CHAPTER 3

Fundamentals of Convolutional Neural Networks

In this chapter, we will look at the main components of a convolutional neural network (CNN): kernels and pooling layers. We will then look at how a typical network looks. We will then try to solve a classification problem with a simple convolutional network and try to visualize the convolutional operation. The purpose of this is to try to understand, at least intuitively, how the learning works.

Kernels and Filters

One of the main components of CNNs are filters, which are square matrices that have dimensions $n_K \times n_K$, where n_K is an integer and is usually a small number, like 3 or 5. Sometimes filters are also called *kernels*. Using kernels comes from classical image processing techniques. If you have used Photoshop or similar software, you are used to do operations like

U. Michelucci, *Advanced Applied Deep Learning*,
https://doi.org/10.1007/978-1-4842-4976-5_3

sharpening, blurring, embossing, and so on.[1] All those operations are done with kernels. We will see in this section what exactly kernels are and how they work. Note that in this book we will use both terms (kernels and filters) interchangeably. Let's define four different filters and let's check later in the chapter their effect when used in convolution operations. For those examples, we will work with 3 × 3 filters. For the moment, just take the following definitions as a reference and we will see how to use them later in the chapter.

- The following kernel will allow the detection of horizontal edges

$$\mathfrak{I}_H = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

- The following kernel will allow the detection of vertical edges

$$\mathfrak{I}_V = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$

- The following kernel will allow the detection of edges when luminosity changes drastically

$$\mathfrak{I}_L = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

[1]You can find a nice overview on Wikipedia at https://en.wikipedia.org/wiki/Kernel_(image_processing).

- The following kernel will blur edges in an image

$$\mathfrak{I}_B = -\frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

In the next sections, we will apply convolution to a test image with the filters, to see what their effect is.

Convolution

The first step to understanding CNNs is to understand convolution. The easiest way is to see it in action with a few simple cases. First, in the context of neural networks, convolution is done between tensors. The operation gets two tensors as input and produces a tensor as output. The operation is usually indicated with the operator *.

Let's see how it works. Consider two tensors, both with dimensions 3 × 3. The convolution operation is done by applying the following formula:

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{pmatrix} * \begin{pmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{pmatrix} = \sum_{i=1}^{9} a_i k_i$$

In this case, the result is merely the sum of each element, a_i, multiplied by the respective element, k_i. In more typical matrix formalism, this formula could be written with a double sum as

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} * \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix} = \sum_{i=1}^{3}\sum_{j=1}^{3} a_{ij} k_{ij}$$

However, the first version has the advantage of making the fundamental idea very clear: each element from one tensor is multiplied by the correspondent element (the element in the same position) of the second tensor, and then all the values are summed to get the result.

In the previous section, we talked about kernels, and the reason is that convolution is usually done between a tensor, that we may indicate here with A, and a kernel. Typically, kernels are small, 3 × 3 or 5 × 5, while the input tensors A are normally bigger. In image recognition for example, the input tensors A are the images that may have dimensions as high as 1024 × 1024 × 3, where 1024 × 1024 is the resolution and the last dimension (3) is the number of the color channels, the RGB values.

In advanced applications, the images may even have higher resolution. To understand how to apply convolution when we have matrices with different dimensions, let's consider a matrix A that is 4 × 4

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

And a Kernel K that we will take for this example to be 3 × 3

$$K = \begin{pmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{pmatrix}$$

The idea is to start in the top-left corner of the matrix A and select a 3 × 3 region. In the example that would be

$$A_1 = \begin{pmatrix} a_1 & a_2 & a_3 \\ a_5 & a_6 & a_7 \\ a_9 & a_{10} & a_{11} \end{pmatrix}$$

Alternatively, the elements marked in boldface here:

$$A = \begin{pmatrix} \boldsymbol{a_1} & \boldsymbol{a_2} & \boldsymbol{a_3} & a_4 \\ \boldsymbol{a_5} & \boldsymbol{a_6} & \boldsymbol{a_7} & a_8 \\ \boldsymbol{a_9} & \boldsymbol{a_{10}} & \boldsymbol{a_{11}} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

Then we perform the convolution, as explained at the beginning between this smaller matrix A_1 and K, getting (we will indicate the result with B_1):

$$B_1 = A_1 * K = a_1k_1 + a_2k_2 + a_3k_3 + k_4a_5 + k_5a_5 + k_6a_7 + k_7a_9 + k_8a_{10} + k_9a_{11}$$

Then we need to shift the selected 3 × 3 region in matrix A of one column to the right and select the elements marked in bold here:

$$A = \begin{pmatrix} a_1 & \boldsymbol{a_2} & \boldsymbol{a_3} & \boldsymbol{a_4} \\ a_5 & \boldsymbol{a_6} & \boldsymbol{a_7} & \boldsymbol{a_8} \\ a_9 & \boldsymbol{a_{10}} & \boldsymbol{a_{11}} & \boldsymbol{a_{12}} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

This will give us the second sub-matrix A_2:

$$A_2 = \begin{pmatrix} a_2 & a_3 & a_4 \\ a_6 & a_7 & a_8 \\ a_{10} & a_{11} & a_{12} \end{pmatrix}$$

We then perform the convolution between this smaller matrix A_2 and K again:

$$B_2 = A_2 * K = a_2k_1 + a_3k_2 + a_4k_3 + a_6k_4 + a_7k_5 + a_8k_6 + a_{10}k_7 + a_{11}k_8 + a_{12}k_9$$

We cannot shift our 3 × 3 region anymore to the right, since we have reached the end of the matrix A, so what we do is shift it one row down and start again from the left side. The next selected region would be

$$A_3 = \begin{pmatrix} a_5 & a_6 & a_7 \\ a_9 & a_{10} & a_{11} \\ a_{13} & a_{14} & a_{15} \end{pmatrix}$$

Again, we perform convolution of A_3 with K

$$B_3 = A_3 * K = a_5k_1 + a_6k_2 + a_7k_3 + a_9k_4 + a_{10}k_5 + a_{11}k_6 + a_{13}k_7 + a_{14}k_8 + a_{15}k_9$$

As you may have guessed at this point, the last step is to shift our 3 × 3 selected region to the right of one column and perform convolution again. Our selected region will now be

$$A_4 = \begin{pmatrix} a_6 & a_7 & a_8 \\ a_{10} & a_{11} & a_{12} \\ a_{14} & a_{15} & a_{16} \end{pmatrix}$$

Moreover, the convolution will give this result:

$$B_4 = A_4 * K = a_6k_1 + a_7k_2 + a_8k_3 + a_{10}k_4 + a_{11}k_5 + a_{12}k_6 + a_{14}k_7 + a_{15}k_8 + a_{16}k_9$$

Now we cannot shift our 3 × 3 region anymore, neither right nor down. We have calculated four values: B_1, B_2, B_3, and B_4. Those elements will form the resulting tensor of the convolution operation giving us the tensor B:

$$B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

The same process can be applied when tensor A is bigger. You will simply get a bigger resulting B tensor, but the algorithm to get the elements B_i is the same. Before moving on, there is still a small detail that we need to discuss, and that is the concept of stride. In the previous process, we moved our 3×3 region always one column to the right and one row down. The number of rows and columns, in this example 1, is called the *stride* and is often indicated with s. Stride $s = 2$ means simply that we shift our 3×3 region two columns to the right and two rows down at each step.

Something else that we need to discuss is the size of the selected region in the input matrix A. The dimensions of the selected region that we shifted around in the process must be the same as of the kernel used. If you use a 5×5 kernel, you will need to select a 5×5 region in A. In general, given a $n_K \times n_K$ kernel, you select a $n_K \times n_K$ region in A.

