Example 8-23. This snippet from A3CLoss defines the policy loss

```
policy_loss = -tf.reduce_mean(
    advantage * tf.reduce_sum(action * log_prob, axis=1))
```

The value loss computes the difference between our estimate of V (`reward`) and the actual value of V observed (`value`). Note the use of the L^2 loss here (Example 8-24).

Example 8-24. This snippet from A3CLoss defines the value loss

```
value_loss = tf.reduce_mean(tf.square(reward - value))
```

The entropy term is an addition that encourages the policy to explore further by adding some noise. This term is effectively a form of regularization for A3C networks. The final loss computed by `A3CLoss` is a linear combination of these component losses. See Example 8-25.

Example 8-25. This snippet from A3CLoss defines an entropy term added to the loss

```
entropy = -tf.reduce_mean(tf.reduce_sum(prob * log_prob, axis=1))
```

Defining Workers

Thus far, you've seen how the policy network is constructed, but you haven't yet seen how the asynchronous training procedure is implemented. Conceptually, asynchronous training consists of individual workers running gradient descent on locally simulated game rollouts and contributing learned knowledge back to a global set of weights periodically. Continuing our object-oriented design, let's introduce the `Worker` class.

Each `Worker` instance holds a copy of the model that's trained asynchronously on a separate thread (Example 8-26). Note that `a3c.build_graph()` is used to construct a local copy of the TensorFlow computation graph for the thread in question. Take special note of `local_vars` and `global_vars` here. We need to make sure to train only the variables associated with this worker's copy of the policy and not with the global copy of the variables (which is used to share information across worker threads). As a result `gradients` uses `tf.gradients` to take gradients of the loss with respect to only `local_vars`.

Example 8-26. The Worker class implements the computation performed by each thread

```
class Worker(object):
  """A Worker object is created for each training thread."""

  def __init__(self, a3c, index):
    self.a3c = a3c
    self.index = index
    self.scope = "worker%d" % index
    self.env = copy.deepcopy(a3c._env)
    self.env.reset()
    (self.graph, self.features, self.rewards, self.actions, self.action_prob,
     self.value, self.advantages) = a3c.build_graph(
        a3c._graph._get_tf("Graph"), self.scope, None)
    with a3c._graph._get_tf("Graph").as_default():
      local_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                                     self.scope)
      global_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                                      "global")
      gradients = tf.gradients(self.graph.loss.out_tensor, local_vars)
      grads_and_vars = list(zip(gradients, global_vars))
      self.train_op = a3c._graph._get_tf("Optimizer").apply_gradients(
          grads_and_vars)
      self.update_local_variables = tf.group(
          * [tf.assign(v1, v2) for v1, v2 in zip(local_vars, global_vars)])
      self.global_step = self.graph.get_global_step()
```

Worker rollouts

Each `Worker` is responsible for simulating game rollouts locally. The `create_rollout()` method uses `session.run` to fetch action probabilities from the TensorFlow graph (Example 8-27). It then samples an action from this policy using `np.random.choice`, weighted by the per-class probabilities. The reward for the action taken is computed from `TicTacToeEnvironment` via a call to `self.env.step(action)`.

Example 8-27. The create_rollout method simulates a game rollout locally

```
def create_rollout(self):
  """Generate a rollout."""
  n_actions = self.env.n_actions
  session = self.a3c._session
  states = []
  actions = []
  rewards = []
  values = []

  # Generate the rollout.
  for i in range(self.a3c.max_rollout_length):
    if self.env.terminated:
```

```
          break
        state = self.env.state
        states.append(state)
        feed_dict = self.create_feed_dict(state)
        results = session.run(
            [self.action_prob.out_tensor, self.value.out_tensor],
            feed_dict=feed_dict)
        probabilities, value = results[:2]
        action = np.random.choice(np.arange(n_actions), p=probabilities[0])
        actions.append(action)
        values.append(float(value))
        rewards.append(self.env.step(action))

      # Compute an estimate of the reward for the rest of the episode.
      if not self.env.terminated:
        feed_dict = self.create_feed_dict(self.env.state)
        final_value = self.a3c.discount_factor * float(
            session.run(self.value.out_tensor, feed_dict))
      else:
        final_value = 0.0
      values.append(final_value)
      if self.env.terminated:
        self.env.reset()
      return states, actions, np.array(rewards), np.array(values)
```

The `process_rollouts()` method does preprocessing needed to compute discounted rewards, values, actions, and advantages (Example 8-28).

Example 8-28. The process_rollout method computes rewards, values, actions, and advantages and then takes a gradient descent step against the loss

```
def process_rollout(self, states, actions, rewards, values, step_count):
    """Train the network based on a rollout."""

    # Compute the discounted rewards and advantages.
    if len(states) == 0:
      # Rollout creation sometimes fails in multithreaded environment.
      # Don't process if malformed
      print("Rollout creation failed. Skipping")
      return

    discounted_rewards = rewards.copy()
    discounted_rewards[-1] += values[-1]
    advantages = rewards - values[:-1] + self.a3c.discount_factor * np.array(
        values[1:])
    for j in range(len(rewards) - 1, 0, -1):
      discounted_rewards[j-1] += self.a3c.discount_factor * discounted_rewards[j]
      advantages[j-1] += (
          self.a3c.discount_factor * self.a3c.advantage_lambda * advantages[j])
     # Convert the actions to one-hot.
    n_actions = self.env.n_actions
```

```
  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

```
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.