

mon to add zeros around the inputs, as shown in the diagram. This is called *zero padding*.

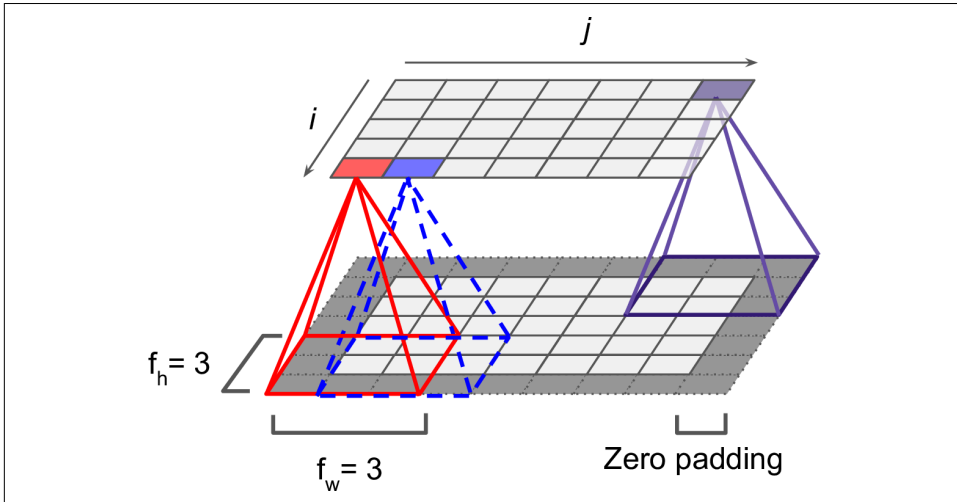


Figure 14-3. Connections between layers and zero padding

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields, as shown in Figure 14-4. The shift from one receptive field to the next is called the *stride*. In the diagram, a  $5 \times 7$  input layer (plus zero padding) is connected to a  $3 \times 4$  layer, using  $3 \times 3$  receptive fields and a stride of 2 (in this example the stride is the same in both directions, but it does not have to be so). A neuron located in row  $i$ , column  $j$  in the upper layer is connected to the outputs of the neurons in the previous layer located in rows  $i \times s_h$  to  $i \times s_h + f_h - 1$ , columns  $j \times s_w$  to  $j \times s_w + f_w - 1$ , where  $s_h$  and  $s_w$  are the vertical and horizontal strides.

