



11. Convolutional Neural Networks

Our entire review of machine learning and neural networks thus far have been leading up to this point: ***understanding Convolutional Neural Networks (CNNs)*** and the role they play in deep learning.

In traditional feedforward neural networks (like the ones we studied in Chapter 10), each neuron in the input layer is connected to every output neuron in the next layer – we call this a *fully-connected* (FC) layer. However, in CNNs, we don't use FC layers until the *very last layer(s)* in the network. We can thus define a CNN as a neural network that swaps in a specialized “convolutional” layer in place of “fully-connected” layer for at least *one* of the layers in the network [10].

A nonlinear activation function, such as ReLU, is then applied to the output of these convolutions and the process of convolution => activation continues (along with a mixture of other layer types to help reduce the width and height of the input volume and help reduce overfitting) until we finally reach the end of the network and apply one or two FC layers where we can obtain our final output classifications.

Each layer in a CNN applies a different set of filters, typically hundreds or thousands of them, and combines the results, feeding the output into the next layer in the network. During training, **a CNN automatically learns the values for these filters.**

In the context of image classification, our CNN may learn to:

- Detect edges from raw pixel data in the first layer.
- Use these edges to detect shapes (i.e., “blobs”) in the second layer.
- Use these shapes to detect higher-level features such as facial structures, parts of a car, etc. in the highest layers of the network.

The last layer in a CNN uses these higher-level features to make predictions regarding the contents of the image. In practice, CNNs give us two key benefits: *local invariance* and *compositionality*. The concept of *local invariance* allows us to classify an image as containing a particular object *regardless* of where in the image the object appears. We obtain this local invariance through the usage of “pooling layers” (discussed later in this chapter) which identifies regions of our input volume with a high response to a particular filter.

The second benefit is compositionality. Each filter composes a local patch of lower-level features

into a higher-level representation, similar to how we can compose a set of mathematical functions that build on the output of previous functions: $f(g(h(x)))$ – this composition allows our network to learn more rich features deeper in the network. For example, our network may build edges from pixels, shapes from edges, and then complex objects from shapes – all in an automated fashion that happens *naturally* during the training process. The concept of building higher-level features from lower-level ones is exactly why CNNs are so powerful in computer vision.

In the rest of this chapter, we’ll discuss exactly *what* convolutions are and the role they play in deep learning. We’ll then move on to the building blocks of CNNs: layers, and the various types of layers you’ll use to build your own CNNs. We’ll wrap up this chapter by looking at common patterns that are used to stack these building blocks to create CNN architectures that perform well on a diverse set of image classification tasks.

After reviewing this chapter, we’ll have (1) a strong understanding of Convolutional Neural Networks and the thought process that goes into building one and (2) a number of CNN “recipes” we can use to construct our own network architectures. In our next chapter, we’ll use these fundamentals and recipes to train CNNs of our own.

11.1 Understanding Convolutions

In this section, we’ll address a number of questions, including:

- *What* are image convolutions?
- *What* do they do?
- *Why* do we use them?
- *How* do we apply them to images?
- **And what role do convolutions play in deep learning?**

The word “convolution” sounds like a fancy, complicated term – but it’s really not. If you have any prior experience with computer vision, image processing, or OpenCV before, you’ve *already* applied convolutions, ***whether you realize it or not!***

Ever apply *blurring* or *smoothing* to an image? Yep, that’s a convolution. What about *edge detection*? Yup, convolution. Have you opened Photoshop or GIMP to *sharpen* an image? You guessed it – convolution. Convolutions are one of the most *critical, fundamental building-blocks* in computer vision and image processing.

But the term itself tends to scare people off – in fact, on the surface, the word even appears to have a negative connotation (why would anyone want to “convolute” something?) Trust me, convolutions are anything but scary. They’re actually quite easy to understand.

In terms of deep learning, an (image) ***convolution is an element-wise multiplication of two matrices followed by a sum.***

Seriously. That’s it. *You just learned what a convolution is:*

1. Take two matrices (which both have the same dimensions).
2. Multiply them, element-by-element (i.e., *not* the dot product, just a simple multiplication).
3. Sum the elements together.

We’ll learn more about convolutions, kernels, and how they are used inside CNNs in the remainder of this section.

11.1.1 Convolutions versus Cross-correlation

A reader with prior background in computer vision and image processing may have identified my description of a *convolution* above as a *cross-correlation* operation instead. Using cross-correlation instead of convolution is actually by design. Convolution (denoted by the \star operator) over a

two-dimensional input image I and two-dimensional kernel K is defined as:

$$S(i, j) = (I \star K)(i, j) = \sum_m \sum_n I(i - m, j - n) K(m, n) \quad (11.1)$$

However, nearly all machine learning and deep learning libraries use the simplified *cross-correlation* function

$$S(i, j) = (I \star K)(i, j) = \sum_m \sum_n I(i + m, j + n) K(m, n) \quad (11.2)$$

All this math amounts to is a sign change in how we access the coordinates of the image I (i.e., we don't have to "flip" the kernel relative to the input when applying cross-correlation).

Again, many deep learning libraries use the simplified cross-correlation operation and call it convolution – **we will use the same terminology here**. For readers interested in learning more about the mathematics behind convolution vs. cross-correlation, please refer to Chapter 3 of *Computer Vision: Algorithms and Applications* by Szelski [124].

11.1.2 The "Big Matrix" and "Tiny Matrix" Analogy

An image is a *multidimension matrix*. Our image has a width (# of columns) and height (# of rows), just like a matrix. But unlike traditional matrices you have worked with back in grade school, images also have a *depth* to them – the number of *channels* in the image.

For a standard RGB image, we have a depth of 3 – one channel for *each* of the Red, Green, and Blue channels, respectively. Given this knowledge, we can think of an image as *big matrix* and a *kernel* or *convolutional matrix* as a *tiny matrix* that is used for blurring, sharpening, edge detection, and other processing functions. Essentially, this *tiny* kernel sits on top of the *big* image and slides from left-to-right and top-to-bottom, applying a mathematical operation (i.e., a *convolution*) at each (x, y) -coordinate of the original image.

It's normal to hand-define kernels to obtain various image processing functions. In fact, you might already be familiar with blurring (average smoothing, Gaussian smoothing, median smoothing, etc.), edge detection (Laplacian, Sobel, Scharr, Prewitt, etc.), and sharpening – *all* of these operations are forms of hand-defined kernels that are *specifically designed* to perform a particular function.

So that raises the question: *is there a way to automatically learn these types of filters?* And even use these filters for *image classification* and *object detection*? **You bet there is.** But before we get there, we need to understand kernels and convolutions a bit more.

11.1.3 Kernels

Again, let's think of an image as a *big matrix* and a kernel as a *tiny matrix* (at least in respect to the original "big matrix" image), depicted in Figure 11.1. As the figure demonstrates, we are sliding the kernel (red region) from left-to-right and top-to-bottom along the original image. At each (x, y) -coordinate of the original image, we stop and examine the neighborhood of pixels located at the **center** of the image kernel. We then take this neighborhood of pixels, *convolve* them with the kernel, and obtain a single output value. The output value is stored in the output image at the same (x, y) -coordinates as the center of the kernel.

If this sounds confusing, no worries, we'll be reviewing an example in the next section. But before we dive into an example, let's take a look at what a kernel looks like (Figure 11.3):

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad (11.3)$$

131	162	232	84	91	207
104	91	109	+11	237	109
243	22	202	+25	135	126
185	135	200	+1	61	225
157	124	25	14	102	108
5	155	116	218	232	249

Figure 11.1: A kernel can be visualized as a small matrix that slides across, from left-to-right and top-to-bottom, of a larger image. At each pixel in the input image, the neighborhood of the image is convolved with the kernel and the output stored.

Above we have defined a square 3×3 kernel (any guesses on what this kernel is used for?). Kernels can be of arbitrary rectangular size $M \times N$, provided that **both** M and N are *odd integers*.

R Most kernels applied to deep learning and CNNs are $N \times N$ *square* matrices, allowing us to take advantage of optimized linear algebra libraries that operate most efficiently on square matrices.

We use an *odd* kernel size to ensure there is a valid integer (x, y) -coordinate at the center of the image (Figure 11.2). On the *left*, we have a 3×3 matrix. The center of the matrix is located at $x = 1, y = 1$ where the top-left corner of the matrix is used as the origin and our coordinates are zero-indexed. But on the *right*, we have a 2×2 matrix. The center of this matrix would be located at $x = 0.5, y = 0.5$.

But as we know, without applying interpolation, there is no such thing as pixel location $(0.5, 0.5)$ – our pixel coordinates must be integers! This reasoning is exactly why we use *odd* kernel sizes: to always ensure there is a valid (x, y) -coordinate at the center of the kernel.

11.1.4 A Hand Computation Example of Convolution

Now that we have discussed the basics of kernels, let's discuss the actual convolution operation and see an example of it actually being applied to help us solidify our knowledge. In image processing, a convolution requires three components:

1. An input image.
2. A kernel matrix that we are going to apply to the input image.

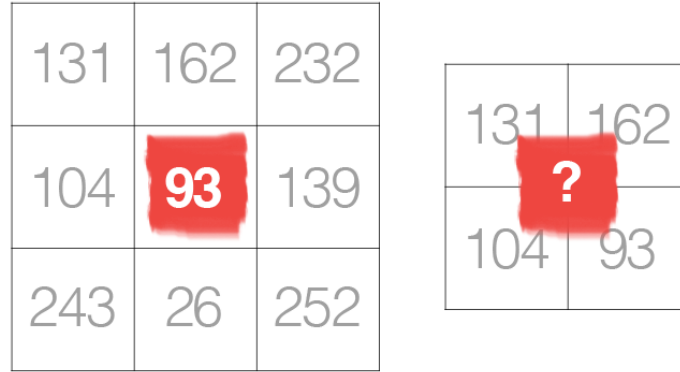


Figure 11.2: **Left:** The center pixel of a 3×3 kernel is located at coordinate $(1, 1)$ (highlighted in red). **Right:** What is the center coordinate of a kernel of size 2×2 ?

3. An output image to store the output of the image convolved with the kernel.
- Convolution (i.e., cross-correlation) is actually very easy. All we need to do is:
1. Select an (x, y) -coordinate from the original image.
 2. Place the **center** of the kernel at this (x, y) -coordinate.
 3. Take the element-wise multiplication of the input image region and the kernel, then sum up the values of these multiplication operations into a single value. The sum of these multiplications is called the **kernel output**.
 4. Use the same (x, y) -coordinates from **Step #1**, but this time, store the kernel output at the same (x, y) -location as the output image.

Below you can find an example of convolving (denoted mathematically as the \star operator) a 3×3 region of an image with a 3×3 kernel used for blurring:

$$O_{i,j} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \star \begin{bmatrix} 93 & 139 & 101 \\ 26 & 252 & 196 \\ 135 & 230 & 18 \end{bmatrix} = \begin{bmatrix} 1/9 \times 93 & 1/9 \times 139 & 1/9 \times 101 \\ 1/9 \times 26 & 1/9 \times 252 & 1/9 \times 196 \\ 1/9 \times 135 & 1/9 \times 230 & 1/9 \times 18 \end{bmatrix} \quad (11.4)$$

Therefore,

$$O_{i,j} = \sum \begin{bmatrix} 10.3 & 15.4 & 11.2 \\ 2.8 & 28.0 & 21.7 \\ 15.0 & 25.5 & 2.0 \end{bmatrix} \approx 132. \quad (11.5)$$

After applying this convolution, we would set the pixel located at the coordinate (i, j) of the output image O to $O_{i,j} = 132$.

That's all there is to it! Convolution is simply the sum of element-wise matrix multiplication between the kernel and neighborhood that the kernel covers of the input image.

11.1.5 Implementing Convolutions with Python

To help us further understand the concept of convolutions, let's look at some actual code that will reveal how kernels and convolutions are implemented. This source code will not only help you understand *how* to apply convolutions to images, but also enable you to understand *what's going on under the hood* when training CNNs.

Open up a new file, name it `convolutions.py`, and let's get to work:

```

1 # import the necessary packages
2 from skimage.exposure import rescale_intensity
3 import numpy as np
4 import argparse
5 import cv2

```

We start on **Lines 2-5** by importing our required Python packages. We'll be using NumPy and OpenCV for our standard numerical array processing and computer vision functions, along with the scikit-image library to help us implement our own custom convolution function.

Next, we can start defining this `convolve` method:

```

7 def convolve(image, K):
8     # grab the spatial dimensions of the image and kernel
9     (iH, iW) = image.shape[:2]
10    (kH, kW) = K.shape[:2]
11
12    # allocate memory for the output image, taking care to "pad"
13    # the borders of the input image so the spatial size (i.e.,
14    # width and height) are not reduced
15    pad = (kW - 1) // 2
16    image = cv2.copyMakeBorder(image, pad, pad, pad, pad,
17                               cv2.BORDER_REPLICATE)
18    output = np.zeros((iH, iW), dtype="float")

```

The `convolve` function requires two parameters: the (grayscale) image that we want to convolve with `kernel`. Given both our image and `kernel` (which we presume to be NumPy arrays), we then determine the spatial dimensions (i.e., width and height) of each (**Lines 9 and 10**).

Before we continue, it's important to understand the process of “sliding” a convolutional matrix across an image, applying the convolution, and then storing the output, which will actually *decrease* the spatial dimensions of our input image. Why is this?

Recall that we “center” our computation around the center (x,y)-coordinate of the input image that the kernel is currently positioned over. *This positioning implies there is no such thing as “center” pixels for pixels that fall along the border of the image* (as the corners of the kernel would be “hanging off” the image where the values are undefined), depicted by Figure 11.3.

The decrease in spatial dimension is simply a side effect of applying convolutions to images. Sometimes this effect is desirable, and other times it is not, it simply depends on your application.

However, in most cases, we want our *output image* to have the *same dimensions as our input image*. To ensure the dimensions are the same, we apply **padding** (**Lines 15-18**). Here we are simply replicating the pixels along the border of the image, such that the output image will match the dimensions of the input image.

Other padding methods exist, including *zero padding* (filling the borders with zeros – very common when building Convolutional Neural Networks) and *wrap around* (where the border pixels are determined by examining the opposite side of the image). In most cases, you will see either replicate or zero padding. Replicate padding is more commonly used when aesthetics are concerned while zero padding is best for efficiency.

We are now ready to apply the actual convolution to our image:

```

20    # loop over the input image, "sliding" the kernel across
21    # each (x, y)-coordinate from left-to-right and top-to-bottom

```

-1	0	+1				
-2	01	+22	232	84	91	207
-1	04	+13	139	101	237	109
	243	26	252	196	135	126
	185	135	230	48	61	225
	157	124	25	14	102	108
	5	155	116	218	232	249

Figure 11.3: If we attempted to apply convolution at the pixel located at (0,0), then our 3×3 kernel would “hang off” off the edge of the image. Notice how there are no input image pixel values for the first row and first column of the kernel. Because of this, we always either (1) start convolution at the first valid position or (2) apply zero padding (covered later in this chapter).

```

22     for y in np.arange(pad, iH + pad):
23         for x in np.arange(pad, iW + pad):
24             # extract the ROI of the image by extracting the
25             # *center* region of the current (x, y)-coordinates
26             # dimensions
27             roi = image[y - pad:y + pad + 1, x - pad:x + pad + 1]
28
29             # perform the actual convolution by taking the
30             # element-wise multiplication between the ROI and
31             # the kernel, then summing the matrix
32             k = (roi * K).sum()
33
34             # store the convolved value in the output (x, y)-
35             # coordinate of the output image
36             output[y - pad, x - pad] = k

```

Lines 22 and 23 loop over our image, “sliding” the kernel from left-to-right and top-to-bottom, one pixel at a time. **Line 27** extracts the Region of Interest (ROI) from the image using NumPy array slicing. The roi will be centered around the current (x,y)-coordinates of the image. The roi will also have the same size as our kernel, which is critical for the next step.

Convolution is performed on **Line 32** by taking the element-wise multiplication between the roi and kernel, followed by summing the entries in the matrix. The output value k is then stored

in the output array at the same (x,y)-coordinates (relative to the input image).

We can now finish up our convolve method:

```

38         # rescale the output image to be in the range [0, 255]
39         output = rescale_intensity(output, in_range=(0, 255))
40         output = (output * 255).astype("uint8")
41
42     # return the output image
43     return output

```

When working with images, we typically deal with pixel values falling in the range [0,255]. However, when applying convolutions, we can easily obtain values that fall *outside* this range. In order to bring our output image back into the range [0,255], we apply the `rescale_intensity` function of `scikit-image` (**Line 39**).

We also convert our image back to an unsigned 8-bit integer data type on **Line 40** (previously, the output image was a floating point type in order to handle pixel values outside the range [0,255]). Finally, the output image is returned to the calling function on **Line 43**.

Now that we've defined our `convolve` function, let's move on to the driver portion of the script. This section of our program will handle parsing command line arguments, defining a series of kernels we are going to apply to our image, and then displaying the output results:

```

45 # construct the argument parse and parse the arguments
46 ap = argparse.ArgumentParser()
47 ap.add_argument("-i", "--image", required=True,
48                 help="path to the input image")
49 args = vars(ap.parse_args())

```

Our script requires only a single command line argument, `--image`, which is the path to our input image. We can then define two kernels used for blurring and smoothing an image:

```

51 # construct average blurring kernels used to smooth an image
52 smallBlur = np.ones((7, 7), dtype="float") * (1.0 / (7 * 7))
53 largeBlur = np.ones((21, 21), dtype="float") * (1.0 / (21 * 21))

```

To convince yourself that this kernel is performing blurring, notice how each entry in the kernel is an *average* of $1/S$ where S is the total number of entries in the matrix. Thus, this kernel will multiply each input pixel by a small fraction and take the sum – this is exactly the definition of the average.

