```
!pip install d2l==1.0.3
```

Show hidden output

COSE474-2024F: Deep Learning

7.5. Pooling

Moreover, when detecting lower-level features, such as edges (as discussed in Section 7.2), we often want our representations to be somewhat invariant to translation. For instance, if we take the image X with a sharp delineation between black and white and shift the whole image by one pixel to the right, i.e., Z[i, j] = X[i, j + 1], then the output for the new image Z might be vastly different. The edge will have shifted by one pixel. In reality, objects hardly ever occur exactly at the same place. In fact, even with a tripod and a stationary object, vibration of the camera due to the movement of the shutter might shift everything by a pixel or so (high-end cameras are loaded with special features to address this problem).

This section introduces pooling layers, which serve the dual purposes of mitigating the sensitivity of convolutional layers to location and of spatially downsampling representations.

```
import torch
from torch import nn
from d2l import torch as d2l
```

7.5.1. Maximum Pooling and Average Pooling

Like convolutional layers, pooling operators consist of a fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed by the fixed-shape window (sometimes known as the pooling window). However, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters (there is no kernel). Instead, pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window. These operations are called maximum pooling (max-pooling for short) and average pooling, respectively.

Average pooling is essentially as old as CNNs. The idea is akin to downsampling an image. Rather than just taking the value of every second (or third) pixel for the lower resolution image, we can average over adjacent pixels to obtain an image with better signal-to-noise ratio since we are combining the information from multiple adjacent pixels. Max-pooling was introduced in Riesenhuber and Poggio (1999) in the context of cognitive neuroscience to describe how information aggregation might be aggregated hierarchically for the purpose of object recognition; there already was an earlier version in speech recognition (Yamaguchi et al., 1990). In almost all cases, max-pooling, as it is also referred to, is preferable to average pooling.

In both cases, as with the cross-correlation operator, we can think of the pooling window as starting from the upper-left of the input tensor and sliding across it from left to right and top to bottom. At each location that the pooling window hits, it computes the maximum or average value of the input subtensor in the window, depending on whether max or average pooling is employed.

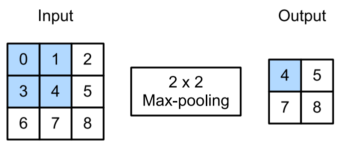

Fig. 7.5.1 Max-pooling with a pooling window shape of 2×2 . The shaded portions are the first output element as well as the input tensor elements used for the output computation: $\max(0, 1, 3, 4) = 4$

The output tensor in Fig. 7.5.1 has a height of 2 and a width of 2. The four elements are derived from the maximum value in each pooling window:

(7.5.1)

$$\begin{aligned}\max(0, 1, 3, 4) &= 4,\\ \max(1, 2, 4, 5) &= 5,\\ \max(3, 4, 6, 7) &= 7,\\ \max(4, 5, 7, 8) &= 8.\end{aligned}$$

More generally, we can define a $p \times q$ pooling layer by aggregating over a region of said size. Returning to the problem of edge detection, we use the output of the convolutional layer as input for 2×2 max-pooling. Denote by `X` the input of the convolutional layer input and `Y` the pooling layer output. Regardless of whether or not the values of `X[i, j]`, `X[i, j + 1]`, `X[i+1, j]` and `X[i+1, j + 1]` are different, the pooling layer always outputs `Y[i, j] = 1`. That is to say, using the 2×2 max-pooling layer, we can still detect if the pattern recognized by the convolutional layer moves no more than one element in height or width.

In the code below, we (**implement the forward propagation of the pooling layer**) in the `pool2d` function. This function is similar to the `corr2d` function in Section 7.2. However, no kernel is needed, computing the output as either the maximum or the average of each region in the input.

```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = torch.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i: i + p_h, j: j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
    return Y
```

We can construct the input tensor X in Fig. 7.5.1 to validate the output of the two-dimensional max-pooling layer.

```
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
tensor([[4., 5.],
        [7., 8.]])
```

Also, we can experiment with the average pooling layer.

```
pool2d(X, (2, 2), 'avg')
```

```
tensor([[2., 3.],
        [5., 6.]])
```

7.5.2. Padding and Stride

As with convolutional layers, pooling layers change the output shape. And as before, we can adjust the operation to achieve a desired output shape by padding the input and adjusting the stride. We can demonstrate the use of padding and strides in pooling layers via the built-in two-dimensional max-pooling layer from the deep learning framework. We first construct an input tensor X whose shape has four dimensions, where the number of examples (batch size) and number of channels are both 1.

```
X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4, 4))
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

Since pooling aggregates information from an area, deep learning frameworks default to matching pooling window sizes and stride. For instance, if we use a pooling window of shape (3, 3) we get a stride shape of (3, 3) by default.

```
pool2d = nn.MaxPool2d(3)
# Pooling has no model parameters, hence it needs no initialization
pool2d(X)
```

```
tensor([[[[10.]]]])
```

Needless to say, the stride and padding can be manually specified to override framework defaults if required.

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

Of course, we can specify an arbitrary rectangular pooling window with arbitrary height and width respectively, as the example below shows.

```
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

7.5.3. Multiple Channels

When processing multi-channel input data, the pooling layer pools each input channel separately, rather than summing the inputs up over channels as in a convolutional layer. This means that the number of output channels for the pooling layer is the same as the number of input channels. Below, we will concatenate tensors X and X + 1 on the channel dimension to construct an input with two channels.

```
X = torch.cat((X, X + 1), 1)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]],

         [[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
```

As we can see, the number of output channels is still two after pooling.

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]],

         [[ 6.,  8.],
          [14., 16.]]]])
```

7.5.4. Discussion

- Pooling is an exceedingly simple operation.
- Aggregate results over a window of values.
- Pooling is indifferent to channels, i.e., it leaves the number of channels unchanged and it applies to each channel separately.
- Max-pooling is preferable to average pooling, as it confers some degree of invariance to output.