```
  Parameters
  ----------
  obj: str
    If "Graph", returns tf.Graph instance. If "Optimizer", returns the
    optimizer. If "train_op", returns the train operation. If "GlobalStep" returns
    the global step.
  Returns
  -------
  TensorFlow Object
  """

  if obj in self.tensor_objects and self.tensor_objects[obj] is not None:
    return self.tensor_objects[obj]
  if obj == "Graph":
    self.tensor_objects["Graph"] = tf.Graph()
  elif obj == "Optimizer":
    self.tensor_objects["Optimizer"] = tf.train.AdamOptimizer(
        learning_rate=self.learning_rate,
        beta1=0.9,
        beta2=0.999,
        epsilon=1e-7)
  elif obj == "GlobalStep":
    with self._get_tf("Graph").as_default():
      self.tensor_objects["GlobalStep"] = tf.Variable(0, trainable=False)
  return self._get_tf(obj)
```

Finally, the `restore()` method restores a saved `TensorGraph` from disk (Example 8-15). (As you will see later, the `TensorGraph` is saved automatically during training.)

Example 8-15. Restore a trained model from disk

```
def restore(self):
  """Reload the values of all variables from the most recent checkpoint file."""
  if not self.built:
    self.build()
  last_checkpoint = tf.train.latest_checkpoint(self.model_dir)
  if last_checkpoint is None:
    raise ValueError("No checkpoint found")
  with self._get_tf("Graph").as_default():
    saver = tf.train.Saver()
    saver.restore(self.session, last_checkpoint)
```

The A3C Algorithm

In this section you will learn how to implement A3C, the asynchronous reinforcement learning algorithm you saw earlier in the chapter. A3C is a significantly more complex training algorithm than those you have seen previously. The algorithm requires running gradient descent in multiple threads, interspersed with game rollout code, and updating learned weights asynchronously. As a result of this extra complexity, we will define the A3C algorithm in an object-oriented fashion. Let's start by defining an A3C object.

The A3C class implements the A3C algorithm (Example 8-16). A few extra bells and whistles are added onto the basic algorithm to encourage learning, notably an entropy term and support for generalized advantage estimation. We won't cover all of these details, but encourage you to follow references into the research literature (listed in the documentation) to understand more.

Example 8-16. Define the A3C class encapsulating the asynchronous A3C training algorithm

```
class A3C(object):
  """
  Implements the Asynchronous Advantage Actor-Critic (A3C) algorithm.

  The algorithm is described in Mnih et al, "Asynchronous Methods for Deep
  Reinforcement Learning" (https://arxiv.org/abs/1602.01783).  This class
  requires the policy to output two quantities: a vector giving the probability
  of taking each action, and an estimate of the value function for the current
  state.  It optimizes both outputs at once using a loss that is the sum of three
  terms:

  1. The policy loss, which seeks to maximize the discounted reward for each action.
  2. The value loss, which tries to make the value estimate match the actual
     discounted reward that was attained at each step.
  3. An entropy term to encourage exploration.

  This class only supports environments with discrete action spaces, not
  continuous ones.  The "action" argument passed to the environment is an
  integer, giving the index of the action to perform.

  This class supports Generalized Advantage Estimation as described in Schulman
  et al., "High-Dimensional Continuous Control Using Generalized Advantage
  Estimation" (https://arxiv.org/abs/1506.02438).  This is a method of trading
  off bias and variance in the advantage estimate, which can sometimes improve
  the rate of convergence.  Use the advantage_lambda parameter to adjust the
  tradeoff.
  """
  self._env = env
  self.max_rollout_length = max_rollout_length
  self.discount_factor = discount_factor
  self.advantage_lambda = advantage_lambda
  self.value_weight = value_weight
  self.entropy_weight = entropy_weight
  self._optimizer = None
  (self._graph, self._features, self._rewards, self._actions,
   self._action_prob, self._value, self._advantages) = self.build_graph(
       None, "global", model_dir)
```

```
  with self._graph._get_tf("Graph").as_default():
    self._session = tf.Session()
```

The heart of the A3C class lies in the `build_graph()` method (Example 8-17), which constructs a `TensorGraph` instance (underneath which lies a TensorFlow computation graph) encoding the policy learned by the model. Notice how succinct this definition is compared with others you have seen previously! There are many advantages to using object orientation.