We then have a kernel responsible for sharpening an image:

```

55 # construct a sharpening filter
56 sharpen = np.array((
57     [0, -1, 0],
58     [-1, 5, -1],
59     [0, -1, 0]), dtype="int")

```

Then the Laplacian kernel used to detect edge-like regions:

```

61 # construct the Laplacian kernel used to detect edge-like
62 # regions of an image
63 laplacian = np.array((
64     [0, 1, 0],
65     [1, -4, 1],
66     [0, 1, 0]), dtype="int")

```

The Sobel kernels can be used to detect edge-like regions along both the x and y axis, respectively:

```

68 # construct the Sobel x-axis kernel
69 sobelX = np.array((
70     [-1, 0, 1],
71     [-2, 0, 2],
72     [-1, 0, 1]), dtype="int")
73
74 # construct the Sobel y-axis kernel
75 sobelY = np.array((
76     [-1, -2, -1],
77     [0, 0, 0],
78     [1, 2, 1]), dtype="int")

```

And finally, we define the emboss kernel:

```

80 # construct an emboss kernel
81 emboss = np.array((
82     [-2, -1, 0],
83     [-1, 1, 1],
84     [0, 1, 2]), dtype="int")

```

Explaining how each of these kernels were formulated is outside the scope of this book, so for the time being simply understand that these are kernels that were *manually built* to perform a given operation.

For a thorough treatment of how kernels are mathematically constructed and proven to perform a given image processing operation, please refer to Szeliksi (Chapter 3) [124]. I also recommend using this excellent kernel visualization tool from Setosa.io [125].

Given all these kernels, we can lump them together into a set of tuples called a “kernel bank”:

```

86 # construct the kernel bank, a list of kernels we're going to apply
87 # using both our custom 'convole' function and OpenCV's 'filter2D'
88 # function
89 kernelBank = (
90     ("small_blur", smallBlur),
91     ("large_blur", largeBlur),
92     ("sharpen", sharpen),
93     ("laplacian", laplacian),
94     ("sobel_x", sobelX),
95     ("sobel_y", sobelY),
96     ("emboss", emboss))

```

Constructing this list of kernels enables use to loop over them and visualize their output in an efficient manner, as the code block below demonstrates:

```

98 # load the input image and convert it to grayscale
99 image = cv2.imread(args["image"])
100 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
101
102 # loop over the kernels
103 for (kernelName, K) in kernelBank:
104     # apply the kernel to the grayscale image using both our custom
105     # 'convolve' function and OpenCV's 'filter2D' function
106     print("[INFO] applying {} kernel".format(kernelName))
107     convolveOutput = convolve(gray, K)
108     opencvOutput = cv2.filter2D(gray, -1, K)
109
110     # show the output images
111     cv2.imshow("Original", gray)
112     cv2.imshow("{} - convolve".format(kernelName), convolveOutput)
113     cv2.imshow("{} - opencv".format(kernelName), opencvOutput)
114     cv2.waitKey(0)
115     cv2.destroyAllWindows()

```

Lines 99 and 100 load our image from disk and convert it to grayscale. Convolution operators can and are applied to RGB or other multi-channel volumes, but for the sake of simplicity, we'll only apply our filters to grayscale images.

We start looping over our set of kernels in the `kernelBank` on **Line 103** and then apply the current kernel to the gray image on **Line 107** by calling our function `convolve` method, defined earlier in the script.

As a sanity check, we also call `cv2.filter2D` which also applies our kernel to the gray image. The `cv2.filter2D` function is OpenCV's much more optimized version of our `convolve` function. The main reason I am including both here is for us to sanity check our custom implementation.

Finally, **Lines 111-115** display the output images to our screen for each kernel type.

Convolution Results

To run our script (and visualize the output of various convolution operations), just issue the following command:

```
$ python convolutions.py --image jemma.png
```

You'll then see the results of applying the `smallBlur` kernel to the input image in Figure 11.4. On the *left*, we have our original image. Then, in the *center*, we have the results from the `convolve` function. And on the *right*, the results from `cv2.filter2D`. A quick visual inspection will reveal that our output matches `cv2.filter2D`, indicating that our `convolve` function is working properly. Furthermore, our image now appears "blurred" and "smoothed", thanks to the smoothing kernel.

Let's apply a larger blur, results of which can be seen in Figure 11.5 (*top-left*). This time I am omitting the `cv2.filter2D` results to save space. Comparing the results from Figure 11.5 to Figure 11.4, notice how as the size of the averaging kernel *increases*, the amount of blurring in the output image *increases* as well.

We can also sharpen our image (Figure 11.5, *top-mid*) and detect edge-like regions via the Laplacian operator (*top-right*).

