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.

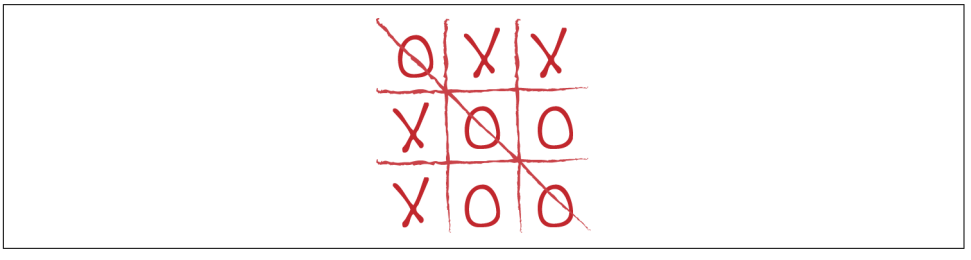

Figure 8-6. A tic-tac-toe game board.

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.

Abstract Environment

Let's start by defining an abstract `Environment` object that encodes the state of a system in a list of NumPy objects (Example 8-1). This `Environment` object is quite general (adapted from DeepChem's reinforcement learning engine) so it can easily serve as a template for other reinforcement learning projects you might seek to implement.

Example 8-1. This class defines a template for constructing new environments

```
class Environment(object):
  """An environment in which an actor performs actions to accomplish a task.

  An environment has a current state, which is represented as either a single NumPy
  array, or optionally a list of NumPy arrays.  When an action is taken, that causes
  the state to be updated.  Exactly what is meant by an "action" is defined by each
  subclass.  As far as this interface is concerned, it is simply an arbitrary object.
  The environment also computes a reward for each action, and reports when the task
  has been terminated (meaning that no more actions may be taken).
  """

  def __init__(self, state_shape, n_actions, state_dtype=None):
    """Subclasses should call the superclass constructor in addition to doing their
       own initialization."""
    self.state_shape = state_shape
    self.n_actions = n_actions
    if state_dtype is None:
      # Assume all arrays are float32.
      if isinstance(state_shape[0], collections.Sequence):
        self.state_dtype = [np.float32] * len(state_shape)
      else:
        self.state_dtype = np.float32
    else:
      self.state_dtype = state_dtype
```

Tic-Tac-Toe Environment

We need to specialize the `Environment` class to create a `TicTacToeEnvironment` suitable for our needs. To do this, we construct a *subclass* of `Environment` that adds on more features, while retaining the core functionality of the original *superclass*. In Example 8-2, we define `TicTacToeEnvironment` as a subclass of `Environment` that adds details specific to tic-tac-toe.

Example 8-2. The TicTacToeEnvironment class defines a template for constructing new tic-tac-toe environments

```
class TicTacToeEnvironment(dc.rl.Environment):
  """
  Play tictactoe against a randomly acting opponent
```