In a more formal definition, convolution with stride s in the neural network context is a process that takes a tensor A of dimensions $n_A \times n_A$ and a kernel K of dimensions $n_K \times n_K$ and gives as output a matrix B of dimensions $n_B \times n_B$ with

$$n_B = \left\lfloor \frac{n_A - n_K}{s} + 1 \right\rfloor$$

Where we have indicated with $\lfloor x \rfloor$ the integer part of x (in the programming world, this is often called the floor of x). A proof of this formula would take too long to discuss, but it's easy to see why it is true (try to derive it). To make things a bit easier, suppose that n_K is odd. You will see soon why this is important (although not fundamental). Let's start explaining formally the case with a stride $s = 1$. The algorithm generates a new tensor B from an input tensor A and a kernel K according to the formula

$$B_{ij} = (A * K)_{ij} = \sum_{f=0}^{n_K - 1} \sum_{h=0}^{n_K - 1} A_{i+f, j+h} K_{i+f, j+h}$$

The formula is cryptic and is very difficult to understand. Let's study some more examples to grasp the meaning better. In Figure 3-1, you can see a visual explanation of how convolution works. Suppose to have a 3 × 3 filter. Then in the Figure 3-1, you can see that the top left nine elements of the matrix *A*, marked by a square drawn with a black continuous line, are the one used to generate the first element of the matrix B_1 according to this formula. The elements marked by the square drawn with a dotted line are the ones used to generate the second element B_2 and so on.

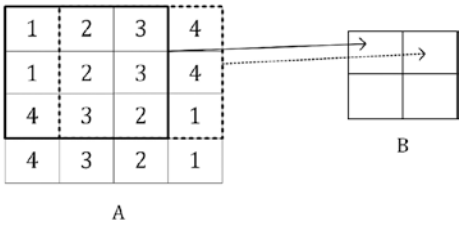

Figure 3-1. *A visual explanation of convolution*

To reiterate what we discuss in the example at the beginning, the basic idea is that each element of the 3 × 3 square from matrix *A* is multiplied by the corresponding element of the kernel *K* and all the numbers are summed. The sum is then the element of the new matrix *B*. After having calculated the value for B_1, you shift the region you are considering in the original matrix of one column to the right (the square indicated in Figure 3-1 with a dotted line) and repeat the operation. You continue to shift your region to the right until you reach the border and then you move one element down and start again from the left. You continue in this fashion until the lower right angle of the matrix. The same kernel is used for all the regions in the original matrix.

Given the kernel $\mathfrak{I}_H$ for example, you can see in Figure 3-2 which element of *A* are multiplied by which elements in $\mathfrak{I}_H$ and the result for the element B_1, that is nothing else as the sum of all the multiplications

$$B_{11} = 1\times1+2\times1+3\times1+1\times0+2\times0+3\times0+4\times(-1)+3\times(-1)+2\times(-1)=-3$$

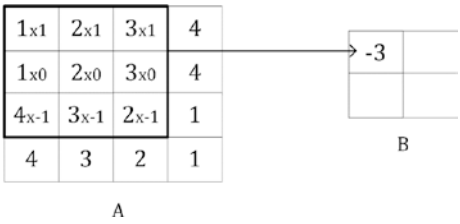

Figure 3-2. *A visualization of convolution with the kernel* $\mathfrak{I}_H$

In Figure 3-3, you can see an example of convolution with stride $s = 2$.

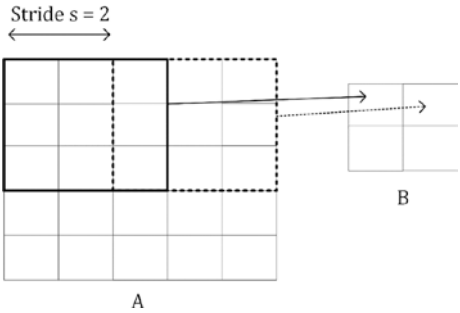

Figure 3-3. *A visual explanation of convolution with stride* $s = 2$

The reason that the dimension of the output matrix takes only the floor (the integer part) of

$$\frac{n_A - n_K}{s} + 1$$

Can be seen intuitively in Figure 3-4. If $s > 1$, what can happen, depending on the dimensions of A, is that at a certain point you cannot shift your window on matrix A (the black square you can see in Figure 3-3 for example) anymore, and you cannot cover all of matrix A completely. In Figure 3-4, you can see how you would need an additional column to the right of matrix A (marked by many X) to be able to perform the convolution

operation. In Figure 3-4, we chose $s = 3$, and since we have $n_A = 5$ and $n_K = 3$, B will be a scalar as a result.

$$n_B = \left\lfloor \frac{n_A - n_K}{s} + 1 \right\rfloor = \left\lfloor \frac{5-3}{3} + 1 \right\rfloor = \left\lfloor \frac{5}{3} \right\rfloor = 1$$

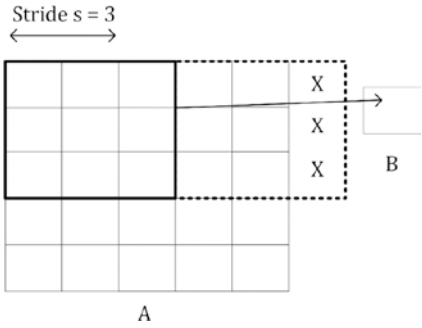

Figure 3-4. *A visual explanation of why the floor function is needed when evaluating the resulting matrix B dimensions*

You can easily see from Figure 3-4, how with a 3 × 3 region, one can only cover the top-left region of A, since with stride $s = 3$ you would end up outside A and therefore can consider one region only for the convolution operation. Therefore, you end up with a scalar for the resulting tensor B.

Let's now look at a few additional examples to make this formula even more transparent. Let's start with a small matrix 3 × 3

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Moreover, let's consider the kernel

$$K = \begin{pmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{pmatrix}$$

with stride $s = 1$. The convolution will be given by

$$B = A * K = 1 \cdot k_1 + 2 \cdot k_2 + 3 \cdot k_3 + 4 \cdot k_4 + 5 \cdot k_5 + 6 \cdot k_6 + 7 \cdot k_7 + 8 \cdot k_8 + 9 \cdot k_9$$

Moreover, the result B will be a scalar, since $n_A = 3$, $n_K = 3$.

$$n_B = \left\lfloor \frac{n_A - n_K}{s} + 1 \right\rfloor = \left\lfloor \frac{3-3}{1} + 1 \right\rfloor = 1$$

If you consider a matrix A with dimensions 4×4, or $n_A = 4$, $n_K = 3$ and $s = 1$, you will get as output a matrix B with dimensions 2×2, since

$$n_B = \left\lfloor \frac{n_A - n_K}{s} + 1 \right\rfloor = \left\lfloor \frac{4-3}{1} + 1 \right\rfloor = 2$$

For example, you can verify that given

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

And

$$K = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

We have with stride $s = 1$

$$B = A * K = \begin{pmatrix} 348 & 393 \\ 528 & 573 \end{pmatrix}$$

We'll verify one of the elements: B_{11} with the formula I gave you. We have

$$B_{11}=\sum_{f=0}^{2}\sum_{h=0}^{2}A_{1+f,1+h}K_{1+f,1+h}=\sum_{f=0}^{2}\left(\mathrm{A}_{1+\mathrm{f},\,1}K_{1+f,1}+\mathrm{A}_{1+\mathrm{f},\,2}K_{1+f,2}+\mathrm{A}_{1+\mathrm{f},\,3}K_{1+f,3}\right)$$
$$=\left(\mathrm{A}_{1,\,1}K_{1,1}+\mathrm{A}_{1,\,2}K_{1,2}+\mathrm{A}_{1,\,3}K_{1,3}\right)+\left(\mathrm{A}_{2,\,1}K_{2,1}+\mathbf{A}_{2,\,2}K_{2,2}+\mathrm{A}_{2,\,3}K_{2,3}\right)$$
$$+\left(\mathrm{A}_{3,\,1}K_{3,1}+\mathrm{A}_{3,\,2}K_{3,2}+\mathrm{A}_{3,\,3}K_{3,3}\right)=\left(1\cdot1+2\cdot2+\mathbf{3}\cdot3\right)+\left(5\cdot4+6\cdot5+7\cdot6\right)$$
$$+\left(9\cdot7+10\cdot8+11\cdot9\right)=14+92+242=348$$

Note that the formula I gave you for the convolution works only for stride $s = 1$, but can be easily generalized for other values of s.

