The code in Example 6-7 defines max pooling in TensorFlow.

Example 6-7. Defining max pooling in TensorFlow

```
tf.nn.max_pool(
    value,
    ksize,
    strides,
    padding,
    data_format='NHWC',
    name=None
)
```

The `tf.nn.max_pool` function performs max pooling. Here `value` has the same shape as `input` for `tf.nn.conv2d`, `(batch, height, width, channels)`. `ksize` is the size of the pooling window and is a list of length 4. `strides` and `padding` behave as for `tf.nn.conv2d`.

The Convolutional Architecture

The architecture defined in this section will closely resemble LeNet-5, the original architecture used to train convolutional neural networks on the MNIST dataset. At the time the LeNet-5 architecture was invented, it was exorbitantly expensive computationally, requiring multiple weeks of compute to complete training. Today's laptops thankfully are more than sufficient to train LeNet-5 models. Figure 6-19 illustrates the structure of the LeNet-5 architecture.

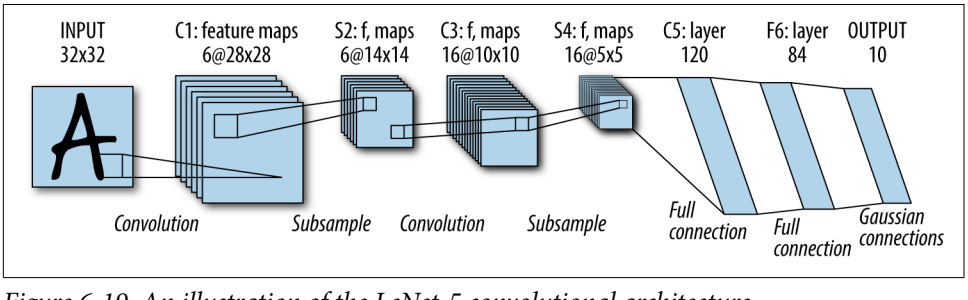

Figure 6-19. An illustration of the LeNet-5 convolutional architecture.

Where Would More Compute Make a Difference?

The LeNet-5 architecture is decades old, but is essentially the right architecture for the problem of digit recognition. However, its computational requirements forced the architecture into relative obscurity for decades. It's interesting to ask, then, what research problems today are similarly solved but limited solely by lack of adequate computational power?

One good contender is video processing. Convolutional models are quite good at processing video. However, it is unwieldy to store and train models on large video datasets, so most academic papers don't report results on video datasets. As a result, it's not so easy to hack together a good video processing system.

This situation will likely change as computing capabilities increase and it's likely that video processing systems will become much more commonplace. However, there's one critical difference between today's hardware improvements and those of past decades. Unlike in years past, Moore's law has slowed dramatically. As a result, improvements in hardware require more than natural transistor shrinkage and often require considerable ingenuity in architecture design. We will return to this topic in later chapters and discuss the architectural needs of deep networks.

Let's define the weights needed to train our LeNet-5 network. We start by defining some basic constants that are used to define our weight tensors (Example 6-8).

Example 6-8. Defining basic constants for the LeNet-5 model

```
NUM_CHANNELS = 1
IMAGE_SIZE = 28
NUM_LABELS = 10
```

The architecture we define will use two convolutional layers interspersed with two pooling layers, topped off by two fully connected layers. Recall that pooling requires no learnable weights, so we simply need to create weights for the convolutional and fully connected layers. For each `tf.nn.conv2d`, we need to create a learnable weight tensor corresponding to the `filter` argument for `tf.nn.conv2d`. In this particular architecture, we will also add a convolutional bias, one for each output channel (Example 6-9).

