reinforcement learning agent in the remainder of this section.

Object Orientation

The code we've introduced thus far in this book has primarily consisted of scripts augmented by smaller helper functions. In this chapter, however, we will swap to an object-oriented programming style. This style of programming might be new to you, especially if you hail from the scientific world rather than from the tech world. Briefly, an object-oriented program defines *objects* that model aspects of the world. For example, you might want to define `Environment` or `Agent` or `Reward` objects that directly correspond to these mathematical concepts. A *class* is a template for objects that can be used to *instantiate* (or create) many new objects. For example, you will shortly see an `Environment` class definition we will use to define many particular `Environment` objects.

Object orientation is particularly powerful for building complex systems, so we will use it to simplify the design of our reinforcement learning system. In practice, your real-world deep learning (or reinforcement learning) systems will likely need to be object oriented as well, so we encourage taking some time to master object-oriented design. There are many superb books that cover the fundamentals of object-oriented design, and we recommend that you check them out as necessary.