```
    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,
```

```
                 learning_rate=0.001,
                 model_dir=None,
                 **kwargs):
    """
    Parameters
    ----------
    batch_size: int
      default batch size for training and evaluating
    graph: tensorflow.Graph
      the Graph in which to create Tensorflow objects.  If None, a new Graph
      is created.
    learning_rate: float or LearningRateSchedule
      the learning rate to use for optimization
    kwargs
    """

    # Layer Management
    self.layers = dict()
    self.features = list()
    self.labels = list()
    self.outputs = list()
    self.task_weights = list()
    self.loss = None
    self.built = False
    self.optimizer = None
    self.learning_rate = learning_rate

    # Singular place to hold Tensor objects which don't serialize
    # See TensorGraph._get_tf() for more details on lazy construction
    self.tensor_objects = {
        "Graph": graph,
        #"train_op": None,
    }
    self.global_step = 0
    self.batch_size = batch_size
    self.random_seed = random_seed
    if model_dir is not None:
      if not os.path.exists(model_dir):
        os.makedirs(model_dir)
    else:
      model_dir = tempfile.mkdtemp()
      self.model_dir_is_temp = True
    self.model_dir = model_dir
    self.save_file = "%s/%s" % (self.model_dir, "model")
    self.model_class = None
```

The private method `_add_layer` does bookkeeping work to add a new `Layer` obect to the `TensorGraph` (Example 8-10).

Example 8-10. The _add_layer method adds a new Layer object

```
def _add_layer(self, layer):
  if layer.name is None:
    layer.name = "%s_%s" % (layer.__class__.__name__, len(self.layers) + 1)
  if layer.name in self.layers:
    return
  if isinstance(layer, Input):
    self.features.append(layer)
  self.layers[layer.name] = layer
  for in_layer in layer.in_layers:
    self._add_layer(in_layer)
```

The layers in a `TensorGraph` must form a directed acyclic graph (there can be no loops in the graph). As a result, we can topologically sort these layers. Intuitively, a topological sort "orders" the layers in the graph so that each `Layer` object's `in_layers` precede it in the ordered list. This topological sort is necessary to make sure all input layers to a given layer are added to the graph before the layer itself (Example 8-11).

Example 8-11. The topsort method orders the layers in the TensorGraph

```
def topsort(self):

  def add_layers_to_list(layer, sorted_layers):
    if layer in sorted_layers:
      return
    for in_layer in layer.in_layers:
      add_layers_to_list(in_layer, sorted_layers)
    sorted_layers.append(layer)

  sorted_layers = []
  for l in self.features + self.labels + self.task_weights + self.outputs:
    add_layers_to_list(l, sorted_layers)
  add_layers_to_list(self.loss, sorted_layers)
  return sorted_layers
```

The `build()` method takes the responsibility of populating the `tf.Graph` instance by calling `layer.create_tensor` for each layer in topological order (Example 8-12).

Example 8-12. The build method populates the underlying TensorFlow graph

```
def build(self):
  if self.built:
    return
  with self._get_tf("Graph").as_default():
    self._training_placeholder = tf.placeholder(dtype=tf.float32, shape=())
    if self.random_seed is not None:
      tf.set_random_seed(self.random_seed)
    for layer in self.topsort():
```

```
      with tf.name_scope(layer.name):
        layer.create_tensor(training=self._training_placeholder)
    self.session = tf.Session()

    self.built = True
```

The method `set_loss()` adds a loss for training to the graph. `add_output()` specifies that the layer in question might be fetched from the graph. `set_optimizer()` specifies the optimizer used for training (Example 8-13).

Example 8-13. These methods add necessary losses, outputs, and optimizers to the computation graph

```
def set_loss(self, layer):
  self._add_layer(layer)
  self.loss = layer

def add_output(self, layer):
  self._add_layer(layer)
  self.outputs.append(layer)

def set_optimizer(self, optimizer):
  """Set the optimizer to use for fitting."""
  self.optimizer = optimizer
```

The method `get_layer_variables()` is used to fetch the learnable `tf.Variable` objects created by a layer. The private method `_get_tf` is used to fetch the `tf.Graph` and optimizer instances underpinning the `TensorGraph`. `get_global_step` is a convenience method for fetching the current step in the training process (starting from 0 at construction). See Example 8-14.

Example 8-14. Fetch the learnable variables associated with each layer

```
def get_layer_variables(self, layer):
  """Get the list of trainable variables in a layer of the graph."""
  if not self.built:
    self.build()
  with self._get_tf("Graph").as_default():
    if layer.variable_scope == "":
      return []
    return tf.get_collection(
        tf.GraphKeys.TRAINABLE_VARIABLES, scope=layer.variable_scope)

def get_global_step(self):
  return self._get_tf("GlobalStep")

def _get_tf(self, obj):
  """Fetches underlying TensorFlow primitives.
```

```
    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