It is a bit ugly due to all the different indices, but all it does is calculate the weighted sum of all the inputs, plus the bias term.

Equation 13-1. Computing the output of a neuron in a convolutional layer

$$z_{i,j,k} = b_k + \sum_{u=1}^{f_h} \sum_{v=1}^{f_w} \sum_{k'=1}^{f_{n'}} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with} \begin{cases} i' = u \cdot s_h + f_h - 1 \\ j' = v \cdot s_w + f_w - 1 \end{cases}$$

- $z_{i,j,k}$ is the output of the neuron located in row i, column j in feature map k of the convolutional layer (layer l).
- As explained earlier, s_h and s_w are the vertical and horizontal strides, f_h and f_w are the height and width of the receptive field, and $f_{n'}$ is the number of feature maps in the previous layer (layer $l - 1$).
- $x_{i',j',k'}$ is the output of the neuron located in layer $l - 1$, row i', column j', feature map k' (or channel k' if the previous layer is the input layer).
- b_k is the bias term for feature map k (in layer l). You can think of it as a knob that tweaks the overall brightness of the feature map k.
- $w_{u,v,k',k}$ is the connection weight between any neuron in feature map k of the layer l and its input located at row u, column v (relative to the neuron's receptive field), and feature map k'.

TensorFlow Implementation

In TensorFlow, each input image is typically represented as a 3D tensor of `shape [height, width, channels]`. A mini-batch is represented as a 4D tensor of `shape [mini-batch size, height, width, channels]`. The weights of a convolutional layer are represented as a 4D tensor of shape $[f_h, f_w, f_n, f_{n'}]$. The bias terms of a convolutional layer are simply represented as a 1D tensor of `shape [`f_n`]`.

Let's look at a simple example. The following code loads two sample images, using Scikit-Learn's `load_sample_images()` (which loads two color images, one of a Chinese temple, and the other of a flower). Then it creates two 7×7 filters (one with a vertical white line in the middle, and the other with a horizontal white line), and applies them to both images using a convolutional layer built using TensorFlow's `conv2d()` function (with zero padding and a stride of 2). Finally, it plots one of the resulting feature maps (similar to the top-right image in Figure 13-5).

```
import numpy as np
from sklearn.datasets import load_sample_images

# Load sample images
dataset = np.array(load_sample_images().images, dtype=np.float32)
batch_size, height, width, channels = dataset.shape

# Create 2 filters
filters_test = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters_test[:, 3, :, 0] = 1  # vertical line
filters_test[3, :, :, 1] = 1  # horizontal line

# Create a graph with input X plus a convolutional layer applying the 2 filters
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
convolution = tf.nn.conv2d(X, filters, strides=[1,2,2,1], padding="SAME")

with tf.Session() as sess:
    output = sess.run(convolution, feed_dict={X: dataset})

plt.imshow(output[0, :, :, 1])  # plot 1st image's 2nd feature map
plt.show()
```

Most of this code is self-explanatory, but the `conv2d()` line deserves a bit of explanation:

- `X` is the input mini-batch (a 4D tensor, as explained earlier).
- `filters` is the set of filters to apply (also a 4D tensor, as explained earlier).
- `strides` is a four-element 1D array, where the two central elements are the vertical and horizontal strides (s_h and s_w). The first and last elements must currently be equal to 1. They may one day be used to specify a batch stride (to skip some instances) and a channel stride (to skip some of the previous layer's feature maps or channels).
- `padding` must be either `"VALID"` or `"SAME"`:
  - If set to `"VALID"`, the convolutional layer does *not* use zero padding, and may ignore some rows and columns at the bottom and right of the input image, depending on the stride, as shown in Figure 13-7 (for simplicity, only the horizontal dimension is shown here, but of course the same logic applies to the vertical dimension).
  - If set to `"SAME"`, the convolutional layer uses zero padding if necessary. In this case, the number of output neurons is equal to the number of input neurons divided by the stride, rounded up (in this example, ceil (13 / 5) = 3). Then zeros are added as evenly as possible around the inputs.

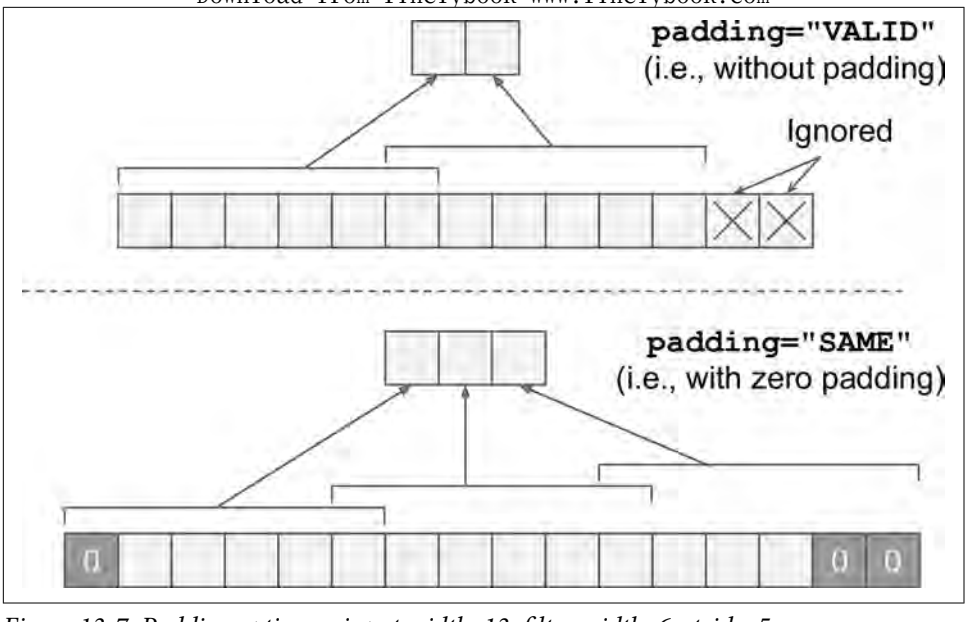

Figure 13-7. Padding options—input width: 13, filter width: 6, stride: 5

Unfortunately, convolutional layers have quite a few hyperparameters: you must choose the number of filters, their height and width, the strides, and the padding type. As always, you can use cross-validation to find the right hyperparameter values, but this is very time-consuming. We will discuss common CNN architectures later, to give you some idea of what hyperparameter values work best in practice.

Memory Requirements

Another problem with CNNs is that the convolutional layers require a huge amount of RAM, especially during training, because the reverse pass of backpropagation requires all the intermediate values computed during the forward pass.

For example, consider a convolutional layer with 5×5 filters, outputting 200 feature maps of size 150×100, with stride 1 and SAME padding. If the input is a 150×100 RGB image (three channels), then the number of parameters is $(5 \times 5 \times 3 + 1) \times 200 = 15{,}200$ (the +1 corresponds to the bias terms), which is fairly small compared to a fully connected layer.[7] However, each of the 200 feature maps contains 150×100 neurons, and each of these neurons needs to compute a weighted sum of its $5 \times 5 \times 3 = 75$ inputs: that's a total of 225 million float multiplications. Not as bad as a fully con-

7 A fully connected layer with 150×100 neurons, each connected to all $150 \times 100 \times 3$ inputs, would have $150^2 \times 100^2 \times 3 = 675$ million parameters!