This calculation is very easy to implement in Python. The following function can evaluate the convolution of two matrices easily enough for $s = 1$ (you can do it in Python with existing functions, but I think it's instructive to see how to do it from scratch):

```
import numpy as np
def conv_2d(A, kernel):
    output = np.zeros([A.shape[0]-(kernel.shape[0]-1),
    A.shape[1]-(kernel.shape[0]-1)])

    for row in range(1,A.shape[0]-1):
        for column in range(1, A.shape[1]-1):
            output[row-1, column-1] = np.tensordot(A[row-
            1:row+2, column-1:column+2], kernel)

    return output
```

Note that the input matrix A does not even need to a square one, but it is assumed that the kernel is and that its dimension n_K is odd. The previous example can be evaluated with the following code:

```
A = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]])
K = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(conv_2d(A,K))
```

This gives the result:

```
[[ 348. 393.]
[ 528. 573.]]
```

Examples of Convolution

Now let's try to apply the kernels we defined at the beginning to a test image and see the results. As a test image, let's create a chessboard of dimensions 160 × 160 pixels with the code:

```
chessboard = np.zeros([8*20, 8*20])
for row in range(0, 8):
    for column in range (0, 8):
        if ((column+8*row) % 2 == 1) and (row % 2 == 0):
            chessboard[row*20:row*20+20,
            column*20:column*20+20] = 1
        elif ((column+8*row) % 2 == 0) and (row % 2 == 1):
            chessboard[row*20:row*20+20,
            column*20:column*20+20] = 1
```

In Figure 3-5, you can see how the chessboard looks.

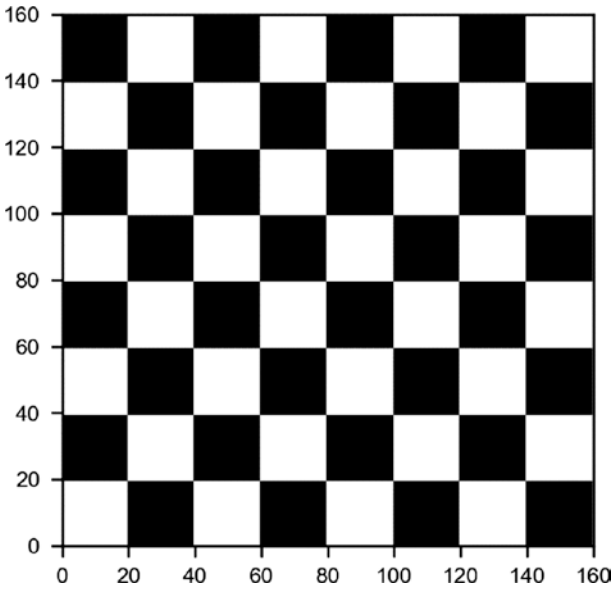

Figure 3-5. *The chessboard image generated with code*

Now let's apply convolution to this image with the different kernels with stride $s = 1$.

Using the kernel, $\mathfrak{I}_H$ will detect the horizontal edges. This can be applied with the code

```
edgeh = np.matrix('1 1 1; 0 0 0; -1 -1 -1')
outputh = conv_2d (chessboard, edgeh)
```

In Figure 3-6, you can see how the output looks. The image can be easily generated with this code:

```
Import matplotlib.pyplot as plt
plt.imshow(outputh)
```

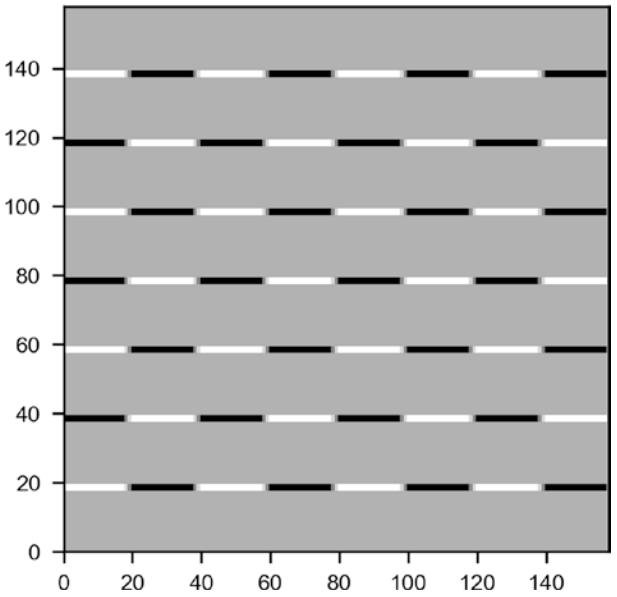

Figure 3-6. *The result of performing a convolution between the kernel* $\mathfrak{I}_H$ *and the chessboard image*

Now you can understand why this kernel detects horizontal edges. Additionally, this kernel detects when you go from light to dark or vice versa. Note this image is only 158 × 158 pixels, as expected, since

$$n_B = \left\lfloor \frac{n_A - n_K}{s} + 1 \right\rfloor = \left\lfloor \frac{160 - 3}{1} + 1 \right\rfloor = \left\lfloor \frac{157}{1} + 1 \right\rfloor = \lfloor 158 \rfloor = 158$$

Now let's apply $\mathfrak{I}_V$ using this code:

```
edgev = np.matrix('1 0 -1; 1 0 -1; 1 0 -1')
outputv = conv_2d (chessboard, edgev)
```

This gives the result shown in Figure 3-7.

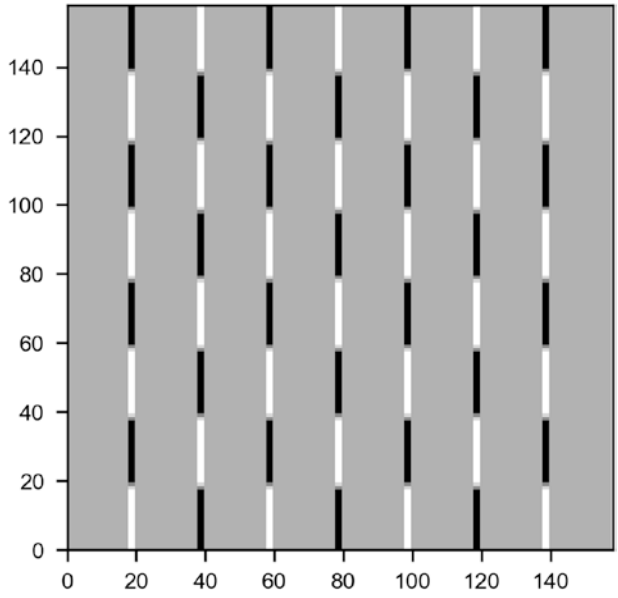

Figure 3-7. *The result of performing a convolution between the kernel* $\mathfrak{I}_V$ *and the chessboard image*

Now we can use kernel $\mathfrak{I}_L$:

```
edgel = np.matrix ('-1 -1 -1; -1 8 -1; -1 -1 -1')
outputl = conv_2d (chessboard, edgel)
```

That gives the result shown in Figure 3-8.

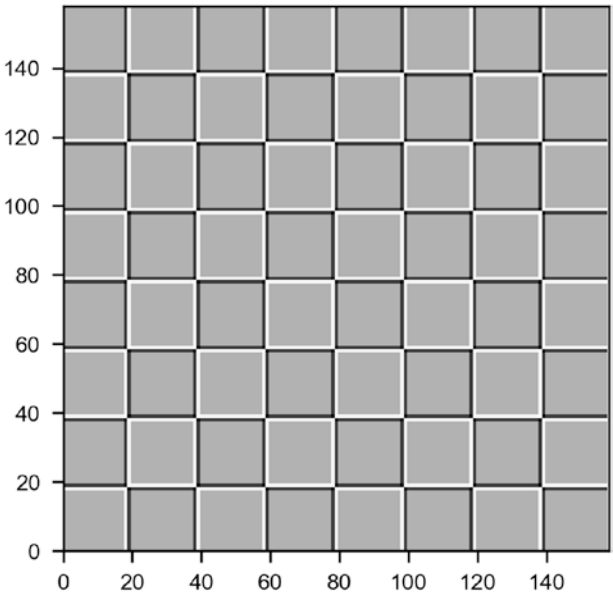

Figure 3-8. *The result of performing a convolution between the kernel* $\mathfrak{I}_L$ *and the chessboard image*

Moreover, we can apply the blurring kernel $\mathfrak{I}_B$:

```
edge_blur = -1.0/9.0*np.matrix('1 1 1; 1 1 1; 1 1 1')
output_blur = conv_2d (chessboard, edge_blur)
```

In Figure 3-9, you can see two plots—on the left the blurred image and on the right the original one. The images show only a small region of the original chessboard to make the blurring clearer.

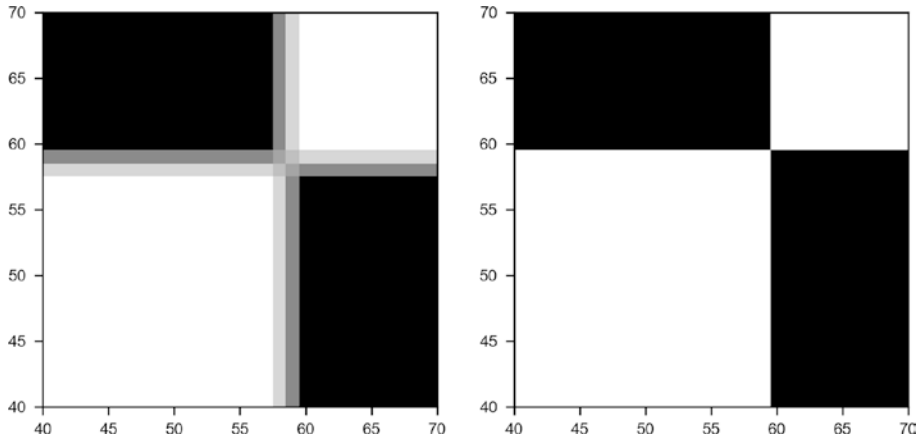

Figure 3-9. *The effect of the blurring kernel* $\mathfrak{I}_B$*. On the left is the blurred image and on the right is the original one.*

To finish this section, let's try to understand better how the edges can be detected. Consider the following matrix with a sharp vertical transition, since the left part is full of 10 and the right part full of 0.

```
ex_mat = np.matrix('10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0;
10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0;
10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0')
```

This looks like this

```
matrix([[10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0]])
```

Let's consider the kernel $\mathfrak{I}_V$. We can perform the convolution with this code:

```
ex_out = conv_2d (ex_mat, edgev)
```

The result is as follows:

```
array([[ 0., 0., 30., 30., 0., 0.],
       [ 0., 0., 30., 30., 0., 0.],
       [ 0., 0., 30., 30., 0., 0.],
       [ 0., 0., 30., 30., 0., 0.],
       [ 0., 0., 30., 30., 0., 0.],
       [ 0., 0., 30., 30., 0., 0.]])
```

In Figure 3-10, you can see the original matrix (on the left) and the output of the convolution on the right. The convolution with the kernel $\mathfrak{I}_V$ has clearly detected the sharp transition in the original matrix, marking with a vertical black line where the transition from black to white happens. For example, consider $B_{11} = 0$

$$B_{11} = \begin{pmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{pmatrix} * \mathfrak{I}_V = \begin{pmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{pmatrix} * \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$

$$= 10\times 1 + 10\times 0 + 10\times -1 + 10\times 1 + 10\times 0 + 10\times -1 + 10\times 1 + 10\times 0 + 10\times -1 = 0$$

Note that in the input matrix

$$\begin{pmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{pmatrix}$$

there is no transition, as all the values are the same. On the contrary, if you consider B_{13} you need to consider this region of the input matrix

$$\begin{pmatrix} 10 & 10 & 0 \\ 10 & 10 & 0 \\ 10 & 10 & 0 \end{pmatrix}$$

where there is a clear transition since the right-most column is made of zeros and the rest of 10. You get now a different result

$$B_{11} = \begin{pmatrix} 10 & 10 & 0 \\ 10 & 10 & 0 \\ 10 & 10 & 0 \end{pmatrix} * \mathfrak{I}_V = \begin{pmatrix} 10 & 10 & 0 \\ 10 & 10 & 0 \\ 10 & 10 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$
$$= 10\times 1 + 10\times 0 + 0\times -1 + 10\times 1 + 10\times 0 + 0\times -1 + 10\times 1 + 10\times 0 + 0\times -1 = 30$$

Moreover, this is precisely how, as soon as there is a significant change in values along the horizontal direction, the convolution returns a high value since the values multiplied by the column with 1 in the kernel will be more significant. When there is a transition from small to high values along the horizontal axis, the elements multiplied by -1 will give a result that is bigger in absolute value. Therefore the final result will be negative and big in absolute value. This is the reason that this kernel can also detect if you pass from a light color to a darker color and vice versa. If you consider the opposite transition (from 0 to 10) in a different hypothetical matrix A, you would have

$$B_{11} = \begin{pmatrix} 0 & 10 & 10 \\ 0 & 10 & 10 \\ 0 & 10 & 10 \end{pmatrix} * \mathfrak{I}_V = \begin{pmatrix} 0 & 10 & 10 \\ 0 & 10 & 10 \\ 0 & 10 & 10 \end{pmatrix} * \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$
$$= 0\times 1 + 10\times 0 + 10\times -1 + 0\times 1 + 10\times 0 + 10\times -1 + 0\times 1 + 10\times 0 + 10\times -1 = -30$$

We move from 0 to 10 along the horizontal direction.

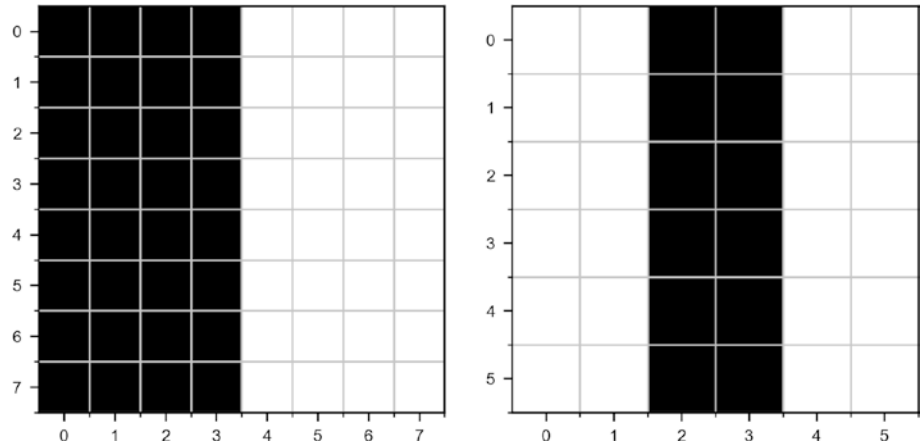

Figure 3-10. *The result of the convolution of the matrix* `ex_mat` *with the kernel* $\mathfrak{I}_V$ *as described in the text*

Note how, as expected, the output matrix has dimensions 5 × 5 since the original matrix has dimensions 7 × 7 and the kernel is 3 × 3.

Pooling

Pooling is the second operation that is fundamental in CNNs. This operation is much easier to understand than convolution. To understand it, let's look at a concrete example and consider what is called *max pooling*. Consider the 4 × 4 matrix we discussed during our convolution discussion again:

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

To perform max pooling, we need to define a region of size $n_K \times n_K$, analogous to what we did for convolution. Let's consider $n_K = 2$. What we need to do is start on the top-left corner of our matrix A and select a $n_K \times n_K$ region, in our case 2×2 from A. Here we would select

$$\begin{pmatrix} a_1 & a_2 \\ a_5 & a_6 \end{pmatrix}$$

Alternatively, the elements marked in boldface in the matrix A here:

$$A = \begin{pmatrix} \boldsymbol{a_1} & \boldsymbol{a_2} & a_3 & a_4 \\ \boldsymbol{a_5} & \boldsymbol{a_6} & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

From the elements selected, a_1, a_2, a_5 and a_6, the max pooling operation selects the maximum value. The result is indicated with B_1

$$B_1 = \max_{i=1,2,5,6} a_i$$

We then need to shift our 2×2 window two columns to the right, typically the same number of columns the selected region has, and select the elements marked in bold:

$$A = \begin{pmatrix} a_1 & a_2 & \boldsymbol{a_3} & \boldsymbol{a_4} \\ a_5 & a_6 & \boldsymbol{a_7} & \boldsymbol{a_8} \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

Or, in other words, the smaller matrix

$$\begin{pmatrix} a_3 & a_4 \\ a_7 & a_8 \end{pmatrix}$$

The max-pooling algorithm will then select the maximum of the values and give a result that we will indicate with B_2

$$B_2 = \max_{i=3,4,7,8} a_i$$

At this point we cannot shift the 2 × 2 region to the right anymore, so we shift it two rows down and start the process again from the left side of A, selecting the elements marked in bold and getting the maximum and calling it B_3.

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ \boldsymbol{a_9} & \boldsymbol{a_{10}} & a_{11} & a_{12} \\ \boldsymbol{a_{13}} & \boldsymbol{a_{14}} & a_{15} & a_{16} \end{pmatrix}$$

The stride s in this context has the same meaning we have already discussed in convolution. It's simply the number of rows or columns you move your region when selecting the elements. Finally, we select the last region 2 × 2 in the bottom-lower part of A, selecting the elements a_{11}, a_{12}, a_{15}, and a_{16}. We then get the maximum and call it B_4. With the values we obtain in this process, in the example the four values B_1, B_2, B_3 and B_4, we will build an output tensor:

$$B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

In the example, we have $s = 2$. Basically, this operation takes as input a matrix A, a stride s, and a kernel size n_K (the dimension of the region we selected in the example before) and returns a new matrix B with dimensions given by the same formula we discussed for convolution:

$$n_B = \left\lfloor \frac{n_A - n_K}{s} + 1 \right\rfloor$$

To reiterate this idea, start from the top-left of matrix A, take a region of dimensions $n_K \times n_K$, apply the max function to the selected elements, then shift the region of s elements toward the right, select a new region again of dimensions $n_K \times n_K$, apply the function to its values, and so on. In Figure 3-11 you can see how you would select the elements from matrix A with stride $s = 2$.

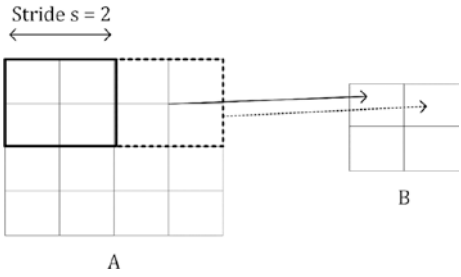

Figure 3-11. *A visualization of pooling with stride s = 2*

For example, applying max-pooling to the input A

$$A = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 4 & 5 & 11 & 3 \\ 4 & 1 & 21 & 6 \\ 13 & 15 & 1 & 2 \end{pmatrix}$$

Will get you this result (it's very easy to verify it):

$$B = \begin{pmatrix} 4 & 11 \\ 15 & 21 \end{pmatrix}$$

Since four is the maximum of the values marked in bold.

$$A = \begin{pmatrix} \mathbf{1} & \mathbf{3} & 5 & 7 \\ \mathbf{4} & \mathbf{5} & 11 & 3 \\ 4 & 1 & 21 & 6 \\ 13 & 15 & 1 & 2 \end{pmatrix}$$

Eleven is the maximum of the values marked in bold here:

$$A = \begin{pmatrix} 1 & 3 & \mathbf{5} & \mathbf{7} \\ 4 & 5 & \mathbf{11} & \mathbf{3} \\ 4 & 1 & 21 & 6 \\ 13 & 15 & 1 & 2 \end{pmatrix}$$

And so on. It's worth mentioning another way of doing pooling, although it's not as widely used as max-pooling: ***average pooling***. Instead of returning the maximum of the selected values, it returns the average.

Note The most commonly used pooling operation is *max pooling*. Average pooling is not as widely used but can be found in specific network architectures.

Padding

Something that's worth mentioning here is *padding*. Sometimes, when dealing with images, it is not optimal to get a result from a convolution operation that has dimensions that are different from the original image. This is when padding is necessary. The idea is straightforward: you add rows of pixels on the top and bottom and columns of pixels on the right and left of the final images so the resulting matrices are the same size as the original. Some strategies fill the added pixels with zeros, with the values of the closest pixels and so on. For example, in our example, our `ex_out` matrix with zero padding would like like this

```
array([[ 0., 0., 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 30., 30., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0., 0., 0.]])
```

Only as a reference, in case you use padding p (the width of the rows and columns you use as padding), the final dimensions of the matrix B, in case of both convolution and pooling, is given by

$$n_B = \left\lfloor \frac{n_A + 2p - n_K}{s} + 1 \right\rfloor$$

Note When dealing with real images, you always have color images, coded in three channels: RGB. That means that convolution and pooling must be done in three dimensions: width, height, and color channel. This will add a layer of complexity to the algorithms.

Building Blocks of a CNN

Convolution and pooling operations are used to build the layers used in CNNs. In CNNs typically you can find the following layers

- Convolutional layers
- Pooling layers
- Fully connected layers

Fully connected layers are precisely what we have seen in all the previous chapters: a layer where neurons are connected to all neurons of previous and subsequent layers. You know them already. The other two require some additional explanation.

Convolutional Layers

A convolutional layer takes as input a tensor (which can be three-dimensional, due to the three color channels), for example an image, applies a certain number of kernels, typically 10, 16, or even more, adds a bias, applies ReLu activation functions (for example) to introduce non-linearity to the result of the convolution, and produces an output matrix B.

Now in the previous sections, I showed you some examples of applying convolutions with just one kernel. How can you apply several kernels at the same time? Well, the answer is straightforward. The final tensor (I use now the word tensor since it will not be a simple matrix anymore) B will have not two dimensions but three. Let's indicate the number of kernels you want to apply with n_c (the c is used since sometimes people talk about channels). You simply apply each filter to the input independently and stack the results. So instead of a single matrix B with dimensions $n_B \times n_B$ you get a final tensor $\tilde{B}$ of dimensions $n_B \times n_B \times n_c$. That means that this

$$\tilde{B}_{i,j,1} \quad \forall i,j \in [1,n_B]$$

Will be the output of convolution of the input image with the first kernel, and

$$\tilde{B}_{i,j,2} \quad \forall i,j \in [1,n_B]$$

Will be the output of convolution with the second kernel, and so on. The convolution layer simply transforms the input into an output tensor. However, what are the weights in this layer? The weights, or the parameters that the network learns during the training phase, are the elements of the kernel themselves. We discussed that we have n_c kernels, each of $n_K \times n_K$ dimensions. That means that we have $n_K^2 n_c$ parameter in a convolutional layer.

Note The number of parameters that you have in a convolutional layer, $n_K^2 n_c$, is independent from the input image size. This fact helps in reducing overfitting, especially when dealing with large input images.

Sometimes this layer is indicated with the word POOL and then a number. In our case, we could indicate this layer with POOL1. In Figure 3-12, you can see a representation of a convolutional layer. The input image is transformed by applying convolution with n_c kernels in a tensor of dimensions $n_A \times n_A \times n_c$.

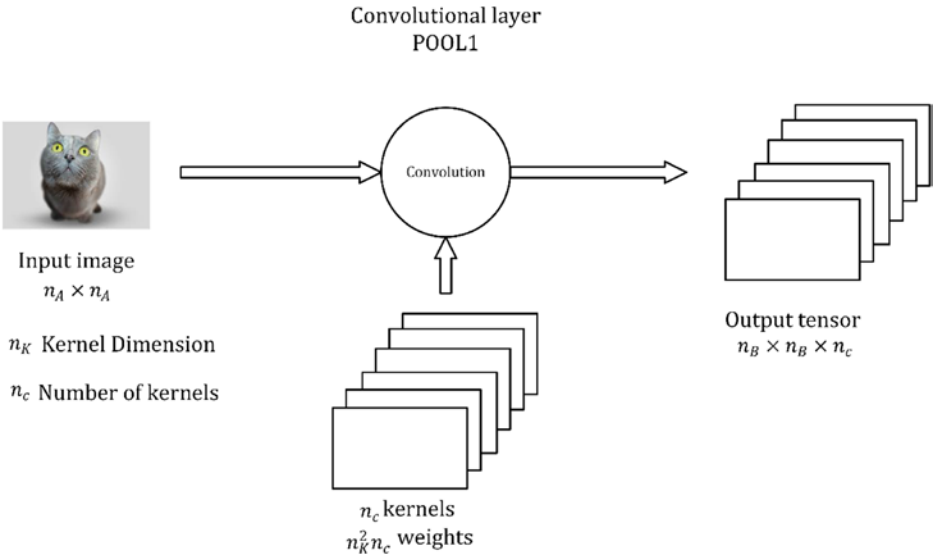

Figure 3-12. *A representation of a convolutional layer*[2]

Of course, a convolutional layer must not necessarily be placed immediately after the inputs. A convolutional layer may get as input the output of any other layer of course. Keep in mind that usually, the input image will have dimensions $n_A \times n_A \times 3$, since an image in color has three channels: Red, Green, and Blue. A complete analysis of the tensors involved in a CNN when considering color images is beyond the scope of this book. Very often in diagrams, the layer is simply indicated as a cube or a square.

[2]Cat image source: https://www.shutterstock.com/

Pooling Layers

A pooling layer is usually indicated with `POOL` and a number: for example, `POOL1`. It takes as input a tensor and gives as output another tensor after applying pooling to the input.

Note A pooling layer has no parameter to learn, but it introduces additional hyperparameters: n_K and stride v. Typically, in pooling layers, you don't use any padding, since one of the reasons to use pooling is often to reduce the dimensionality of the tensors.

Stacking Layers Together

In CNNs you usually stack convolutional and pooling layer together. One after the other. In Figure 3-13, you can see a convolutional and a pooling layer stack. A convolutional layer is always followed by a pooling layer. Sometimes the two together are called a *layer*. The reason is that a pooling layer has no learnable weights and therefore it is merely seen as a simple operation that is associated with the convolutional layer. So be aware when you read papers or blogs and check what they intend.

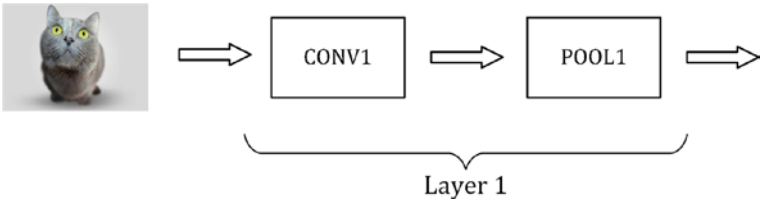

Figure 3-13. *A representation of how to stack convolutional and pooling layers*

To conclude this part of CNN in Figure 3-14, you can see an example of a CNN. In Figure 3-14, you see an example like the very famous LeNet-5 network, about which you can read more here: `https://goo.gl/hM1kAL`. You have the inputs, then two times convolution-pooling layer, then three fully connected layers, and then an output layers, with a `softmax` activation function to perform multiclass classification. I put some indicative numbers in the figure to give you an idea of the size of the different layers.

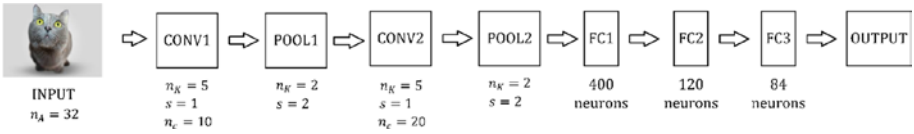

Figure 3-14. *A representation of a CNN similar to the famous LeNet-5 network*

Number of Weights in a CNN

It is important to point out where the weights in a CNN are in the different layers.

Convolutional Layer

In a convolutional layer, the parameters that are learned are the filters themselves. For example, if you have 32 filters, each of dimension 5x5, you will get 32x5x5=832 learnable parameters, since for each filter there is also a bias term that you will need to add. Note that this number is not dependent on the input image size. In a typical feed-forward neural network, the number of weights in the first layer is dependent on the input size, but not here.

CHAPTER 3 FUNDAMENTALS OF CONVOLUTIONAL NEURAL NETWORKS

The number of weights in a convolutional layer is, in general terms, given by the following:

$$n_C \cdot n_K \cdot n_K + n_C$$

Pooling Layer

The pooling layer has no learnable parameters, and as mentioned, this is the reason it's typically associated with the convolutional layer. In this layer (operation), there are no learnable weights.

Dense Layer

In this layer, the weights are the ones you know from traditional feed-forward networks. So the number depends on the number of neurons and the number of neurons in the preceding and subsequent layers.

Note The only layers in a CNN that have learnable parameters are the convolutional and dense layers.

Example of a CNN: MNIST Dataset

Let's start with some coding. We will develop a very simple CNN and try to do classification on the MNIST dataset. You should know the dataset very well by now, from Chapter 2.

We start, as usual, by importing the necessary packages:

```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D,
MaxPool2D
from keras.utils import np_utils
```

```
import numpy as np
import matplotlib.pyplot as plt
```

We need an additional step before we can start loading the data:

```
from keras import backend as K
K.set_image_dim_ordering('th')
```

The reason is the following. When you load images for your model, you will need to convert them to tensors, each with three dimensions:

- Number of pixels along the x-axis
- Number of pixels along the y-axis
- Number of color channels (in a gray image, this number is; if you have color images, this number is 3, one for each of the RGB channels)

When doing convolution, Keras must know on which axis it finds the information. In particular, it is relevant to define if the index of the color channel's dimension is the first or the last. To achieve this, we can define the ordering of the data with `keras.backend.set_image_dim_ordering()`. This function accepts as input a string that can assume two possible values:

- `'th'` (for the convention used by the library Theano): Theano expects the channel dimensions to be the second one (the first one will be the observation index).
- `'tf'` (for the convention used by TensorFlow): TensorFlow expects the channel dimension to be the last one.

You can use both, but simply pay attention when preparing the data to use the right convention. Otherwise, you will get error messages about tensor dimensions. In what follows, we will convert images in tensors having the color channel dimensions as the second one, as you can see later.

