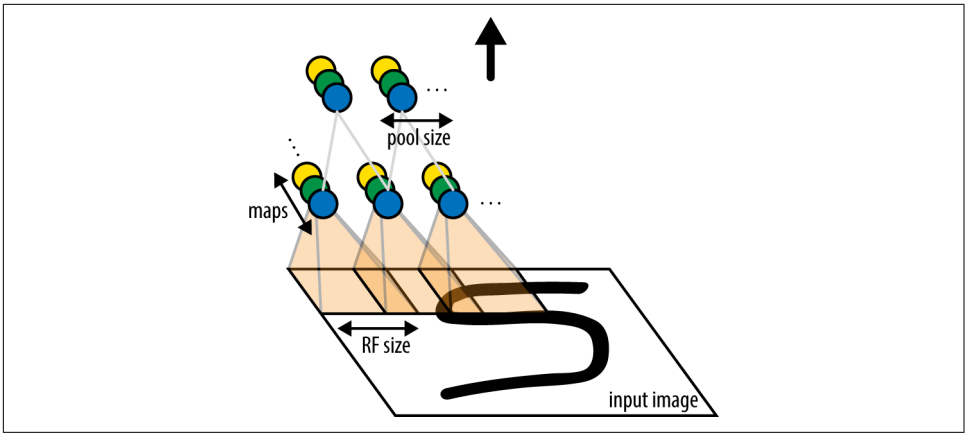

Figure 6-2. The local receptive field (RF) of a "neuron" in a convolutional network.

A layer of such "convolutional neurons" can be combined into a convolutional layer. This layer can viewed as a transformation of one spatial region into another. In the case of images, one batch of images is transformed into another by a convolutional layer. Figure 6-3 illustrates such a transformation. In the next section, we will show you more details about how a convolutional layer is constructed.

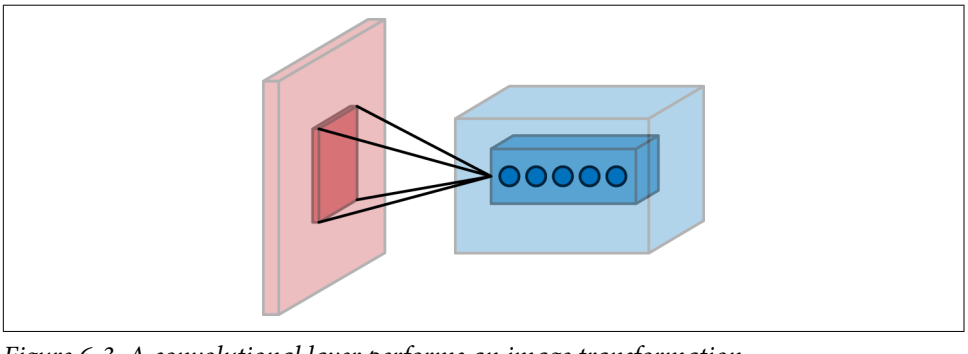

Figure 6-3. A convolutional layer performs an image transformation.

It's worth emphasizing that local receptive fields don't have to be limited to image data. For example, in stacked convolutional architectures, where the output of one convolutional layer feeds into the input of the next, the local receptive field will correspond to a "patch" of processed feature data.

Convolutional Kernels

In the last section, we mentioned that a convolutional layer applies nonlinear function to a local receptive field in its input. This locally applied nonlinearity is at the heart of convolutional architectures, but it's not the only piece. The second part of the

convolution is what's called a "convolutional kernel." A convolutional kernel is just a matrix of weights, much like the weights associated with a fully connected layer. Figure 6-4 diagrammatically illustrates how a convolutional kernel is applied to inputs.

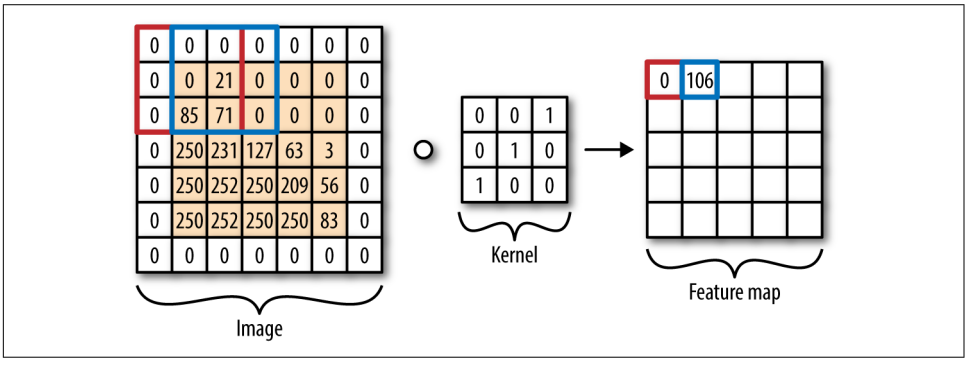

Figure 6-4. A convolutional kernel is applied to inputs. The kernel weights are multiplied elementwise with the corresponding numbers in the local receptive field and the multiplied numbers are summed. Note that this corresponds to a convolutional layer without a nonlinearity.

The key idea behind convolutional networks is that the same (nonlinear) transformation is applied to every local receptive field in the image. Visually, picture the local receptive field as a sliding window dragged over the image. At each positioning of the local receptive field, the nonlinear function is applied to return a single number corresponding to that image patch. As Figure 6-4 demonstrates, this transformation turns one grid of numbers into another grid of numbers. For image data, it's common to label the size of the local receptive field in terms of the number of pixels on each size of the receptive field. For example, 5×5 and 7×7 local receptive field sizes are commonly seen in convolutional networks.