Example 8-17. This method builds the computation graph for the A3C algorithm. Note that the policy network is defined here using the Layer abstractions you saw previously.

```
def build_graph(self, tf_graph, scope, model_dir):
  """Construct a TensorGraph containing the policy and loss calculations."""
  state_shape = self._env.state_shape
  features = []
  for s in state_shape:
    features.append(Input(shape=[None] + list(s), dtype=tf.float32))
  d1 = Flatten(in_layers=features)
  d2 = Dense(
      in_layers=[d1],
      activation_fn=tf.nn.relu,
      normalizer_fn=tf.nn.l2_normalize,
      normalizer_params={"dim": 1},
      out_channels=64)
  d3 = Dense(
      in_layers=[d2],
      activation_fn=tf.nn.relu,
      normalizer_fn=tf.nn.l2_normalize,
      normalizer_params={"dim": 1},
      out_channels=32)
  d4 = Dense(
      in_layers=[d3],
      activation_fn=tf.nn.relu,
      normalizer_fn=tf.nn.l2_normalize,
      normalizer_params={"dim": 1},
      out_channels=16)
  d4 = BatchNorm(in_layers=[d4])
  d5 = Dense(in_layers=[d4], activation_fn=None, out_channels=9)
  value = Dense(in_layers=[d4], activation_fn=None, out_channels=1)
  value = Squeeze(squeeze_dims=1, in_layers=[value])
  action_prob = SoftMax(in_layers=[d5])

  rewards = Input(shape=(None,))
  advantages = Input(shape=(None,))
  actions = Input(shape=(None, self._env.n_actions))
  loss = A3CLoss(
      self.value_weight,
      self.entropy_weight,
      in_layers=[rewards, actions, action_prob, value, advantages])
```

```
  graph = TensorGraph(
      batch_size=self.max_rollout_length,
      graph=tf_graph,
      model_dir=model_dir)
  for f in features:
    graph._add_layer(f)
  graph.add_output(action_prob)
  graph.add_output(value)
  graph.set_loss(loss)
  graph.set_optimizer(self._optimizer)
  with graph._get_tf("Graph").as_default():
    with tf.variable_scope(scope):
      graph.build()
  return graph, features, rewards, actions, action_prob, value, advantages
```

There's a lot of code in this example. Let's break it down into multiple examples and discuss more carefully. Example 8-18 takes the array encoding of the `TicTacToeEnvironment` and feeds it into the `Input` instances for the graph directly.

Example 8-18. This snippet from the build_graph() method feeds in the array encoding of TicTacToeEnvironment

```
state_shape = self._env.state_shape
features = []
for s in state_shape:
  features.append(Input(shape=[None] + list(s), dtype=tf.float32))
```

Example 8-19 shows the code used to construct inputs for rewards from the environment, advantages observed, and actions taken.

Example 8-19. This snippet from the build_graph() method defines Input objects for rewards, advantages, and actions

```
rewards = Input(shape=(None,))
advantages = Input(shape=(None,))
actions = Input(shape=(None, self._env.n_actions))
```

The policy network is responsible for learning the policy. In Example 8-20, the input board state is first flattened into an input feature vector. A series of fully connected (or `Dense`) transformations are applied to the flattened board. At the very end, a `Softmax` layer is used to predict action probabilities from `d5` (note that `out_channels` is set to 9, one for each possible move on the tic-tac-toe board).

Example 8-20. This snippet from the build_graph() method defines the policy network

```
d1 = Flatten(in_layers=features)
d2 = Dense(
    in_layers=[d1],
```

```
    activation_fn=tf.nn.relu,
    normalizer_fn=tf.nn.l2_normalize,
    normalizer_params={"dim": 1},
    out_channels=64)
d3 = Dense(
    in_layers=[d2],
    activation_fn=tf.nn.relu,
    normalizer_fn=tf.nn.l2_normalize,
    normalizer_params={"dim": 1},
    out_channels=32)
d4 = Dense(
    in_layers=[d3],
    activation_fn=tf.nn.relu,
    normalizer_fn=tf.nn.l2_normalize,
    normalizer_params={"dim": 1},
    out_channels=16)
d4 = BatchNorm(in_layers=[d4])
d5 = Dense(in_layers=[d4], activation_fn=None, out_channels=9)
value = Dense(in_layers=[d4], activation_fn=None, out_channels=1)
value = Squeeze(squeeze_dims=1, in_layers=[value])
action_prob = SoftMax(in_layers=[d5])
```

Is Feature Engineering Dead?

In this section, we feed the raw tic-tac-toe game board into TensorFlow for training the policy. However, it's important to note that for more complex games than tic-tac-toe, this may not yield satisfactory results. One of the lesser known facts about AlphaGo is that DeepMind performs sophisticated feature engineering to extract "interesting" patterns of Go pieces upon the board to make AlphaGo's learning easier. (This fact is tucked away into the supplemental information of DeepMind's paper.)

The fact remains that reinforcement learning (and deep learning methods broadly) often still need human-guided feature engineering to extract meaningful information before learning algorithms can learn effective policies and models. It's likely that as more computational power becomes available through hardware advances, this need for feature engineering will be reduced, but for the near term, plan on manually extracting information about your systems as needed for performance.

The A3C Loss Function

We now have the object-oriented machinery set in place to define a loss for the A3C policy network. This loss function will itself be implemented as a `Layer` object (it's a convenient abstraction that all parts of the deep architecture are simply layers). The `A3CLoss` object implements a mathematical loss consisting of the sum of three terms: