COSE474-2024F: Deep Learning

7.3. Padding and Stride

ecall the example of a convolution in Fig. 7.2.1. The input had both a height and width of 3 and the convolution kernel had both a height and width of 2, yielding an output representation with dimension 2×2. Assuming that the input shape is $n_\mathrm{h} \times n_\mathrm{w}$ and the convolution kernel shape is $k_\mathrm{h} \times k_\mathrm{w}$, the output shape will be $(n_\mathrm{h} - k_\mathrm{h} + 1) \times (n_\mathrm{w} - k_\mathrm{w} + 1)$: we can only shift the convolution kernel so far until it runs out of pixels to apply the convolution to.

In the following we will explore a number of techniques, including padding and strided convolutions, that offer more control over the size of the output. As motivation, note that since kernels generally have width and height greater than 1, after applying many successive convolutions, we tend to wind up with outputs that are considerably smaller than our input. If we start with a 240×240 pixel image, ten layers of 5×5 convolutions reduce the image to 200×200 pixels, slicing off 30% of the image and with it obliterating any interesting information on the boundaries of the original image. *Padding* is the most popular tool for handling this issue. In other cases, we may want to reduce the dimensionality drastically, e.g., if we find the original input resolution to be unwieldy. *Strided convolutions* are a popular technique that can help in these instances.

```
import torch
from torch import nn
```

7.3.1. Padding

As described above, one tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image. Consider Fig. 7.3.1 that depicts the pixel utilization as a function of the convolution kernel size and the position within the image. The pixels in the corners are hardly used at all.

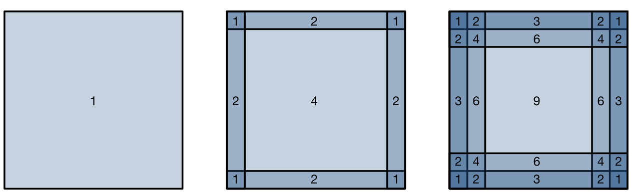

Fig. 7.3.1 Pixel utilization for convolutions of size 1×1, 2×2, and 3×3 respectively.

Since we typically use small kernels, for any given convolution we might only lose a few pixels but this can add up as we apply many successive convolutional layers. One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image. Typically, we set the values of the extra pixels to zero. In Fig. 7.3.2, we pad a 3×3 input, increasing its size to 5×5. The corresponding output then increases to a 4×4 matrix. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$.

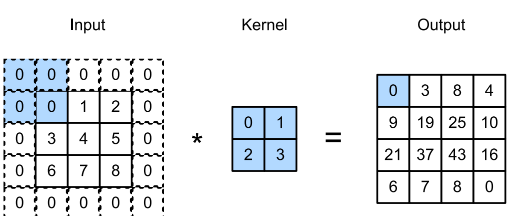

Fig. 7.3.2 Two-dimensional cross-correlation with padding.

In general, if we add a total of p_h rows of padding (roughly half on top and half on bottom) and a total of p_w columns of padding (roughly half on the left and half on the right), the output shape will be

$$(n_\mathrm{h} - k_\mathrm{h} + p_\mathrm{h} + 1) \times (n_\mathrm{w} - k_\mathrm{w} + p_\mathrm{w} + 1).$$

This means that the height and width of the output will increase by p_h and p_w, respectively.

In many cases, we will want to set $p_\mathrm{h} = k_\mathrm{h} - 1$ and $p_\mathrm{w} = k_\mathrm{w} - 1$ to give the input and output the same height and width. This will make it easier to predict the output shape of each layer when constructing the network. Assuming that k_h is odd here, we will pad $p_\mathrm{h}/2$ rows on both sides of the height. If k_h is even, one possibility is to pad $\lceil p_h/2 \rceil$ rows on the top of the input and $\lfloor p_h/2 \rfloor$ rows on the bottom. We will pad both sides of the width in the same way.

CNNs commonly use convolution kernels with odd height and width values, such as 1, 3, 5, or 7. Choosing odd kernel sizes has the benefit that we can preserve the dimensionality while padding with the same number of rows on top and bottom, and the same number of columns on left and right.

Moreover, this practice of using odd kernels and padding to precisely preserve dimensionality offers a clerical benefit. For any two-dimensional tensor `X`, when the kernel's size is odd and the number of padding rows and columns on all sides are the same, thereby producing an output with the same height and width as the input, we know that the output `Y[i, j]` is calculated by cross-correlation of the input and convolution kernel with the window centered on `X[i, j]`.

