```python
  actions_matrix = []
  for action in actions:
    a = np.zeros(n_actions)
    a[action] = 1.0
    actions_matrix.append(a)

  # Rearrange the states into the proper set of arrays.
  state_arrays = [[] for i in range(len(self.features))]
  for state in states:
    for j in range(len(state)):
      state_arrays[j].append(state[j])

  # Build the feed dict and apply gradients.
  feed_dict = {}
  for f, s in zip(self.features, state_arrays):
    feed_dict[f.out_tensor] = s
  feed_dict[self.rewards.out_tensor] = discounted_rewards
  feed_dict[self.actions.out_tensor] = actions_matrix
  feed_dict[self.advantages.out_tensor] = advantages
  feed_dict[self.global_step] = step_count
  self.a3c._session.run(self.train_op, feed_dict=feed_dict)
```

The `Worker.run()` method performs the training step for the `Worker`, relying on `process_rollouts()` to issue the actual call to `self.a3c._session.run()` under the hood (Example 8-29).

Example 8-29. The run() method is the top level invocation for Worker

```python
def run(self, step_count, total_steps):
  with self.graph._get_tf("Graph").as_default():
    while step_count[0] < total_steps:
      self.a3c._session.run(self.update_local_variables)
      states, actions, rewards, values = self.create_rollout()
      self.process_rollout(states, actions, rewards, values, step_count[0])
      step_count[0] += len(actions)
```

Training the Policy

The `A3C.fit()` method brings together all the disparate pieces introduced to train the model. The `fit()` method takes the responsibility for spawning `Worker` threads using the Python `threading` library. Since each `Worker` takes responsibility for training itself, the `fit()` method simply is responsible for periodically checkpointing the trained model to disk. See Example 8-30.

Example 8-30. The fit() method brings everything together and runs the A3C training algorithm

```
def fit(self,
        total_steps,
        max_checkpoints_to_keep=5,
        checkpoint_interval=600,
        restore=False):
  """Train the policy.

  Parameters
  ----------
  total_steps: int
    the total number of time steps to perform on the environment, across all
    rollouts on all threads
  max_checkpoints_to_keep: int
    the maximum number of checkpoint files to keep.  When this number is
    reached, older files are deleted.
  checkpoint_interval: float
    the time interval at which to save checkpoints, measured in seconds
  restore: bool
    if True, restore the model from the most recent checkpoint and continue
    training from there.  If False, retrain the model from scratch.
  """
  with self._graph._get_tf("Graph").as_default():
    step_count = [0]
    workers = []
    threads = []
    for i in range(multiprocessing.cpu_count()):
      workers.append(Worker(self, i))
    self._session.run(tf.global_variables_initializer())
    if restore:
      self.restore()
    for worker in workers:
      thread = threading.Thread(
          name=worker.scope,
          target=lambda: worker.run(step_count, total_steps))
      threads.append(thread)
      thread.start()
    variables = tf.get_collection(
        tf.GraphKeys.GLOBAL_VARIABLES, scope="global")
    saver = tf.train.Saver(variables, max_to_keep=max_checkpoints_to_keep)
    checkpoint_index = 0
    while True:
      threads = [t for t in threads if t.isAlive()]
      if len(threads) > 0:
        threads[0].join(checkpoint_interval)
      checkpoint_index += 1
      saver.save(
          self._session, self._graph.save_file, global_step=checkpoint_index)
      if len(threads) == 0:
        break
```

Challenge for the Reader

We strongly encourage you to try training tic-tac-toe models for yourself! Note that this example is more involved than other examples in the book, and will require greater computational power. We recommend a machine with at least a few CPU cores. This requirement isn't too onerous; a good laptop should suffice. Try using a tool like `htop` to check that the code is indeed multithreaded. See how good a model you can train! You should be able to beat the random baseline most of the time, but this basic implementation won't give you a model that always wins. We recommend exploring the RL literature and expanding upon the base implementation to see how well you can do.

Review

In this chapter, we introduced you to the core concepts of reinforcement learning (RL). We walked you through some recent successes of RL methods on ATARI, upside-down helicopter flight, and computer Go. We then taught you about the mathematical framework of Markov decision processes. We brought it together with a detailed case study walking you through the construction of a tic-tac-toe agent. This algorithm uses a sophisticated training method, A3C, that makes use of multiple CPU cores to speed up training. In Chapter 9, you'll learn more about training models with multiple GPUs.