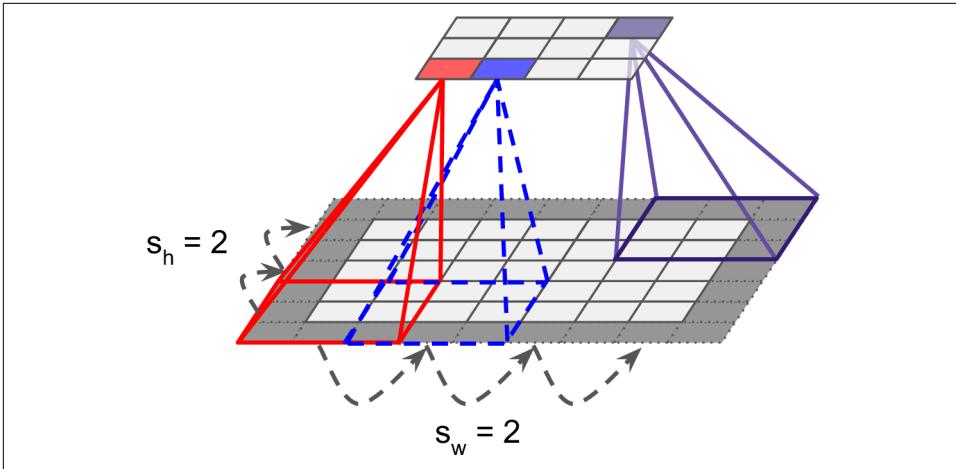


Figure 14-4. Reducing dimensionality using a stride of 2

## Filters

A neuron's weights can be represented as a small image the size of the receptive field. For example, [Figure 14-5](#) shows two possible sets of weights, called *filters* (or *convolution kernels*). The first one is represented as a black square with a vertical white line in the middle (it is a  $7 \times 7$  matrix full of 0s except for the central column, which is full of 1s); neurons using these weights will ignore everything in their receptive field except for the central vertical line (since all inputs will get multiplied by 0, except for the ones located in the central vertical line). The second filter is a black square with a horizontal white line in the middle. Once again, neurons using these weights will ignore everything in their receptive field except for the central horizontal line.

Now if all neurons in a layer use the same vertical line filter (and the same bias term), and you feed the network the input image shown in [Figure 14-5](#) (bottom image), the layer will output the top-left image. Notice that the vertical white lines get enhanced while the rest gets blurred. Similarly, the upper-right image is what you get if all neurons use the same horizontal line filter; notice that the horizontal white lines get enhanced while the rest is blurred out. Thus, a layer full of neurons using the same filter outputs a *feature map*, which highlights the areas in an image that activate the filter the most. Of course you do not have to define the filters manually: instead, during training the convolutional layer will automatically learn the most useful filters for its task, and the layers above will learn to combine them into more complex patterns.

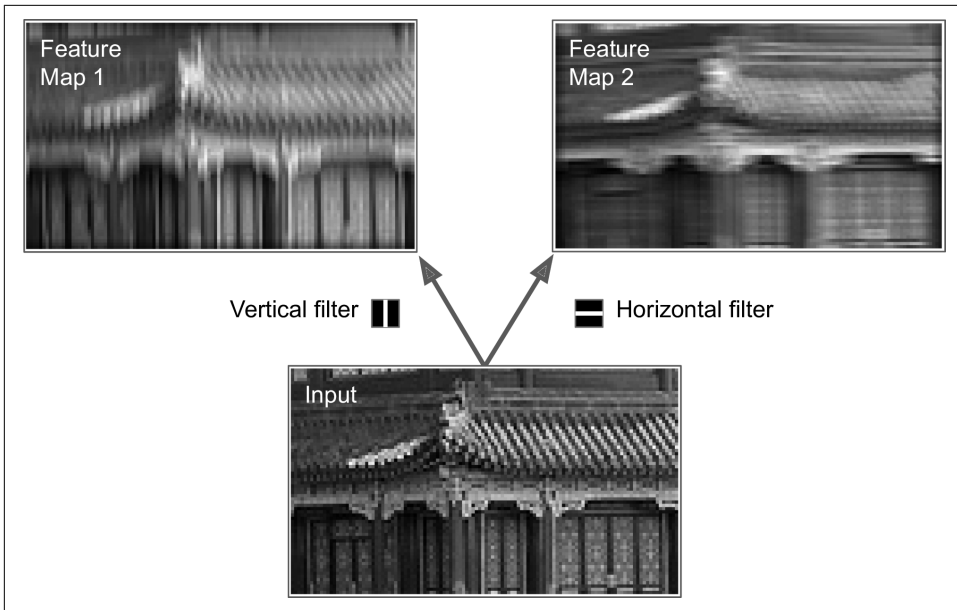


Figure 14-5. Applying two different filters to get two feature maps

## Stacking Multiple Feature Maps

Up to now, for simplicity, I have represented the output of each convolutional layer as a thin 2D layer, but in reality a convolutional layer has multiple filters (you decide how many), and it outputs one feature map per filter, so it is more accurately represented in 3D (see Figure 14-6). To do so, it has one neuron per pixel in each feature map, and all neurons within a given feature map share the same parameters (i.e., the same weights and bias term). However, neurons in different feature maps use different parameters. A neuron's receptive field is the same as described earlier, but it extends across all the previous layers' feature maps. In short, a convolutional layer simultaneously applies multiple trainable filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.



