

**Notebook credit:** Based on the original D2L notebook [here](#).

## ✓ Padding and Stride

- **Padding** 是指在输入数据的边缘周围添加额外的、通常是零值的像素，以调整数据的空间尺寸。这样做的目的有几个：
  1. **保持空间尺寸**：卷积操作通常会缩小数据的空间尺寸，添加Padding可以保持输出图像的尺寸不变，这对于设计深层网络特别有用，因为可以防止过早的尺寸缩小。
  2. **改善性能**：通过Padding，可以确保网络中更深层的卷积层能够接触到输入图像边缘的信息。
  3. **两种常见的Padding方式**：
    - **Valid Padding**（无Padding）：不添加任何Padding，导致输出尺寸小于输入尺寸。
    - **Same Padding**：添加足够的Padding以确保输出和输入具有相同的空间尺寸。
- **Stride** 指的是在执行卷积操作时，卷积核沿着输入数据滑动的步幅。Stride决定了卷积核覆盖输入数据的方式。
  1. **减少输出尺寸**：较大的Stride会减少卷积操作后输出的空间维度，这有助于降低数据的空间分辨率，从而减少计算量。
  2. **加快计算速度**：通过增大Stride，可以在覆盖整个输入图像的同时减少所需的卷积操作数，从而加快整体的计算速度。
  3. **常见的Stride设置**：
    - **Stride为1**：卷积核每次移动一个像素。
    - **Stride大于1**：卷积核跳过中间的像素，每次移动超过一个像素。

这两个参数对CNN的性能和效率有着直接的影响，通过合理的设置可以优化网络的结构和学习能力。

Input		Kernel		Output																	
<table border="1" style="border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>2</td></tr> <tr><td>3</td><td>4</td><td>5</td></tr> <tr><td>6</td><td>7</td><td>8</td></tr> </table>	0	1	2	3	4	5	6	7	8	*	<table border="1" style="border-collapse: collapse;"> <tr><td>0</td><td>1</td></tr> <tr><td>2</td><td>3</td></tr> </table>	0	1	2	3	=	<table border="1" style="border-collapse: collapse;"> <tr><td>19</td><td>25</td></tr> <tr><td>37</td><td>43</td></tr> </table>	19	25	37	43
0	1	2																			
3	4	5																			
6	7	8																			
0	1																				
2	3																				
19	25																				
37	43																				

In the above example, our input had both a height and width of 3 and our convolution kernel had both a height and width of 2, yielding an output representation with dimension  $2 \times 2$ . Assuming that the input shape is  $n_h \times n_w$  and the convolution kernel shape is  $k_h \times k_w$ , then the output

shape will be  $(n_h - k_h + 1) \times (n_w - k_w + 1)$ . Therefore, the output shape of the convolutional layer is determined by the shape of the input and the shape of the convolution kernel.

In several cases, we incorporate techniques, including padding and strided convolutions, that affect 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  $240 \times 240$  pixel image, 10 layers of  $5 \times 5$  convolutions reduce the image to  $200 \times 200$  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.

## ✓ Padding ()

As described above, one tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image. 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 the figure below, we pad a  $3 \times 3$  input, increasing its size to  $5 \times 5$ . The corresponding output then increases to a  $4 \times 4$  matrix. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation:  $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$ .

Input		Kernel		Output																																													
<table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>1</td><td>2</td><td>0</td></tr><tr><td>0</td><td>3</td><td>4</td><td>5</td><td>0</td></tr><tr><td>0</td><td>6</td><td>7</td><td>8</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	1	2	0	0	3	4	5	0	0	6	7	8	0	0	0	0	0	0	*	<table><tr><td>0</td><td>1</td></tr><tr><td>2</td><td>3</td></tr></table>	0	1	2	3	=	<table><tr><td>0</td><td>3</td><td>8</td><td>4</td></tr><tr><td>9</td><td>19</td><td>25</td><td>10</td></tr><tr><td>21</td><td>37</td><td>43</td><td>16</td></tr><tr><td>6</td><td>7</td><td>8</td><td>0</td></tr></table>	0	3	8	4	9	19	25	10	21	37	43	16	6	7	8	0
0	0	0	0	0																																													
0	0	1	2	0																																													
0	3	4	5	0																																													
0	6	7	8	0																																													
0	0	0	0	0																																													
0	1																																																
2	3																																																
0	3	8	4																																														
9	19	25	10																																														
21	37	43	16																																														
6	7	8	0																																														

## ✓ Padding 后 output shape 的公式

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

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1).$$

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

很多时候我们会设置  $p_h = k_h - 1$  and  $p_w = k_w - 1$  来使得 input 和 output 有相同的 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  $p_h/2$  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 spatial 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, 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.

```
import tensorflow as tf
from math import floor

# We define a convenience function to calculate the convolutional layer. This
# function initializes the convolutional layer weights and performs
# corresponding dimensionality elevations and reductions on the input and
# output
def comp_conv2d(conv2d, X):
    # Here (1, 1) indicates that the batch size and the number of channels
    # are both 1
    X = tf.reshape(X, (1, ) + X.shape + (1, ))
    Y = conv2d(X)
    # Exclude the first two dimensions that do not interest us: examples and
    # channels
    return tf.reshape(Y, Y.shape[1:3])
# Note that here 1 row or column is padded on either side, so a total of 2
# rows or columns are added
conv2d = tf.keras.layers.Conv2D(1, kernel_size=3, padding='same')
X = tf.random.uniform(shape=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
TensorShape([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.]**

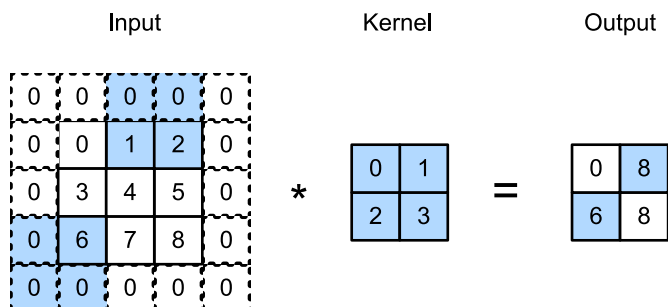
```
# Here, we use a convolution kernel with a height of 5 and a width of 3. The
# padding numbers on either side of the height and width are 2 and 1,
# respectively
conv2d = tf.keras.layers.Conv2D(1, kernel_size=(5, 3), padding='same')
comp_conv2d(conv2d, X).shape
```

```
TensorShape([8, 8])
```

## ✓ Stride (改变 kernel 滑动步长)

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 previous examples, we default 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.

We refer to the number of rows and columns traversed per slide as the *stride*. So far, we have used strides of 1, both for height and width. Sometimes, we may want to use a larger stride. The figure below 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 outputted, 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 outputted. 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).



## ✓ 改变 strides 后 output shape 的公式

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

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor.$$

If we set  $p_h = k_h - 1$  and  $p_w = k_w - 1$ , then the output shape will be simplified to

$\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$ . 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  $(n_h/s_h) \times (n_w/s_w)$ .

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

```
conv2d = tf.keras.layers.Conv2D(1, kernel_size=3, padding='same', strides=2)
comp_conv2d(conv2d, X).shape
```

```
TensorShape([4, 4])
```

Next, we will look at (a slightly more complicated example).

```
n_w, n_h = X.shape
k_w, k_h = 3, 5
s_w, s_h = 3, 4
```

```
conv2d = tf.keras.layers.Conv2D(1, kernel_size=(k_w,k_h), padding='valid',
                                strides=(s_w, s_h))
```

```
out_w = floor((n_w - k_w + 1)/s_w)
out_h = floor((n_h - k_h + 1)/s_h)
```

```
print(f"predicted shape: {(out_w, out_h)}")
print(f"actual shape: {comp_conv2d(conv2d, X).shape}")
```

```
predicted shape: (2, 1)
actual shape: (2, 1)
```

For the sake of brevity, when the padding number on both sides of the input height and width are  $p_h$  and  $p_w$  respectively, we call the padding  $(p_h, p_w)$ . Specifically, when  $p_h = p_w = p$ , the padding is  $p$ . When the strides on the height and width are  $s_h$  and  $s_w$ , respectively, we call the stride  $(s_h, s_w)$ . Specifically, when  $s_h = s_w = s$ , the stride is  $s$ . By default, the padding is 0 and the stride is 1. In practice, we rarely use inhomogeneous strides or padding, i.e., we usually have  $p_h = p_w$  and  $s_h = s_w$ .

## Summary

- Padding can increase the height and width of the output. This is often used to give the output the same height and width as the input.
- 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 ( $n$  is an integer greater than 1).
- Padding and stride can be used to adjust the dimensionality of the data effectively.