Example 6-12. Define placeholders for the architecture

```
BATCH_SIZE = 64
EVAL_BATCH_SIZE = 64

train_data_node = tf.placeholder(
    tf.float32,
    shape=(BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, NUM_CHANNELS))
train_labels_node = tf.placeholder(tf.int64, shape=(BATCH_SIZE,))
eval_data = tf.placeholder(
    tf.float32,
    shape=(EVAL_BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, NUM_CHANNELS))
```

With these definitions in place, we now have the data processed, inputs and weights specified, and the model constructed. We are now prepared to train the network (Example 6-13).

Example 6-13. Training the LeNet-5 architecture

```
# Create a local session to run the training.
start_time = time.time()
with tf.Session() as sess:
  # Run all the initializers to prepare the trainable parameters.
  tf.global_variables_initializer().run()
  # 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)
```

The structure of this fitting code looks quite similar to other code for fitting we've seen so far in this book. In each step, we construct a feed dictionary, and then run a step of the optimizer. Note that we use minibatch training as before.

Evaluating Trained Models

We now have a model training. How can we evaluate the accuracy of the trained model? A simple method is to define an error metric. As in previous chapters, we shall use a simple classification metric to gauge accuracy (Example 6-14).

Example 6-14. Evaluating the error of trained architectures

```
def error_rate(predictions, labels):
  """Return the error rate based on dense predictions and sparse labels."""
  return 100.0 - (
      100.0 *
      numpy.sum(numpy.argmax(predictions, 1) == labels) /
      predictions.shape[0])
```

We can use this function to evaluate the error of the network as we train. Let's introduce an additional convenience function that evaluates predictions on any given dataset in batches (Example 6-15). This convenience is necessary since our network can only handle inputs with fixed batch sizes.

Example 6-15. Evaluating a batch of data at a time

```
def eval_in_batches(data, sess):
  """Get predictions for a dataset by running it in small batches."""
  size = data.shape[0]
  if size < EVAL_BATCH_SIZE:
    raise ValueError("batch size for evals larger than dataset: %d"
                     % size)
  predictions = numpy.ndarray(shape=(size, NUM_LABELS),
                              dtype=numpy.float32)
  for begin in xrange(0, size, EVAL_BATCH_SIZE):
    end = begin + EVAL_BATCH_SIZE
    if end <= size:
      predictions[begin:end, :] = sess.run(
          eval_prediction,
          feed_dict={eval_data: data[begin:end, ...]})
    else:
      batch_predictions = sess.run(
          eval_prediction,
          feed_dict={eval_data: data[-EVAL_BATCH_SIZE:, ...]})
      predictions[begin:, :] = batch_predictions[begin - size:, :]
  return predictions
```

We can now add a little instrumentation (in the inner `for`-loop of training) that periodically evaluates the model's accuracy on the validation set. We can end training by scoring test accuracy. Example 6-16 shows the full fitting code with instrumentation added in.

Example 6-16. The full code for training the network, with instrumentation added

```
# Create a local session to run the training.
start_time = time.time()
with tf.Session() as sess:
  # Run all the initializers to prepare the trainable parameters.
  tf.global_variables_initializer().run()
```

```
# 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.