The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the model. Moreover, once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. In contrast, once a regular DNN has learned to recognize a pattern in one location, it can recognize it only in that particular location.

Moreover, input images are also composed of multiple sublayers: one per *color channel*. There are typically three: red, green, and blue (RGB). Grayscale images have just

one channel, but some images may have much more—for example, satellite images that capture extra light frequencies (such as infrared).

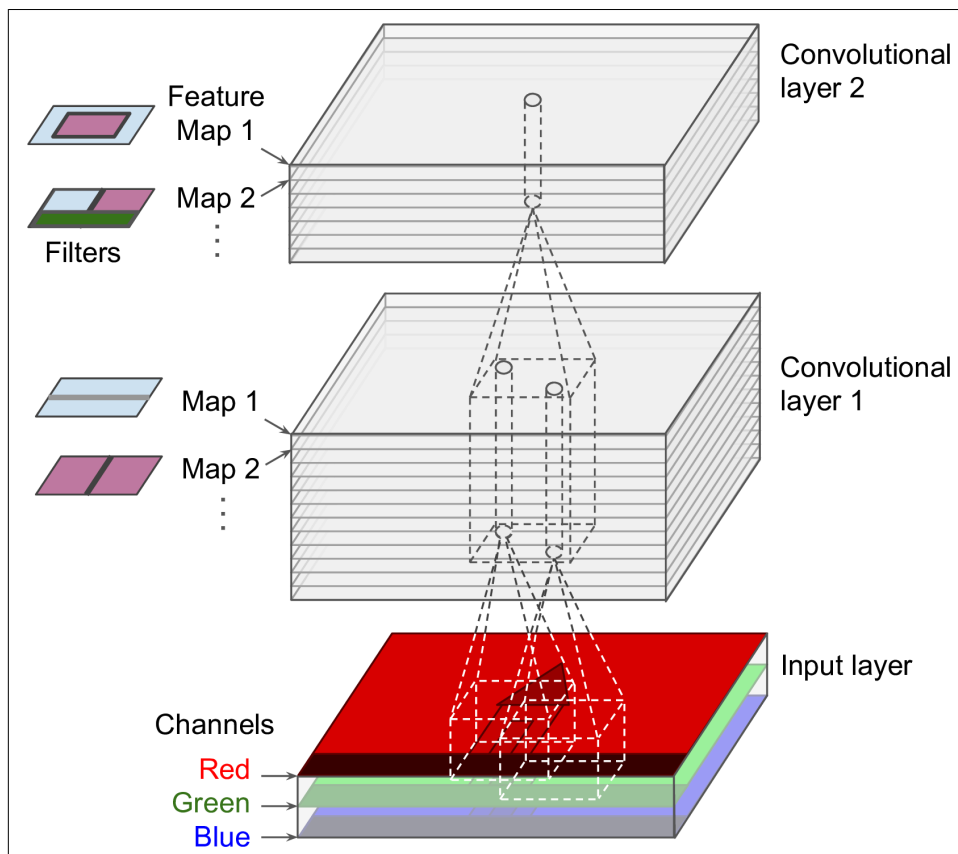


Figure 14-6. Convolution layers with multiple feature maps, and images with three color channels

Specifically, a neuron located in row  $i$ , column  $j$  of the feature map  $k$  in a given convolutional layer  $l$  is connected to the outputs of the neurons in the previous layer  $l - 1$ , located in rows  $i \times s_h$  to  $i \times s_h + f_h - 1$  and columns  $j \times s_w$  to  $j \times s_w + f_w - 1$ , across all feature maps (in layer  $l - 1$ ). Note that all neurons located in the same row  $i$  and column  $j$  but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.

**Equation 14-1** summarizes the preceding explanations in one big mathematical equation: it shows how to compute the output of a given neuron in a convolutional layer.

