```
# Loop through training steps.
for step in xrange(int(num_epochs * train_size) // BATCH_SIZE):
  # Compute the offset of the current minibatch in the data.
  # Note that we could use better randomization across epochs.
  offset = (step * BATCH_SIZE) % (train_size - BATCH_SIZE)
  batch_data = train_data[offset:(offset + BATCH_SIZE), ...]
  batch_labels = train_labels[offset:(offset + BATCH_SIZE)]
  # This dictionary maps the batch data (as a NumPy array) to the
  # node in the graph it should be fed to.
  feed_dict = {train_data_node: batch_data,
               train_labels_node: batch_labels}
  # Run the optimizer to update weights.
  sess.run(optimizer, feed_dict=feed_dict)
  # print some extra information once reach the evaluation frequency
  if step % EVAL_FREQUENCY == 0:
    # fetch some extra nodes' data
    l, lr, predictions = sess.run([loss, learning_rate,
                                   train_prediction],
                                  feed_dict=feed_dict)
    elapsed_time = time.time() - start_time
    start_time = time.time()
    print('Step %d (epoch %.2f), %.1f ms' %
          (step, float(step) * BATCH_SIZE / train_size,
           1000 * elapsed_time / EVAL_FREQUENCY))
    print('Minibatch loss: %.3f, learning rate: %.6f' % (l, lr))
    print('Minibatch error: %.1f%%'
          % error_rate(predictions, batch_labels))
    print('Validation error: %.1f%%' % error_rate(
        eval_in_batches(validation_data, sess), validation_labels))
    sys.stdout.flush()
# Finally print the result!
test_error = error_rate(eval_in_batches(test_data, sess),
                        test_labels)
print('Test error: %.1f%%' % test_error)
```

Challenge for the Reader

Try training the network yourself. You should be able to achieve test error of < 1%!

Review

In this chapter, we have shown you the basic concepts of convolutional network design. These concepts include convolutional and pooling layers that constitute core building blocks of convolutional networks. We then discussed applications of convolutional architectures such as object detection, image segmentation, and image generation. We ended the chapter with an in-depth case study that showed you how to train a convolutional architecture on the MNIST handwritten digit dataset.

In Chapter 7, we will cover recurrent neural networks, another core deep learning architecture. Unlike convolutional networks, which were designed for image processing, recurrent architectures are powerfully suited to handling sequential data such as natural language datasets.

CHAPTER 7
Recurrent Neural Networks

So far in this book, we've introduced you to the use of deep learning to process various types of inputs. We started from simple linear and logistic regression on fixed dimensional feature vectors, and then followed up with a discussion of fully connected deep networks. These models take in arbitrary feature vectors with fixed, predetermined sizes. These models make no assumptions about the type of data encoded into these vectors. On the other hand, convolutional networks place strong assumptions upon the structure of their data. Inputs to convolutional networks have to satisfy a locality assumption that allows for the definition of a local receptive field.

How could we use the networks that we've described thus far to process data like sentences? Sentences do have some locality properties (nearby words are typically related), and it is indeed possible to use a one-dimensional convolutional network to process sentence data. That said, most practitioners resort to a different type of architecture, the recurrent neural network, in order to handle sequences of data.

Recurrent neural networks (RNNs) are designed natively to allow deep networks to process sequences of data. RNNs assume that incoming data takes the form of a sequence of vectors or tensors. If we transform each word in a sentence into a vector (more on how to do this later), sentences can be fed into RNNs. Similarly, video (considered as a sequence of images) can similarly be processed by an RNN. At each sequence position, an RNN applies an arbitrary nonlinear transformation to the input at that sequence location. This nonlinear transformation is shared for all sequence steps.

The description in the previous paragraph is a little abstract, but turns out to be immensely powerful. In this chapter, you will learn more details about how RNNs are structured and about how to implement RNNs in TensorFlow. We will also discuss how RNNs can be used in practice to perform tasks like sampling new sentences or generating text for applications such as chatbots.