What if we want to specify that local receptive fields should not overlap? The way to do this is to alter the *stride size* of the convolutional kernel. The stride size controls how the receptive field is moved over the input. Figure 6-4 demonstrates a one-dimensional convolutional kernel, with stride sizes 1 and 2, respectively. Figure 6-5 illustrates how altering the stride size changes how the receptive field is moved over the input.

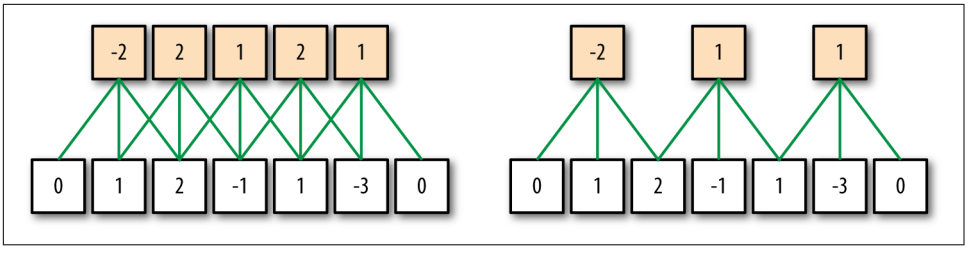

Figure 6-5. The stride size controls how the local receptive field "slides" over the input. This is easiest to visualize on a one-dimensional input. The network on the left has stride 1, while that on the right has stride 2. Note that each local receptive field computes the maximum of its inputs.

Now, note that the convolutional kernel we have defined transforms a grid of numbers into another grid of numbers. What if we want more than one grid of numbers output? It's easy enough; we simply need to add more convolutional kernels for processing the image. Convolutional kernels are also called *filters*, so the number of filters in a convolutional layer controls the number of transformed grids we obtain. A collection of convolutional kernels forms a *convolutional layer*.

Convolutional Kernels on Multidimensional Inputs

In this section, we primarily described convolutional kernels as transforming grids of numbers into other grids of numbers. Recalling our tensorial language from earlier chapters, convolutions transform matrices into matrices.

What if your input has more dimensions? For example, an RGB image typically has three color channels, so an RGB image is rightfully a rank-3 tensor. The simplest way to handle RGB data is to dictate that each local receptive field includes all the color channels associated with pixels in that patch. You might then say that the local receptive field is of size $5 \times 5 \times 3$ for a local receptive field of size 5×5 pixels with three color channels.

In general, you can generalize to tensors of higher dimension by expanding the dimensionality of the local receptive field correspondingly. This may also necessitate having multidimensional strides, especially if different dimensions are to be handled separately. The details are straightforward to work out, and we leave exploration of multidimensional convolutional kernels as an exercise for you to undertake.

Pooling Layers

In the previous section, we introduced the notion of convolutional kernels. These kernels apply learnable nonlinear transformations to local patches of inputs. These transformations are learnable, and by the universal approximation theorem, capable of learning arbitrarily complex input transformations on local patches. This flexibility gives convolutional kernels much of their power. But at the same time, having many learnable weights in a deep convolutional network can slow training.

Instead of using a learnable transformation, it's possible to instead use a fixed nonlinear transformation in order to reduce the computational cost of training a convolutional network. A popular fixed nonlinearity is "max pooling." Such layers select and output the maximally activating input within each local receptive patch. Figure 6-6 demonstrates this process. Pooling layers are useful for reducing the dimensionality of input data in a structured fashion. More mathematically, they take a local receptive field and replace the nonlinear activation function at each portion of the field with the max (or min or average) function.

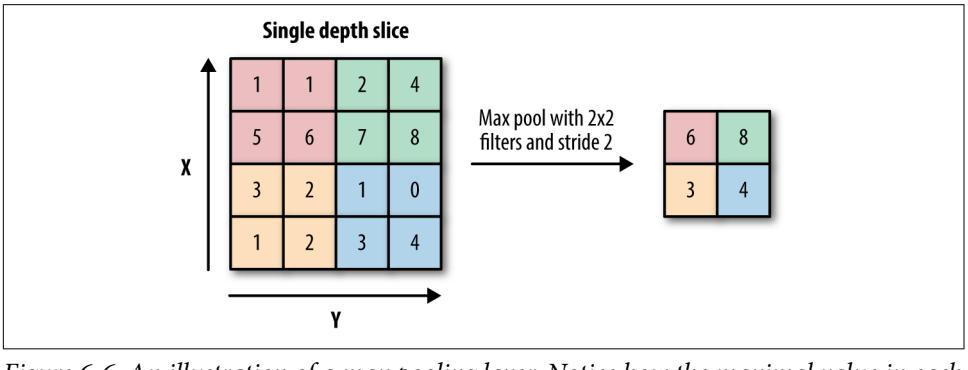

Figure 6-6. An illustration of a max pooling layer. Notice how the maximal value in each colored region (each local receptive field) is reported in the output.

Pooling layers have become less useful as hardware has improved. While pooling is still useful as a dimensionality reduction technique, recent research tends to avoid using pooling layers due to their inherent lossiness (it's not possible to back out of pooled data which pixel in the input originated the reported activation). Nonetheless, pooling appears in many standard convolutional architectures so it's worth understanding.

Constructing Convolutional Networks

A simple convolutional architecture applies a series of convolutional layers and pooling layers to its input to learn a complex function on the input image data. There are a lot of details in forming these networks, but at its heart, architecture design is sim-