Example 6-9. Defining learnable weights for the convolutional layers

```
conv1_weights = tf.Variable(
    tf.truncated_normal([5, 5, NUM_CHANNELS, 32],  # 5x5 filter, depth 32.
                        stddev=0.1,
                        seed=SEED, dtype=tf.float32))
conv1_biases = tf.Variable(tf.zeros([32], dtype=tf.float32))
conv2_weights = tf.Variable(tf.truncated_normal(
    [5, 5, 32, 64], stddev=0.1,
    seed=SEED, dtype=tf.float32))
conv2_biases = tf.Variable(tf.constant(0.1, shape=[64], dtype=tf.float32))
```

Note that the convolutional weights are 4-tensors, while the biases are 1-tensors. The first fully connected layer converts the outputs of the convolutional layer to a vector of size 512. The input images start with size `IMAGE_SIZE=28`. After the two pooling layers (each of which reduces the input by a factor of 2), we end with images of size `IMAGE_SIZE//4`. We create the shape of the fully connected weights accordingly.

The second fully connected layer is used to provide the 10-way classification output, so it has weight shape `(512,10)` and bias shape `(10)`, shown in Example 6-10.

Example 6-10. Defining learnable weights for the fully connected layers

```
fc1_weights = tf.Variable(  # fully connected, depth 512.
    tf.truncated_normal([IMAGE_SIZE // 4 * IMAGE_SIZE // 4 * 64, 512],
                        stddev=0.1,
                        seed=SEED,
                        dtype=tf.float32))
fc1_biases = tf.Variable(tf.constant(0.1, shape=[512], dtype=tf.float32))
fc2_weights = tf.Variable(tf.truncated_normal([512, NUM_LABELS],
                                              stddev=0.1,
                                              seed=SEED,
                                              dtype=tf.float32))
fc2_biases = tf.Variable(tf.constant(
    0.1, shape=[NUM_LABELS], dtype=tf.float32))
```

With all the weights defined, we are now free to define the architecture of the network. The architecture has six layers in the pattern conv-pool-conv-pool-full-full (Example 6-11).

Example 6-11. Defining the LeNet-5 architecture. Calling the function defined in this example will instantiate the architecture.

```
def model(data, train=False):
  """The Model definition."""
  # 2D convolution, with 'SAME' padding (i.e. the output feature map has
  # the same size as the input). Note that {strides} is a 4D array whose
```

```
  # shape matches the data layout: [image index, y, x, depth].
  conv = tf.nn.conv2d(data,
                      conv1_weights,
                      strides=[1, 1, 1, 1],
                      padding='SAME')
  # Bias and rectified linear non-linearity.
  relu = tf.nn.relu(tf.nn.bias_add(conv, conv1_biases))
  # Max pooling. The kernel size spec {ksize} also follows the layout of
  # the data. Here we have a pooling window of 2, and a stride of 2.
  pool = tf.nn.max_pool(relu,
                        ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1],
                        padding='SAME')
  conv = tf.nn.conv2d(pool,
                      conv2_weights,
                      strides=[1, 1, 1, 1],
                      padding='SAME')
  relu = tf.nn.relu(tf.nn.bias_add(conv, conv2_biases))
  pool = tf.nn.max_pool(relu,
                        ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1],
                        padding='SAME')
  # Reshape the feature map cuboid into a 2D matrix to feed it to the
  # fully connected layers.
  pool_shape = pool.get_shape().as_list()
  reshape = tf.reshape(
      pool,
      [pool_shape[0], pool_shape[1] * pool_shape[2] * pool_shape[3]])
  # Fully connected layer. Note that the '+' operation automatically
  # broadcasts the biases.
  hidden = tf.nn.relu(tf.matmul(reshape, fc1_weights) + fc1_biases)
  # Add a 50% dropout during training only. Dropout also scales
  # activations such that no rescaling is needed at evaluation time.
  if train:
    hidden = tf.nn.dropout(hidden, 0.5, seed=SEED)
  return tf.matmul(hidden, fc2_weights) + fc2_biases
```

As noted previously, the basic architecture of the network intersperses `tf.nn.conv2d`, `tf.nn.max_pool`, with nonlinearities, and a final fully connected layer. For regularization, a dropout layer is applied after the final fully connected layer, but only during training. Note that we pass in the input as an argument `data` to the function `model()`.

The only part of the network that remains to be defined are the placeholders (Example 6-12). We need to define two placeholders for inputting the training images and the training labels. In this particular network, we also define a separate placeholder for evaluation that allows us to input larger batches when evaluating.

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).