```
  return result
```

Deep Dive on the Architecture

The architecture for the network is a standard multilayer convnet, similar to a more complicated version of the LeNet5 architecture you saw in Chapter 6. The `inference()` method constructs the architecture (Example 9-2). This convolutional architecture follows a relatively standard architecture, with convolutional layers interspersed with local normalization layers.

Example 9-2. This function builds the Cifar10 architecture

```
def inference(images):
  """Build the CIFAR10 model.

  Args:
    images: Images returned from distorted_inputs() or inputs().

  Returns:
    Logits.
  """
  # We instantiate all variables using tf.get_variable() instead of
  # tf.Variable() in order to share variables across multiple GPU training runs.
  # If we only ran this model on a single GPU, we could simplify this function
  # by replacing all instances of tf.get_variable() with tf.Variable().
  #
  # conv1
  with tf.variable_scope('conv1') as scope:
    kernel = _variable_with_weight_decay('weights',
                                         shape=[5, 5, 3, 64],
                                         stddev=5e-2,
                                         wd=0.0)
    conv = tf.nn.conv2d(images, kernel, [1, 1, 1, 1], padding='SAME')
    biases = _variable_on_cpu('biases', [64], tf.constant_initializer(0.0))
    pre_activation = tf.nn.bias_add(conv, biases)
    conv1 = tf.nn.relu(pre_activation, name=scope.name)
    _activation_summary(conv1)

  # pool1
  pool1 = tf.nn.max_pool(conv1, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
                         padding='SAME', name='pool1')
  # norm1
  norm1 = tf.nn.lrn(pool1, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                    name='norm1')

  # conv2
  with tf.variable_scope('conv2') as scope:
    kernel = _variable_with_weight_decay('weights',
                                         shape=[5, 5, 64, 64],
```

```
                                        stddev=5e-2,
                                        wd=0.0)
  conv = tf.nn.conv2d(norm1, kernel, [1, 1, 1, 1], padding='SAME')
  biases = _variable_on_cpu('biases', [64], tf.constant_initializer(0.1))
  pre_activation = tf.nn.bias_add(conv, biases)
  conv2 = tf.nn.relu(pre_activation, name=scope.name)
  _activation_summary(conv2)

# norm2
norm2 = tf.nn.lrn(conv2, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                  name='norm2')
# pool2
pool2 = tf.nn.max_pool(norm2, ksize=[1, 3, 3, 1],
                       strides=[1, 2, 2, 1], padding='SAME', name='pool2')

# local3
with tf.variable_scope('local3') as scope:
  # Move everything into depth so we can perform a single matrix multiply.
  reshape = tf.reshape(pool2, [FLAGS.batch_size, -1])
  dim = reshape.get_shape()[1].value
  weights = _variable_with_weight_decay('weights', shape=[dim, 384],
                                        stddev=0.04, wd=0.004)
  biases = _variable_on_cpu('biases', [384], tf.constant_initializer(0.1))
  local3 = tf.nn.relu(tf.matmul(reshape, weights) + biases, name=scope.name)
  _activation_summary(local3)

# local4
with tf.variable_scope('local4') as scope:
  weights = _variable_with_weight_decay('weights', shape=[384, 192],
                                        stddev=0.04, wd=0.004)
  biases = _variable_on_cpu('biases', [192], tf.constant_initializer(0.1))
  local4 = tf.nn.relu(tf.matmul(local3, weights) + biases, name=scope.name)
  _activation_summary(local4)

# linear layer(WX + b),
# We don't apply softmax here because
# tf.nn.sparse_softmax_cross_entropy_with_logits accepts the unscaled logits
# and performs the softmax internally for efficiency.
with tf.variable_scope('softmax_linear') as scope:
  weights = _variable_with_weight_decay('weights', [192, cifar10.NUM_CLASSES],
                                        stddev=1/192.0, wd=0.0)
  biases = _variable_on_cpu('biases', [cifar10.NUM_CLASSES],
                            tf.constant_initializer(0.0))
  softmax_linear = tf.add(tf.matmul(local4, weights), biases, name=scope.name)
  _activation_summary(softmax_linear)

return softmax_linear
```

Missing Object Orientation?

Contrast the model code presented in this architecture with the policy code from the previous architecture. Note how the introduction of the `Layer` object allows for dramatically simplified code with concomitant improvements in readability. This sharp improvement in readability is part of the reason most developers prefer to use an object-oriented overlay on top of TensorFlow in practice.

That said, in this chapter, we use raw TensorFlow, since making classes like `TensorGraph` work with multiple GPUs would require significant additional overhead. In general, raw TensorFlow code offers maximum flexibility, but object orientation offers convenience. Pick the abstraction necessary for the problem at hand.

Training on Multiple GPUs

We instantiate a separate version of the model and architecture on each GPU. We then use the CPU to average the weights for the separate GPU nodes (Example 9-3).

Example 9-3. This function trains the Cifar10 model

```
def train():
  """Train CIFAR10 for a number of steps."""
  with tf.Graph().as_default(), tf.device('/cpu:0'):
    # Create a variable to count the number of train() calls. This equals the
    # number of batches processed * FLAGS.num_gpus.
    global_step = tf.get_variable(
        'global_step', [],
        initializer=tf.constant_initializer(0), trainable=False)

    # Calculate the learning rate schedule.
    num_batches_per_epoch = (cifar10.NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN /
                             FLAGS.batch_size)
    decay_steps = int(num_batches_per_epoch * cifar10.NUM_EPOCHS_PER_DECAY)

    # Decay the learning rate exponentially based on the number of steps.
    lr = tf.train.exponential_decay(cifar10.INITIAL_LEARNING_RATE,
                                    global_step,
                                    decay_steps,
                                    cifar10.LEARNING_RATE_DECAY_FACTOR,
                                    staircase=True)

    # Create an optimizer that performs gradient descent.
    opt = tf.train.GradientDescentOptimizer(lr)

    # Get images and labels for CIFAR-10.
    images, labels = cifar10.distorted_inputs()
```