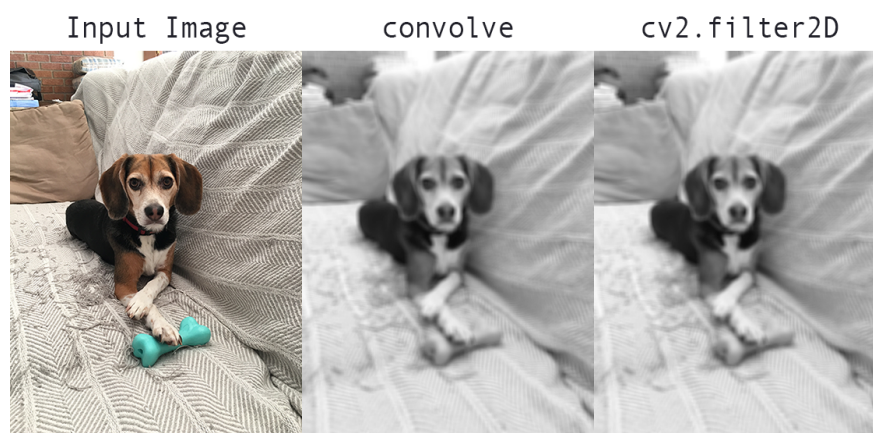


Figure 11.4: **Left:** Our original input image. **Center:** Applying a 7×7 average blur using our custom `convolve` function. **Right:** Applying the same 7×7 blur using OpenCV’s `cv2.filter2D` – notice how the output of the two functions is identical, implying that our `convolve` method is implemented correctly.

The `sobelX` kernel is used to find vertical edges in the image (Figure 11.5, *bottom-left*), while the `sobelY` kernel reveals horizontal edges (*bottom-mid*). Finally, we can see the result of the `emboss` kernel in the *bottom-right*.

11.1.6 The Role of Convolutions in Deep Learning

As you’ve gathered from this section, we must *manually hand-define* each of our kernels for each of our various image processing operations, such as smoothing, sharpening, and edge detection. That’s all fine and good, **but what if there was a way to learn these filters instead?**

Is it possible to define a machine learning algorithm that can look at our input images and eventually *learn* these types of operators? In fact, there is – these types of algorithms are the primary focus of this book: **Convolutional Neural Networks (CNNs)**.

By applying convolutions filters, nonlinear activation functions, pooling, and backpropagation, CNNs are able to learn filters that can detect edges and blob-like structures in lower-level layers of the network – and then use the edges and structures as “building blocks”, eventually detecting high-level objects (e.x., faces, cats, dogs, cups, etc.) in the deeper layers of the network.

This process of using the lower-level layers to learn high-level features is exactly the *compositionality* of CNNs that we were referring to earlier. But exactly *how* do CNNs do this? The answer is by stacking a specific set of layers in a purposeful manner. In our next section, we’ll discuss these types of layers, followed by examining common layer stacking patterns that are widely used among many image classification tasks.

11.2 CNN Building Blocks

As we learned from Chapter 10, neural networks accept an input image/feature vector (one input node for each entry) and transform it through a series of hidden layers, commonly using nonlinear activation functions. Each hidden layer is also made up of a set of neurons, where each neuron is *fully-connected* to all neurons in the previous layer. The last layer of a neural network (i.e., the “output layer”) is also fully-connected and represents the final output classifications of the network.