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 14-1. Computing the output of a neuron in a convolutional layer*

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_n-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with} \quad \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

- $z_{i,j,k}$  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). The pixel intensities (for each color channel) is represented as a byte from 0 to 255, so we scale these features simply by dividing by 255, to get floats ranging from 0 to 1. Then we create two  $7 \times 7$  filters (one with a vertical white line in the middle, and the other with a horizontal white line in the middle), and we apply them to both images using the `tf.nn.conv2d()` function, which is part of TensorFlow's low-level Deep Learning API. In this example, we use zero padding (`padding="SAME"`) and a stride of 2. Finally, we plot one of the resulting feature maps (similar to the top-right image in [Figure 14-5](#)).

```

from sklearn.datasets import load_sample_image

# Load sample images
china = load_sample_image("china.jpg") / 255
flower = load_sample_image("flower.jpg") / 255
images = np.array([china, flower])
batch_size, height, width, channels = images.shape

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

outputs = tf.nn.conv2d(images, filters, strides=1, padding="SAME")

plt.imshow(outputs[0, :, :, 1], cmap="gray") # plot 1st image's 2nd feature map
plt.show()

```

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

- `images` 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 equal to 1, but it could also be a 1D array with 4 elements, 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 14-7](#) (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,  $13 / 5 = 2.6$ , rounded up to 3). Then zeros are added as evenly as possible around the inputs.

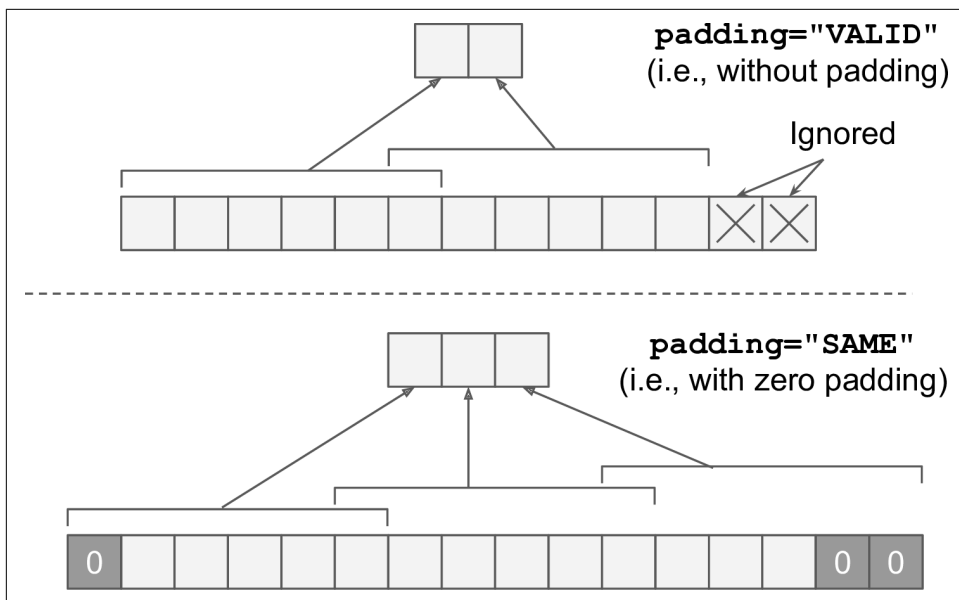


Figure 14-7. Padding options—input width: 13, filter width: 6, stride: 5

In this example, we manually defined the filters, but in a real CNN you would normally define filters as trainable variables, so the neural net can learn which filters work best, as explained earlier. Instead of manually creating the variables, however, you can simply use the `keras.layers.Conv2D` layer:

