```python
  # 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.

A `Layer` is a portion of a TensorFlow graph that accepts a list `in_layers` of input layers. In this abstraction, a deep architecture consists of a *directed graph* of layers. Directed graphs are similar to the undirected graphs you saw in Chapter 6, but have directions on their edges. In this case, the `in_layers` have edges to the new `Layer`, with the direction pointing toward the new layer. You will learn more about this concept in the next section.

We use `tf.register_tensor_conversion_function`, a utility that allows arbitrary classes to register themselves as tensor convertible. This registration will mean that a `Layer` can be converted into a TensorFlow tensor via a call to `tf.convert_to_tensor`. The `_get_input_tensors()` private method is a utility that uses `tf.convert_to_tensor` to transform input layers into input tensors. Each `Layer` is responsible for implementing a `create_tensor()` method that specifies the operations to add to the TensorFlow computational graph. See Example 8-6.

Example 8-6. The Layer object is the fundamental abstraction in object-oriented deep architectures. It encapsulates a part of the netwok such as a fully connected layer or a convolutional layer. This example defines a generic superclass for all such layers.

```
class Layer(object):

  def __init__(self, in_layers=None, **kwargs):
    if "name" in kwargs:
      self.name = kwargs["name"]
    else:
      self.name = None
    if in_layers is None:
      in_layers = list()
    if not isinstance(in_layers, Sequence):
      in_layers = [in_layers]
    self.in_layers = in_layers
    self.variable_scope = ""
    self.tb_input = None

  def create_tensor(self, in_layers=None, **kwargs):
    raise NotImplementedError("Subclasses must implement for themselves")

  def _get_input_tensors(self, in_layers):
    """Get the input tensors to his layer.

    Parameters
    ----------
    in_layers: list of Layers or tensors
      the inputs passed to create_tensor().  If None, this layer's inputs will
      be used instead.
    """
    if in_layers is None:
      in_layers = self.in_layers
```

```
    if not isinstance(in_layers, Sequence):
      in_layers = [in_layers]
    tensors = []
    for input in in_layers:
      tensors.append(tf.convert_to_tensor(input))
    return tensors

def _convert_layer_to_tensor(value, dtype=None, name=None, as_ref=False):
  return tf.convert_to_tensor(value.out_tensor, dtype=dtype, name=name)

tf.register_tensor_conversion_function(Layer, _convert_layer_to_tensor)
```

The preceding description is abstract, but in practice easy to use. Example 8-7 shows a `Squeeze` layer that wraps `tf.squeeze` with a `Layer` (you will find this class convenient later). Note that `Squeeze` is a subclass of `Layer`.

Example 8-7. The Squeeze layer squeezes its input

```
class Squeeze(Layer):

  def __init__(self, in_layers=None, squeeze_dims=None, **kwargs):
    self.squeeze_dims = squeeze_dims
    super(Squeeze, self).__init__(in_layers, **kwargs)

  def create_tensor(self, in_layers=None, **kwargs):
    inputs = self._get_input_tensors(in_layers)
    parent_tensor = inputs[0]
    out_tensor = tf.squeeze(parent_tensor, squeeze_dims=self.squeeze_dims)
    self.out_tensor = out_tensor
    return out_tensor
```

The `Input` layer wraps placeholders for convenience (Example 8-8). Note that the `Layer.create_tensor` method must be invoked for each layer we use in order to construct a TensorFlow computational graph.

Example 8-8. The Input layer adds placeholders to the computation graph

```
class Input(Layer):

  def __init__(self, shape, dtype=tf.float32, **kwargs):
    self._shape = tuple(shape)
    self.dtype = dtype
    super(Input, self).__init__(**kwargs)

  def create_tensor(self, in_layers=None, **kwargs):
    if in_layers is None:
      in_layers = self.in_layers
    out_tensor = tf.placeholder(dtype=self.dtype, shape=self._shape)
```

```
    self.out_tensor = out_tensor
    return out_tensor
```

tf.keras and tf.estimator

TensorFlow has now integrated the popular Keras object-oriented frontend into the core TensorFlow library. Keras includes a `Layer` class definition that closely matches the `Layer` objects you've just learned about in this section. In fact, the `Layer` objects here were adapted from the DeepChem library, which in turn adapted them from an earlier version of Keras.

It's worth noting, though, that `tf.keras` has not yet become the standard higher-level interface to TensorFlow. The `tf.estimator` module provides an alternative (albeit less rich) high-level interface to raw TensorFlow.

Regardless of which frontend eventually becomes standard, we think that understanding the fundamental design principles for building your own frontend is instructive and worthwhile. You might need to build a new system for your organization that requires an alternative design, so a solid grasp of design principles will serve you well.

Defining a Graph of Layers

We mentioned briefly in the previous section that a deep architecture could be visualized as a directed graph of `Layer` objects. In this section, we transform this intuition into the `TensorGraph` object. These objects are responsible for constructing the underlying TensorFlow computation graph.

A `TensorGraph` object is responsible for maintaining a `tf.Graph`, a `tf.Session`, and a list of layers (`self.layers`) internally (Example 8-9). The directed graph is represented implicitly, by the `in_layers` belonging to each `Layer` object. `TensorGraph` also contains utilities for saving this `tf.Graph` instance to disk and consequently assigns itself a directory (using `tempfile.mkdtemp()` if none is specified) to store checkpoints of the weights associated with its underlying TensorFlow graph.

Example 8-9. The TensorGraph contains a graph of layers; TensorGraph objects can be thought of as the "model" object holding the deep architecture you want to train

```
class TensorGraph(object):

  def __init__(self,
               batch_size=100,
               random_seed=None,
               graph=None,
```