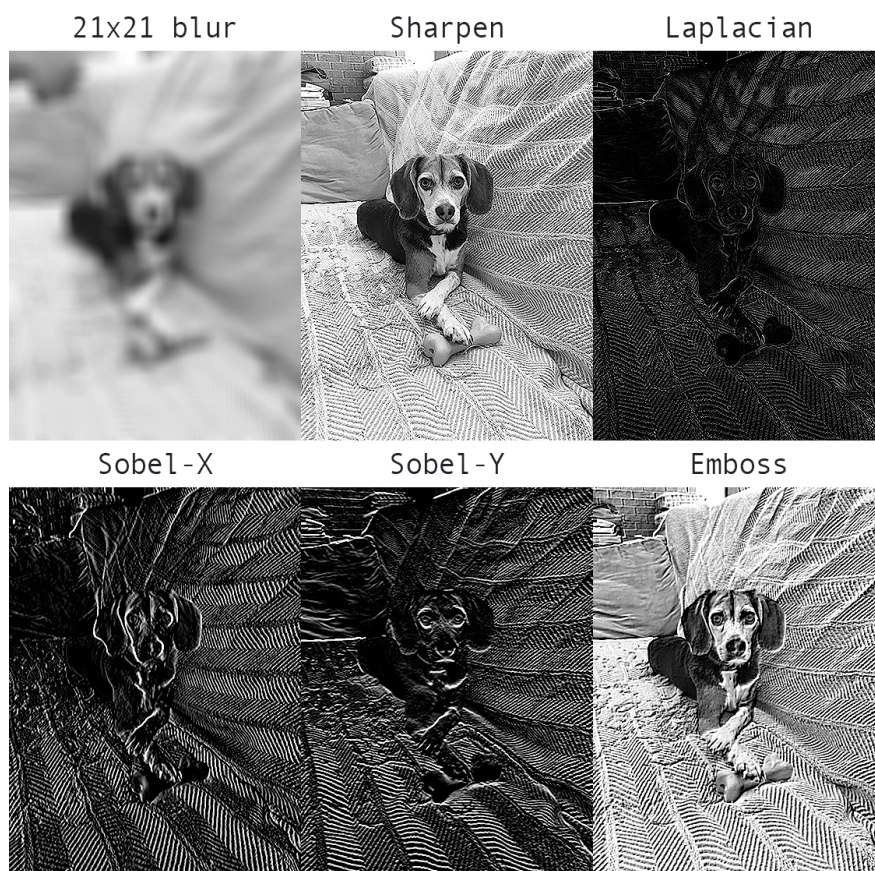


Figure 11.5: **Top-left:** Applying a 21×21 average blur. Notice how this image is more blurred than in Figure 11.4. **Top-mid:** Using a sharpening kernel to enhance details. **Top-right:** Detecting edge-like operations via the Laplacian operator. **Bottom-left:** Computing vertical edges using the Sobel-X kernel. **Bottom-mid:** Finding horizontal edges using the Sobel-Y kernel. **Bottom-right:** Applying an emboss kernel.

However, as the results of Section 10.1.4 demonstrate, neural networks operating directly on raw pixel intensities:

1. Do not scale well as the image size increases.
2. Leaves much accuracy to be desired (i.e., a standard feedforward neural network on CIFAR-10 obtained only 15% accuracy).

To demonstrate how standard neural networks do not scale well as image size increases, let's again consider the CIFAR-10 dataset. Each image in CIFAR-10 is 32×32 with a Red, Green, and Blue channel, yielding a total of $32 \times 32 \times 3 = 3,072$ total inputs to our network.

A total of 3,072 inputs does not seem to amount to much, but consider if we were using 250×250 pixel images – the total number of inputs and weights would jump to $250 \times 250 \times 3 = 187,500$ – and this number is only for the input layer alone! Surely, we would want to add multiple hidden layers with varying number of nodes per layer – these parameters can quickly add up, and given the poor performance of standard neural networks on raw pixel intensities, this bloat is hardly worth it.

Instead, we can use *Convolutional Neural Networks (CNNs)* that take advantage of the input

image structure and define a network architecture in a more sensible way. Unlike a standard neural network, layers of a CNN are arranged in a *3D volume* in three dimensions: **width**, **height**, and **depth** (where *depth* refers to the third dimension of the volume, such as the number of channels in an image or the number of filters in a layer).

To make this example more concrete, again consider the CIFAR-10 dataset: the input volume will have dimensions $32 \times 32 \times 3$ (width, height, and depth, respectively). Neurons in subsequent layers will only be connected to a *small region* of the layer before it (rather than the fully-connected structure of a standard neural network) – we call this *local connectivity* which enables us to save a *huge* amount of parameters in our network. Finally, the output layer will be a $1 \times 1 \times N$ volume which represents the image distilled into a single vector of class scores. In the case of CIFAR-10, given ten classes, $N = 10$, yielding a $1 \times 1 \times 10$ volume.

11.2.1 Layer Types

There are many types of layers used to build Convolutional Neural Networks, but the ones you are most likely to encounter include:


- Convolutional (CONV)
- Activation (ACT or RELU, where we use the same or the actual activation function)
- Pooling (POOL)
- Fully-connected (FC)
- Batch normalization (BN)
- Dropout (DO)

Stacking a series of these layers in a specific manner yields a CNN. We often use simple text diagrams to describe a CNN: INPUT \Rightarrow CONV \Rightarrow RELU \Rightarrow FC \Rightarrow SOFTMAX

Here we define a simple CNN that accepts an input, applies a convolution layer, then an activation layer, then a fully-connected layer, and, finally, a softmax classifier to obtain the output classification probabilities. The SOFTMAX activation layer is often omitted from the network diagram as it is assumed it directly follows the final FC.

Of these layer types, CONV and FC, (and to a lesser extent, BN) are the only layers that contain parameters that are *learned* during the training process. Activation and dropout layers are not considered true “layers” themselves, but are often included in network diagrams to make the architecture *explicitly* clear. Pooling layers (POOL), of equal importance as CONV and FC, are also included in network diagrams as they have a *substantial impact* on the spatial dimensions of an image as it moves through a CNN.

