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
```

```
  """
  X = np.array([1.0, 0.0])
  O = np.array([0.0, 1.0])
  EMPTY = np.array([0.0, 0.0])

  ILLEGAL_MOVE_PENALTY = -3.0
  LOSS_PENALTY = -3.0
  NOT_LOSS = 0.1
  DRAW_REWARD = 5.0
  WIN_REWARD = 10.0

  def __init__(self):
    super(TicTacToeEnvironment, self).__init__([(3, 3, 2)], 9)
    self.terminated = None
    self.reset()
```

The first interesting tidbit to note here is that we define the board state as a NumPy array of shape `(3, 3, 2)`. We use a one-hot encoding of `X` and `O` (one-hot encodings aren't only useful for natural language processing!).

The second important thing to note is that the environment explicitly defines the reward function by setting penalties for illegal moves and losses, and rewards for draws and wins. This snippet powerfully illustrates the arbitrary nature of reward function engineering. Why these particular numbers?

Empirically, these choices appear to result in stable behavior, but we encourage you to experiment with alternate reward settings to observe results. In this implementation, we specify that the agent always plays `X`, but randomize whether `X` or `O` goes first. The function `get_O_move()` simply places an `O` on a random open tile on the game board. `TicTacToeEnvironment` encodes an opponent that plays `O` while always selecting a random move. The `reset()` function simply clears the board, and places an `O` tile randomly if `O` is going first during this game. See Example 8-3.

Example 8-3. More methods from the TicTacToeEnvironment class

```
def reset(self):
  self.terminated = False
  self.state = [np.zeros(shape=(3, 3, 2), dtype=np.float32)]

  # Randomize who goes first
  if random.randint(0, 1) == 1:
    move = self.get_O_move()
    self.state[0][move[0]][move[1]] = TicTacToeEnvironment.O

def get_O_move(self):
  empty_squares = []
  for row in range(3):
    for col in range(3):
      if np.all(self.state[0][row][col] == TicTacToeEnvironment.EMPTY):
```

```
      empty_squares.append((row, col))
  return random.choice(empty_squares)
```

The utility function `game_over()` reports that the game has ended if all tiles are filled. `check_winner()` checks whether the specified player has achieved three in a row and won the game (Example 8-4).

Example 8-4. Utility methods from the TicTacToeEnvironment class for detecting when the game has ended and who won

```
def check_winner(self, player):
  for i in range(3):
    row = np.sum(self.state[0][i][:], axis=0)
    if np.all(row == player * 3):
      return True
    col = np.sum(self.state[0][:][i], axis=0)
    if np.all(col == player * 3):
      return True

  diag1 = self.state[0][0][0] + self.state[0][1][1] + self.state[0][2][2]
  if np.all(diag1 == player * 3):
    return True
  diag2 = self.state[0][0][2] + self.state[0][1][1] + self.state[0][2][0]
  if np.all(diag2 == player * 3):
    return True
  return False

def game_over(self):
  for i in range(3):
    for j in range(3):
      if np.all(self.state[0][i][j] == TicTacToeEnvironment.EMPTY):
        return False
  return True
```

In our implementation, an action is simply a number between 0 and 8 specifying the tile on which the `X` tile is placed. The `step()` method checks whether this tile is occupied (returning a penalty if so), then places the tile. If `X` has won, a reward is returned. Else, the random `O` opponent is allowed to make a move. If `O` won, then a penalty is returned. If the game has ended as a draw, then a penalty is returned. Else, the game continues with a `NOT_LOSS` reward. See Example 8-5.

Example 8-5. This method performs a step of the simulation

```
def step(self, action):
  self.state = copy.deepcopy(self.state)
  row = action // 3
  col = action % 3
```

```
# Illegal move -- the square is not empty
if not np.all(self.state[0][row][col] == TicTacToeEnvironment.EMPTY):
  self.terminated = True
  return TicTacToeEnvironment.ILLEGAL_MOVE_PENALTY

# Move X
self.state[0][row][col] = TicTacToeEnvironment.X

# Did X Win
if self.check_winner(TicTacToeEnvironment.X):
  self.terminated = True
  return TicTacToeEnvironment.WIN_REWARD

if self.game_over():
  self.terminated = True
  return TicTacToeEnvironment.DRAW_REWARD

move = self.get_O_move()
self.state[0][move[0]][move[1]] = TicTacToeEnvironment.O

# Did O Win
if self.check_winner(TicTacToeEnvironment.O):
  self.terminated = True
  return TicTacToeEnvironment.LOSS_PENALTY

if self.game_over():
  self.terminated = True
  return TicTacToeEnvironment.DRAW_REWARD

return TicTacToeEnvironment.NOT_LOSS
```

The Layer Abstraction

Running an asynchronous reinforcement learning algorithm such as A3C requires that each thread have access to a separate copy of the policy model. These copies of the model have to be periodically re-synced with one another for training to proceed. What is the easiest way we can construct multiple copies of the TensorFlow graph that we can distribute to each thread?

One simple possibility is to create a function that creates a copy of the model in a separate TensorFlow graph. This approach works well, but gets to be a little messy, especially for sophisticated networks. Using a little bit of object orientation can significantly simplify this process. Since our reinforcement learning code is adapted from the DeepChem library, we use a simplified version of the TensorGraph framework from DeepChem (see *https://deepchem.io* for information and docs). This framework is similar to other high-level TensorFlow frameworks such as Keras. The core abstraction in all such models is the introduction of a `Layer` object that encapsulates a portion of a deep network.