In the following example, we create a two-dimensional convolutional layer with a height and width of 3 and (**apply 1 pixel of padding on all sides.**) Given an input with a height and width of 8, we find that the height and width of the output is also 8.

```
# We define a helper function to calculate convolutions. It initializes the
# convolutional layer weights and performs corresponding dimensionality
# elevations and reductions on the input and output
def comp_conv2d(conv2d, X):
    # (1, 1) indicates that batch size and the number of channels are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Strip the first two dimensions: examples and channels
    return Y.reshape(Y.shape[2:])

# 1 row and column is padded on either side, so a total of 2 rows or columns
# are added
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

When the height and width of the convolution kernel are different, we can make the output and input have the same height and width by setting different padding numbers for height and width.

```
# We use a convolution kernel with height 5 and width 3. The padding on either
# side of the height and width are 2 and 1, respectively
conv2d = nn.LazyConv2d(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

7.3.2. Stride

When computing the cross-correlation, we start with the convolution window at the upper-left corner of the input tensor, and then slide it over all locations both down and to the right. In the previous examples, we defaulted to sliding one element at a time. However, sometimes, either for computational efficiency or because we wish to downsample, we move our window more than one element at a time, skipping the intermediate locations. This is particularly useful if the convolution kernel is large since it captures a large area of the underlying image.

We refer to the number of rows and columns traversed per slide as stride. So far, we have used strides of 1, both for height and width. Sometimes, we may want to use a larger stride. Fig. 7.3.3 shows a two-dimensional cross-correlation operation with a stride of 3 vertically and 2 horizontally. The shaded portions are the output elements as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8, 0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$.

We can see that when the second element of the first column is generated, the convolution window slides down three rows. The convolution window slides two columns to the right when the second element of the first row is generated. When the convolution window continues to slide two columns to the right on the input, there is no output because the input element cannot fill the window (unless we add another column of padding).

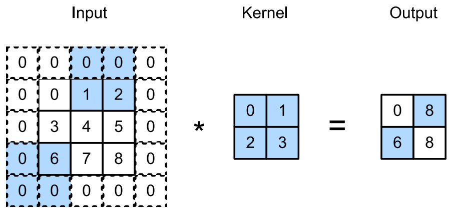

Fig. 7.3.3 Cross-correlation with strides of 3 and 2 for height and width, respectively.

In general, when the stride for the height is s_h and the stride for the width is s_w, the output shape is

$$\lfloor(n_\mathrm{h}-k_\mathrm{h}+p_\mathrm{h}+s_\mathrm{h})/s_\mathrm{h}\rfloor \times \lfloor(n_\mathrm{w}-k_\mathrm{w}+p_\mathrm{w}+s_\mathrm{w})/s_\mathrm{w}\rfloor.$$

If we set $p_\mathrm{h}=k_\mathrm{h}-1$ and $p_\mathrm{w}=k_\mathrm{w}-1$, then the output shape can be simplified to $\lfloor(n_\mathrm{h}+s_\mathrm{h}-1)/s_\mathrm{h}\rfloor \times \lfloor(n_\mathrm{w}+s_\mathrm{w}-1)/s_\mathrm{w}\rfloor$. Going a step further, if the input height and width are divisible by the strides on the height and width, then the output shape will be $(n_\mathrm{h}/s_\mathrm{h}) \times (n_\mathrm{w}/s_\mathrm{w})$.

Below, we [**set the strides on both the height and width to 2**], thus halving the input height and width.

```
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

```
torch.Size([4, 4])
```

```
conv2d = nn.LazyConv2d(1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([2, 2])
```

7.3.3. Summary and Discussion

- Padding can increase the height and width of the output.
- Ensures that all pixels are used equally frequently.
- Typically we pick symmetric padding on both sides of the input height and width. In this case we refer to $(p_\mathrm{h}, p_\mathrm{w})$ padding. Most commonly we set $p_\mathrm{h} = p_\mathrm{w}$, in which case we simply state that we choose padding p.
- A similar convention applies to strides. When horizontal stride s_h and vertical stride s_w match, we simply talk about stride s. The stride can reduce the resolution of the output, for example reducing the height and width of the output to only $1/n$ of the height and width of the input for $n > 1$. By default, the padding is 0 and the stride is 1.
- Allows CNNs to encode implicit position information within an image, simply by learning where the "whitespace" is. There are many alternatives to zero-padding.