```
and performs mini-batching of the inputs.

The PTB dataset comes from Tomas Mikolov's webpage:
http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz

Args:
  data_path: string path to the directory where simple-examples.tgz
             has been extracted.

Returns:
  tuple (train_data, valid_data, test_data, vocabulary)
  where each of the data objects can be passed to PTBIterator.
"""

train_path = os.path.join(data_path, "ptb.train.txt")
valid_path = os.path.join(data_path, "ptb.valid.txt")
test_path = os.path.join(data_path, "ptb.test.txt")

word_to_id = _build_vocab(train_path)
train_data = _file_to_word_ids(train_path, word_to_id)
valid_data = _file_to_word_ids(valid_path, word_to_id)
test_data = _file_to_word_ids(test_path, word_to_id)
vocabulary = len(word_to_id)
return train_data, valid_data, test_data, vocabulary
```

tf.GFile and tf.Flags

TensorFlow is a large project that contains many bits and pieces. While most of the library is devoted to machine learning, there's also a large proportion that's dedicated to loading and massaging data. Some of these functions provide useful capabilities that aren't found elsewhere. Other parts of the loading functionality are less useful, however.

`tf.GFile` and `tf.FLags` provide functionality that is more or less identical to standard Python file handling and `argparse`. The provenance of these tools is historical. With Google, custom file handlers and flag handling are required by internal code standards. For the rest of us, though, it's better style to use standard Python tools whenever possible. It's much better for readability and stability.

Loading Data into TensorFlow

In this section, we cover the code needed to load our processed indices into TensorFlow. To do so, we will introduce you to a new bit of TensorFlow machinery. Until now, we've used feed dictionaries to pass data into TensorFlow. While feed dictionaries are fine for small toy datasets, they are often not good choices for larger datasets, since large Python overheads involving packing and unpacking dictionaries are introduced. For more performant code, it's better to use TensorFlow queues.

tf.Queue provides a way to load data asynchronously. This allows decoupling of the GPU compute thread from the CPU-bound data preprocessing thread. This decoupling is particularly useful for large datasets where we want to keep the GPU maximally active.

It's possible to feed tf.Queue objects into TensorFlow placeholders to train models and achieve greater performance. We will demonstrate how to do so later in this chapter.

The function ptb_producer introduced in Example 7-5 transforms raw lists of indices into tf.Queues that can pass data into a TensorFlow computational graph. Let's start by introducing some of the computational primitives we use. tf.train.range_input_producer is a convenience operation that produces a tf.Queue from an input tensor. The method tf.Queue.dequeue() pulls a tensor from the queue for training. tf.strided_slice extracts the part of this tensor that corresponds to the data for the current minibatch.

Example 7-5. This function loads the Penn Treebank data from the specified location

```
def ptb_producer(raw_data, batch_size, num_steps, name=None):
  """Iterate on the raw PTB data.

  This chunks up raw_data into batches of examples and returns
  Tensors that are drawn from these batches.

  Args:
    raw_data: one of the raw data outputs from ptb_raw_data.
    batch_size: int, the batch size.
    num_steps: int, the number of unrolls.
    name: the name of this operation (optional).

  Returns:
    A pair of Tensors, each shaped [batch_size, num_steps]. The
    second element of the tuple is the same data time-shifted to the
    right by one.

  Raises:
    tf.errors.InvalidArgumentError: if batch_size or num_steps are
    too high.
  """
  with tf.name_scope(name, "PTBProducer",
                     [raw_data, batch_size, num_steps]):
    raw_data = tf.convert_to_tensor(raw_data, name="raw_data",
                                    dtype=tf.int32)

    data_len = tf.size(raw_data)
    batch_len = data_len // batch_size
    data = tf.reshape(raw_data[0 : batch_size * batch_len],
                      [batch_size, batch_len])
```

```
epoch_size = (batch_len - 1) // num_steps
assertion = tf.assert_positive(
    epoch_size,
    message="epoch_size == 0, decrease batch_size or num_steps")
with tf.control_dependencies([assertion]):
  epoch_size = tf.identity(epoch_size, name="epoch_size")

i = tf.train.range_input_producer(epoch_size,
                                  shuffle=False).dequeue()
x = tf.strided_slice(data, [0, i * num_steps],
                     [batch_size, (i + 1) * num_steps])
x.set_shape([batch_size, num_steps])
y = tf.strided_slice(data, [0, i * num_steps + 1],
                     [batch_size, (i + 1) * num_steps + 1])
y.set_shape([batch_size, num_steps])
return x, y
```

tf.data

TensorFlow (from version 1.4 onward) supports a new module `tf.data` with a new class `tf.data.Dataset` that provides an explicit API for representing streams of data. It's likely that `tf.data` will eventually supersede queues as the preferred input modality, especially since it has a well-thought-out functional API.

At the time of writing, the `tf.data` module was just released and remained relatively immature compared with other parts of the API, so we decided to stick with queues for the examples. However, we encourage you to learn about `tf.data` yourself.

The Basic Recurrent Architecture

We will use an LSTM cell for modeling the Penn Treebank, since LSTMs often offer superior performance for language modeling challenges. The function `tf.contrib.rnn.BasicLSTMCell` implements the basic LSTM cell for us already, so no need to implement it ourselves (Example 7-6).

Example 7-6. This function wraps an LSTM cell from tf.contrib

```
def lstm_cell():
  return tf.contrib.rnn.BasicLSTMCell(
      size, forget_bias=0.0, state_is_tuple=True,
      reuse=tf.get_variable_scope().reuse)
```