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,
                             padding="SAME", activation="relu")
```

This code creates a `Conv2D` layer with 32 filters, each  $3 \times 3$ , using a stride of 1 (both horizontally and vertically), `SAME` padding, and applying the `ReLU` activation function to its outputs. As you can see, 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. This is especially true 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  $5 \times 5$  filters, outputting 200 feature maps of size  $150 \times 100$ , with stride 1 and `SAME` padding. If the input is a  $150 \times 100$

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.<sup>7</sup> However, each of the 200 feature maps contains  $150 \times 100$  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 connected layer, but still quite computationally intensive. Moreover, if the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy  $200 \times 150 \times 100 \times 32 = 96$  million bits (12 MB) of RAM.<sup>8</sup> And that's just for one instance! If a training batch contains 100 instances, then this layer will use up 1.2 GB of RAM!

During inference (i.e., when making a prediction for a new instance) the RAM occupied by one layer can be released as soon as the next layer has been computed, so you only need as much RAM as required by two consecutive layers. But during training everything computed during the forward pass needs to be preserved for the reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers.



If training crashes because of an out-of-memory error, you can try reducing the mini-batch size. Alternatively, you can try reducing dimensionality using a stride, or removing a few layers. Or you can try using 16-bit floats instead of 32-bit floats. Or you could distribute the CNN across multiple devices.

Now let's look at the second common building block of CNNs: the *pooling layer*.

## Pooling Layer

Once you understand how convolutional layers work, the pooling layers are quite easy to grasp. Their goal is to *subsample* (i.e., shrink) the input image in order to reduce the computational load, the memory usage, and the number of parameters (thereby limiting the risk of overfitting).

Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. You must define its size, the stride, and the padding type, just like before. However, a pooling neuron has no weights; all it does is aggregate the inputs using an aggregation function such as the max or mean. **Figure 14-8** shows a *max pooling layer*, which is the most common type of pooling layer. In this example,

---

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

<sup>8</sup> In the international system of units (SI),  $1 \text{ MB} = 1,000 \text{ kB} = 1,000 \times 1,000 \text{ bytes} = 1,000 \times 1,000 \times 8 \text{ bits}$ .



we use a  $2 \times 2$  pooling kernel<sup>9</sup>, with a stride of 2, and no padding. Only the max input value in each receptive field makes it to the next layer, while the other inputs are dropped. For example, in the lower left receptive field in Figure 14-8, the input values are 1, 5, 3, 2, so only the max value, 5, is propagated to the next layer. Because of the stride of 2, the output image has half the height and half the width of the input image (rounded down since we use no padding).

