



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!

nected 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 (about 11.4 MB) of RAM.⁸ And that's just for one instance! If a training batch contains 100 instances, then this layer will use up over 1 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). Reducing the input image size also makes the neural network tolerate a little bit of image shift (*location invariance*).

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 13-8** shows a *max pooling layer*, which is the most common type of pooling layer. In this example, we use a 2×2 *pooling kernel*, a stride of 2, and no padding. Note that only the max input value in each kernel makes it to the next layer. The other inputs are dropped.

⁸ 1 MB = 1,024 kB = 1,024 × 1,024 bytes = 1,024 × 1,024 × 8 bits.