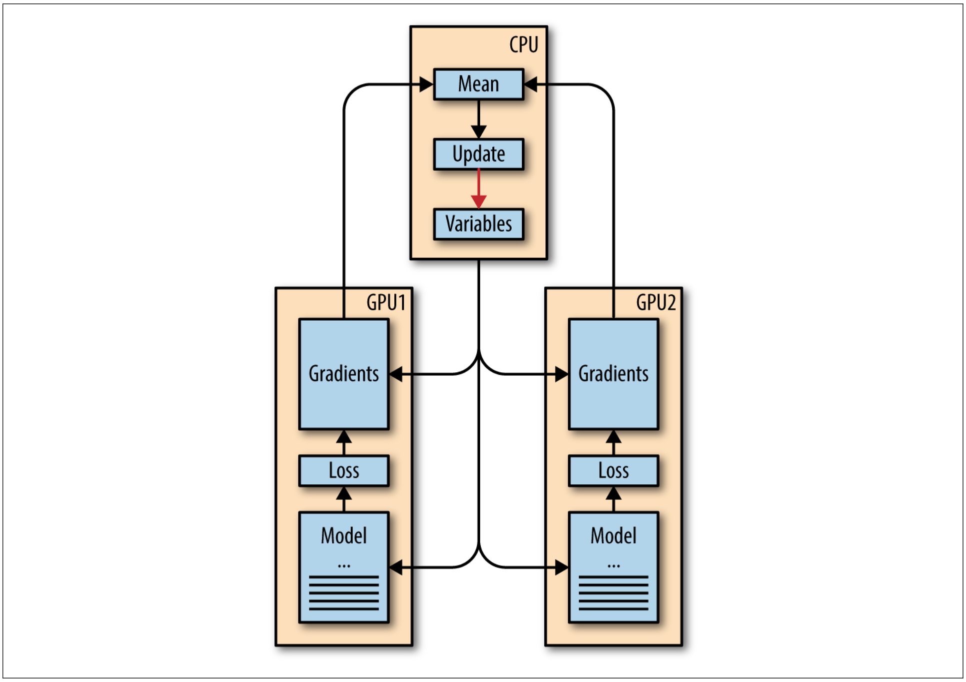

Figure 9-8. The data parallel architecture you will train in this chapter.

Downloading and Loading the DATA

The `read_cifar10()` method reads and parses the Cifar10 raw data files. Example 9-1 uses `tf.FixedLengthRecordReader` to read raw data from the Cifar10 files.

Example 9-1. This function reads and parses data from Cifar10 raw data files

```
def read_cifar10(filename_queue):
  """Reads and parses examples from CIFAR10 data files.

  Recommendation: if you want N-way read parallelism, call this function
  N times.  This will give you N independent Readers reading different
  files & positions within those files, which will give better mixing of
  examples.

  Args:
    filename_queue: A queue of strings with the filenames to read from.

  Returns:
    An object representing a single example, with the following fields:
      height: number of rows in the result (32)
      width: number of columns in the result (32)
      depth: number of color channels in the result (3)
```

```
    key: a scalar string Tensor describing the filename & record number
      for this example.
    label: an int32 Tensor with the label in the range 0..9.
    uint8image:: a [height, width, depth] uint8 Tensor with the image data
"""

class CIFAR10Record(object):
  pass
result = CIFAR10Record()

# Dimensions of the images in the CIFAR-10 dataset.
# See http://www.cs.toronto.edu/~kriz/cifar.html for a description of the
# input format.
label_bytes = 1  # 2 for CIFAR-100
result.height = 32
result.width = 32
result.depth = 3
image_bytes = result.height * result.width * result.depth
# Every record consists of a label followed by the image, with a
# fixed number of bytes for each.
record_bytes = label_bytes + image_bytes

# Read a record, getting filenames from the filename_queue.  No
# header or footer in the CIFAR-10 format, so we leave header_bytes
# and footer_bytes at their default of 0.
reader = tf.FixedLengthRecordReader(record_bytes=record_bytes)
result.key, value = reader.read(filename_queue)

# Convert from a string to a vector of uint8 that is record_bytes long.
record_bytes = tf.decode_raw(value, tf.uint8)

# Read a record, getting filenames from the filename_queue.  No
# header or footer in the CIFAR-10 format, so we leave header_bytes
# and footer_bytes at their default of 0.
reader = tf.FixedLengthRecordReader(record_bytes=record_bytes)
result.key, value = reader.read(filename_queue)

# Convert from a string to a vector of uint8 that is record_bytes long.
record_bytes = tf.decode_raw(value, tf.uint8)

# The first bytes represent the label, which we convert from uint8->int32.
result.label = tf.cast(
    tf.strided_slice(record_bytes, [0], [label_bytes]), tf.int32)

# The remaining bytes after the label represent the image, which we reshape
# from [depth * height * width] to [depth, height, width].
depth_major = tf.reshape(
    tf.strided_slice(record_bytes, [label_bytes],
                     [label_bytes + image_bytes]),
    [result.depth, result.height, result.width])
# Convert from [depth, height, width] to [height, width, depth].
result.uint8image = tf.transpose(depth_major, [1, 2, 0])
```

```
  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],
```