Now we are ready to load the MNIST data with this code:

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

The code will deliver the images "flattened," meaning each image will be a one-dimensional vector of 784 elements (28x28). We need to reshape them as proper images, since our convolutional layers want images as inputs. After that, we need to normalize the data (remember the images are in a grayscale, and each pixel can have a value from 0 to 255).

```
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).
astype('float32')
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).
astype('float32')
X_train = X_train / 255.0
X_test = X_test / 255.0
```

Note how, since we have defined the ordering as `'th'`, the number of channels (in this case 1) is the second element of the `X` arrays. As a next step, we need to one-hot-encode the labels:

```
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
```

We know we have 10 classes so we can simply define them:

```
num_classes = 10
```

Now let's define a function to create and compile our Keras model:

```
def baseline_model():
    # create model
    model = Sequential()
    model.add(Conv2D(32, (5, 5), input_shape=(1, 28, 28),
    activation='relu'))
    model.add(MaxPool2D(pool_size=(2, 2)))
```

```
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
# Compile model
model.compile(loss='categorical_crossentropy',
optimizer='adam', metrics=['accuracy'])
return model
```

You can see a diagram of this CNN in Figure 3-15.

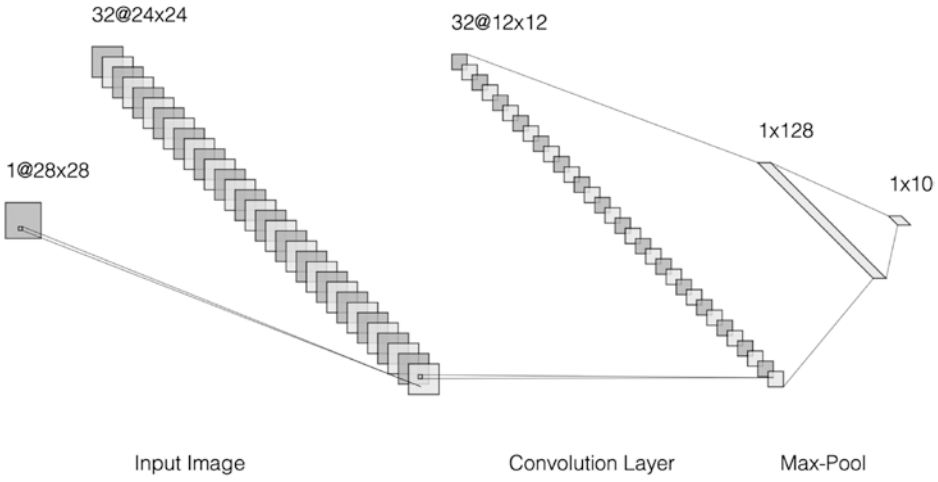

Figure 3-15. *A diagram depicting the CNN we used in the text. The numbers are the dimensions of the tensors produced by each layer.*

To determine what kind of model we have, we simply use the `model.summary()` call. Let's first create a model and then check it:

```
model = baseline_model()
model.summary()
```

The output (check the diagram form in Figure 3-15) is as follows:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 24, 24)        832
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 32, 12, 12)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 32, 12, 12)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 4608)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               589952
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 592,074
Trainable params: 592,074
Non-trainable params: 0
```

In case you are wondering why the max-pooling layer produces tensors of dimensions 12x12, the reason is that since we haven't specified the stride, Keras will take as a standard value the dimension of the filter, which in our case is 2x2. Having input tensors that are 24x24 with stride 2 you will get tensors that are 12x12.

This network is quite simple. In the model we defined just one convolutional and pooling layer, we added a bit of dropout, then we added a dense layer with 128 neurons and then an output layer for the `softmax` with 10 neurons. Now we can simply train it with the `fit()` method:

```
model.fit(X_train, y_train, validation_data=(X_test, y_test),
epochs=1, batch_size=200, verbose=1)
```

This will train the network for only one epoch and should give you output similar to this (your numbers may vary a bit):

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/1
60000/60000 [==============================] - 151s 3ms/step -
loss: 0.0735 - acc: 0.9779 - val_loss: 0.0454 - val_acc: 0.9853
```

We have already reached good accuracy, without having any overfitting.

Note When you pass to the `compile()` method the optimizer parameter, Keras will use its standard parameters. If you want to change them, you need to define an optimizer separately. For example, to specify an Adam optimizer with a starting learning rate of 0.001 you can use `AdamOpt = adam(lr=0.001)` and then pass it to the compile method with `model.compile(optimizer=AdamOpt, loss='categorical_crossentropy', metrics=['accuracy'])`.

Visualization of CNN Learning

Brief Digression: keras.backend.function()

Sometime it's useful to get intermediate results from a computational graph. For example, you may be interested in the output of a specific layer for debugging purposes. In low-level TensorFlow, you can simply evaluate in the session the relevant node in the graph, but it's not so easy to understand how to do it in Keras. To find out, we need to consider what

the Keras backend is. The best way of explaining it is to cite the official documentation (https://keras.io/backend/):

> *Keras is a model-level library, providing high-level building blocks for developing deep learning models. It does not handle low-level operations such as tensor products, convolutions, and so on itself. Instead, it relies on a specialized, well optimized tensor manipulation library to do so, serving as the "backend engine" of Keras.*

To be complete, it is important to note that Keras uses (at the time of writing) three backends: the *TensorFlow* backend, the *Theano* backend, and the *CNTK* backend. When you want to write your own specific function, you should use the abstract Keras backend API that can be loaded with this code:

```
from keras import backend as K
```

Understanding how to use the Keras backend goes beyond the scope of this book (remember the focus of this book is not Keras), but I suggest you spend some time getting to know it. It may be very useful. For example, to reset a session when using Keras you can use this:

```
keras.backend.clear_session()
```

What we are really interested in this chapter is a specific method available in the backed: `function()`. Its arguments are as follows:

- inputs: List of placeholder tensors
- outputs: List of output tensors
- updates: List of update ops
- `**kwargs`: Passed to `tf.Session.run`

We will use only the first two in this chapter. To understand how to use them, let's consider for example the model we created in the previous sections:

```
model = Sequential()
model.add(Conv2D(32, (5, 5), input_shape=(1, 28, 28),
activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

How do we get, for example, the output of the first convolutional layer? We can do this easily by creating a function:

```
get_1st_layer_output = K.function([model.layers[0].
input],[model.layers[0].output])
```

This will use the following arguments

- inputs: `model.layers[0].input`, which is the input of our network
- outputs: `model.layers[0].output`, which is the output of the first layer (with index 0)

You simply ask Keras to evaluate specific nodes in your computational graph, given a specific set of inputs. Note that up to now we have only defined a function. Now we need to apply it to a specific dataset. For example, if we want to apply it to one single image, we could do this:

```
layer_conv_output = get_1st_layer_output([np.expand_dims(X_
test[21], axis=0)])[0]
```

This multidimensional array will have dimensions `(1, 32, 24, 24)` as expected: one image, 32 filters, 24x24 output. In the next section, we will use this function to see the effect of the learned filter in the network.

Effect of Kernels

It is interesting to see what the effect of the learned kernels is on the input image. For this purpose, let's take an image from the test dataset (if you shuffled your dataset, you may get a different digit at index 21).