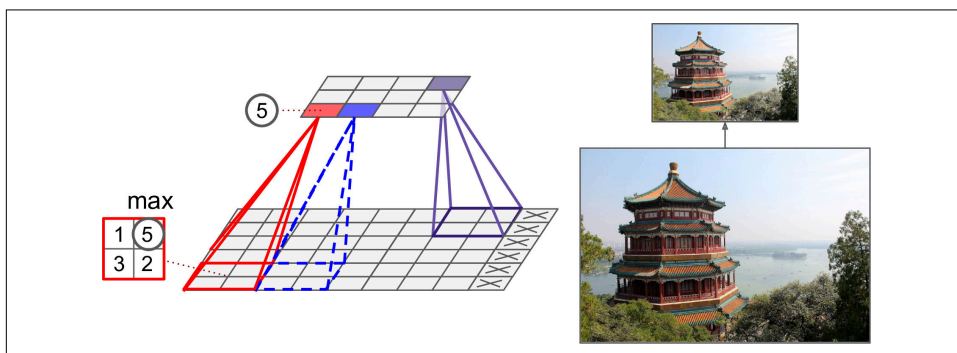


Figure 14-8. Max pooling layer ( $2 \times 2$  pooling kernel, stride 2, no padding)



A pooling layer typically works on every input channel independently, so the output depth is the same as the input depth.

Other than reducing computations, memory usage and the number of parameters, a max pooling layer also introduces some level of *invariance* to small translations, as shown in Figure 14-9. Here we assume that the bright pixels have a lower value than dark pixels, and we consider 3 images (A, B, C) going through a max pooling layer with a  $2 \times 2$  kernel and stride 2. Images B and C are the same as image A, but shifted by one and two pixels to the right. As you can see, the outputs of the max pooling layer for images A and B are identical. This is what translation invariance means. However, for image C, the output is different: it is shifted by one pixel to the right (but there is still 75% invariance). By inserting a max pooling layer every few layers in a CNN, it is possible to get some level of translation invariance at a larger scale. Moreover, max pooling also offers a small amount of rotational invariance and a slight scale invariance. Such invariance (even if it is limited) can be useful in cases where the prediction should not depend on these details, such as in classification tasks.

<sup>9</sup> Other kernels we discussed so far had weights, but pooling kernels do not: they are just stateless sliding windows.

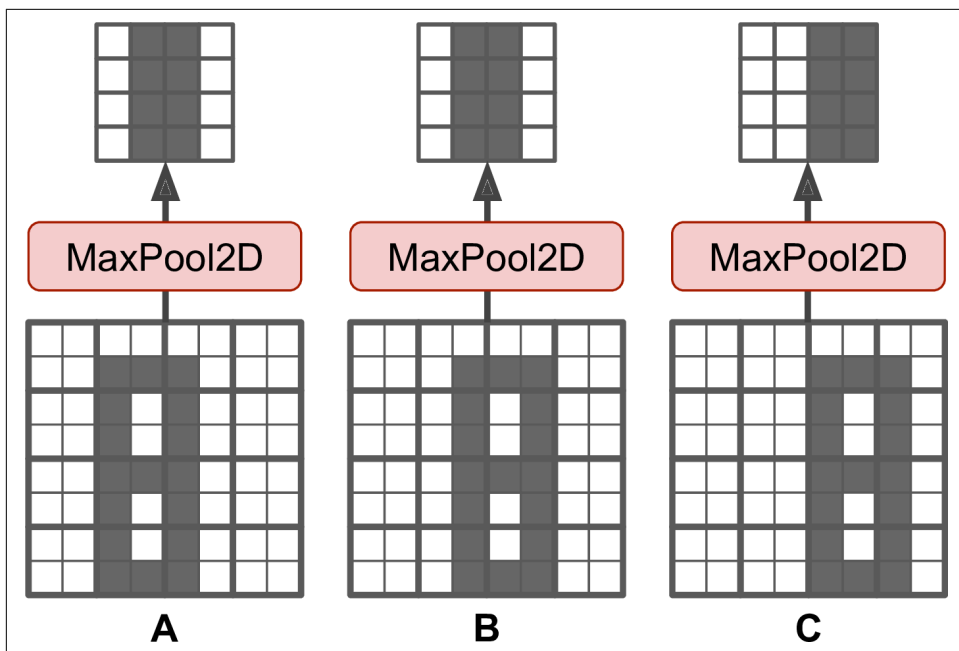


Figure 14-9. Invariance to small translations

But max pooling has some downsides: firstly, it is obviously very destructive: even with a tiny  $2 \times 2$  kernel and a stride of 2, the output will be two times smaller in both directions (so its area will be four times smaller), simply dropping 75% of the input values. And in some applications, invariance is not desirable, for example for *semantic segmentation*: this is the task of classifying each pixel in an image depending on the object that pixel belongs to: obviously, if the input image is translated by 1 pixel to the right, the output should also be translated by 1 pixel to the right. The goal in this case is *equivariance*, not invariance: a small change to the inputs should lead to a corresponding small change in the output.

## TensorFlow Implementation

Implementing a max pooling layer in TensorFlow is quite easy. The following code creates a max pooling layer using a  $2 \times 2$  kernel. The strides default to the kernel size, so this layer will use a stride of 2 (both horizontally and vertically). By default, it uses VALID padding (i.e., no padding at all):

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

To create an *average pooling layer*, just use AvgPool2D instead of MaxPool2D. As you might expect, it works exactly like a max pooling layer, except it computes the mean rather than the max. Average pooling layers used to be very popular, but people