```
    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)
```

Is Using TensorFlow Contrib Code OK?

Note that the LSTM implementation we use is drawn from tf.con trib. Is it acceptable to use code from tf.contrib for industrial-strength projects? The jury still appears to be out on this one. From our personal experience, code in tf.contrib tends to be a bit shakier than code in the core TensorFlow library, but is usually still pretty solid. There are often many useful libraries and utilities that are only available as part of tf.contrib. Our recommendation is to use pieces from tf.contrib as necessary, but make note of the pieces you use and replace them if an equivalent in the core TensorFlow library becomes available.

The snippet in Example 7-7 instructs TensorFlow to learn a word embedding for each word in our vocabulary. The key function for us is tf.nn.embedding_lookup, which allows us to perform the correct tensorial lookup operation. Note that we need to manually define the embedding matrix as a TensorFlow variable.

Example 7-7. Learn a word embedding for each word in the vocabulary

```
with tf.device("/cpu:0"):
  embedding = tf.get_variable(
      "embedding", [vocab_size, size], dtype=tf.float32)
  inputs = tf.nn.embedding_lookup(embedding, input_.input_data)
```

With our word vectors in hand, we simply need to apply the LSTM cell (using function lstm_cell) to each word vector in our sequence. To do this, we simply use a Python for-loop to construct the needed set of calls to cell(). There's only one trick here: we need to make sure we reuse the same variables at each timestep, since the LSTM cell should perform the same operation at each timestep. Luckily, the method reuse_variables() for variable scopes allows us to do so without much effort. See Example 7-8.

Example 7-8. Apply LSTM cell to each word vector in input sequence

```
outputs = []
state = self._initial_state
with tf.variable_scope("RNN"):
  for time_step in range(num_steps):
    if time_step > 0: tf.get_variable_scope().reuse_variables()
    (cell_output, state) = cell(inputs[:, time_step, :], state)
    outputs.append(cell_output)
```

All that remains now is to define the loss associated with the graph in order to train it. Conveniently, TensorFlow offers a loss for training language models in `tf.contrib`. We need only make a call to `tf.contrib.seq2seq.sequence_loss` (Example 7-9). Underneath the hood, this loss turns out to be a form of perplexity.

Example 7-9. Add the sequence loss

```
# use the contrib sequence loss and average over the batches
loss = tf.contrib.seq2seq.sequence_loss(
   logits,
   input_.targets,
   tf.ones([batch_size, num_steps], dtype=tf.float32),
   average_across_timesteps=False,
   average_across_batch=True
)
# update the cost variables
self._cost = cost = tf.reduce_sum(loss)
```

Perplexity

Perplexity is often used for language modeling challenges. It is a variant of the binary cross-entropy that is useful for measuring how close the learned distribution is to the true distribution of data. Empirically, perplexity has proven useful for many language modeling challenges and we make use of it here in that capacity (since the `sequence_loss` just implements perplexity specialized to sequences inside).

We can then train this graph using a standard gradient descent method. We leave out some of the messy details of the underlying code, but suggest you check GitHub if curious. Evaluating the quality of the trained model turns out to be straightforward as well, since the perplexity is used both as the training loss and the evaluation metric. As a result, we can simply display `self._cost` to gauge how the model is training. We encourage you to train the model for yourself!

Challenge for the Reader

Try lowering perplexity on the Penn Treebank by experimenting with different model architectures. Note that these experiments might be time-consuming without a GPU.

Review

This chapter introduced you to recurrent neural networks (RNNs), a powerful architecture for learning on sequential data. RNNs are capable of learning the underlying evolution rule that governs a sequence of data. While RNNs can be used for modeling