```
tst = X_test[21]
```

Note how this array has dimensions `(1,28,28)`. This is a six, as you can see in Figure 3-16.

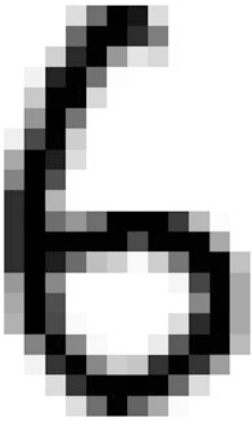

Figure 3-16. *The first image in the test dataset*

To get the effect of the first layer (the convolutional one), we can use the following code (explained in the previous section)

```
get_1st_layer_output = K.function([model.layers[0].
input],[model.layers[0].output])
layer_conv_output = get_1st_layer_output([tst])[0]
```

Note how the `layer_conv_output` is a multidimensional array, and it will contain the convolution of the input image with each filter, stacked on top of each other. Its dimensions are (1,32,24,24). The first number is 1 since we applied the layer only to one single image, the second, 32, is the number of filters we have, and the second is 24 since, as we have discussed, the output tensor dimensions of a `conv` layer are given by

$$n_B = \left\lfloor \frac{n_A + 2p - n_K}{s} + 1 \right\rfloor$$

Moreover, in our case

$$n_B = \left\lfloor \frac{28 - 5}{1} + 1 \right\rfloor = 24$$

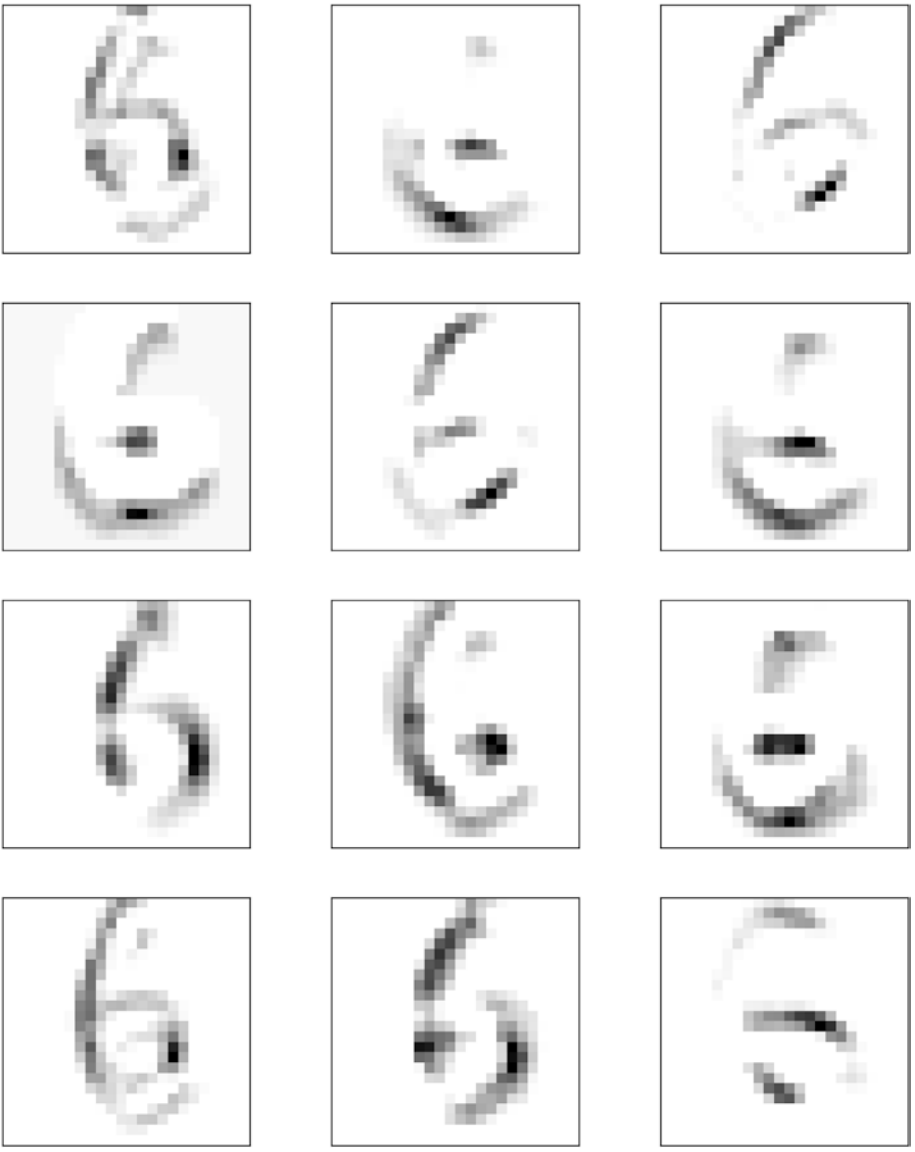

Figure 3-17. *The test image (a 6) convoluted with the first 12 filters learned by the network*

Since in our network, we have $n_A = 28$, $p = 0$, $n_K = 5$, and stride $s = 1$. In Figure 3-17, you can see our test image convoluted with the first 12 filters (32 was too many for a figure).

From Figure 3-17 you can see how different filters learn to detect different features. For example, the third filter (as can be seen in Figure 3-18) learned to detect diagonal lines.

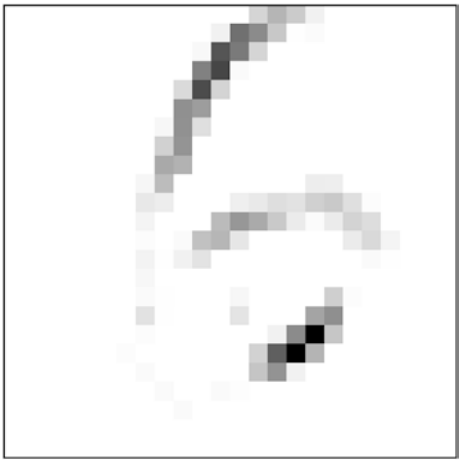

Figure 3-18. *The test image convoluted with the third filter. It learned to detect diagonal lines.*

Other filters learn to detect horizontal lines or other features.

Effect of Max-Pooling

The subsequent step is to apply max pooling to the output of the convolutional layer. As we discussed, this will reduce the dimensions of the tensors and will try to (intuitively) condense the relevant information.

You can see the output on the tensor coming from the first 12 filters in Figure 3-19.

Figure 3-19. *The output of the pooling layer when applied to the first 12 tensors coming from the convolutional layer*

Let's see how our test image is transformed from one convolutional and the pooling layer by one of the filters (consider the third, just for illustration purposes). You can easily see the effect in Figure 3-20.

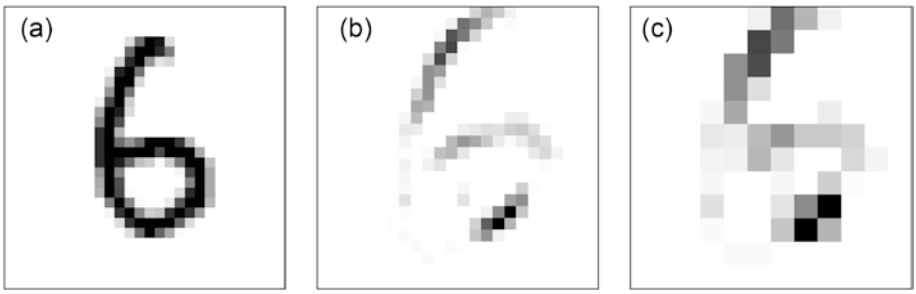

Figure 3-20. *The original test image as in the dataset (in panel a); the image convoluted with the third learned filter (panel b); the image convoluted with the third filter after the max pooling layer (panel c)*

Note how the resolution of the image is changing, since we are not using any padding. In the next chapter, we will look at more complicated architectures, called *inception networks,* that are known to work better than traditional CNN (what we have described in this chapter) when dealing with images. In fact, simply adding more and more convolutional layers will not easily increase the accuracy of the predictions, and more complex architecture are known to be much more effective.

Now that we have seen the very basic building blocks of a CNN, we are ready to move to some more advanced topics. In the next chapter, we will look at many exciting topics as inception networks, multiple loss functions, custom loss functions, and transfer learning.