CONV, POOL, RELU, and FC are the most important when defining your actual network architecture. That’s not to say that the other layers are not critical, but take a backseat to this critical set of four as they define the *actual architecture itself*.

 Activation functions themselves are practically **assumed** to be part of the architecture. When defining CNN architectures we often omit the activation layers from a table/diagram to save space; however, the activation layers are *implicitly* assumed to be part of the architecture.

In the remainder of this section, we’ll review each of these layer types in detail and discuss the parameters associated with each layer (and how to set them). Later in this chapter I’ll discuss in more detail how to stack these layers properly to build your own CNN architectures.

11.2.2 Convolutional Layers

The CONV layer is the core building block of a Convolutional Neural Network. The CONV layer parameters consist of a set of K learnable filters (i.e., “kernels”), where each filter has a width and a height, and are nearly always square. These filters are small (in terms of their spatial dimensions) but extend throughout the full depth of the volume.

For inputs to the CNN, the depth is the number of channels in the image (i.e., a depth of three when working with RGB images, one for each channel). For volumes deeper in the network, the depth will be the number of filters applied in the *previous* layer.

To make this concept more clear, let's consider the forward-pass of a CNN, where we convolve each of the K filters across the width and height of the input volume, just like we did in Section 11.1.5 above. More simply, we can think of each of our K kernels sliding across the input region, computing an element-wise multiplication, summing, and then storing the output value in a 2-dimensional *activation map*, such as in Figure 11.6.

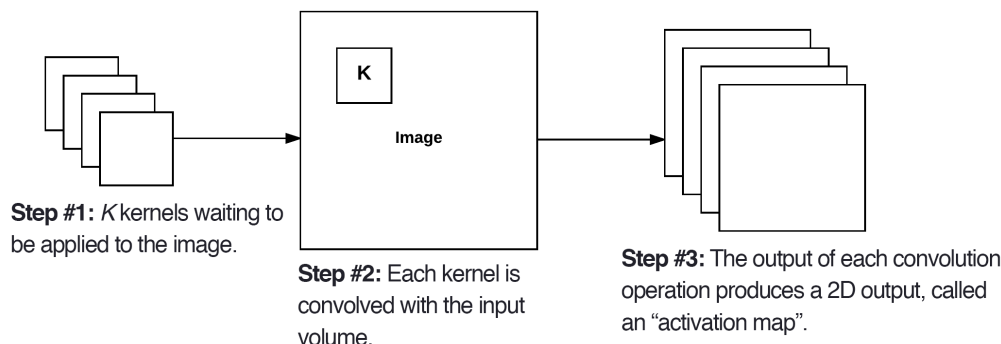


Figure 11.6: **Left:** At each convolutional layer in a CNN, there are K kernels applied to the input volume. **Middle:** Each of the K kernels is convolved with the input volume. **Right:** Each kernel produces an 2D output, called an *activation map*.

After applying all K filters to the input volume, we now have K , 2-dimensional activation maps. We then stack our K activation maps along the depth dimension of our array to form the final output volume (Figure 11.7).

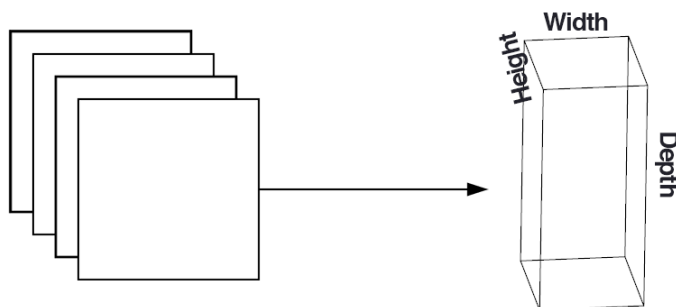


Figure 11.7: After obtaining the K activation maps, they are stacked together to form the input volume to the next layer in the network.

Every entry in the output volume is thus an output of a neuron that “looks” at only a small region of the input. In this manner, the network “learns” filters that activate when they see a specific type of feature at a *given spatial location* in the input volume. In lower layers of the network, filters may activate when they see edge-like or corner-like regions.

Then, in the deeper layers of the network, filters may activate in the presence of high-level features, such as parts of the face, the paw of a dog, the hood of a car, etc. This activation concept

goes back to our neural network analogy in Chapter 10 – these neurons are becoming “excited” and “activating” when they see a particular pattern in an input image.

The concept of convolving a small filter with a large(r) input volume has special meaning in Convolutional Neural Networks – specifically, the **local connectivity** and the **receptive field** of a neuron. When working with images, it’s often impractical to connect neurons in the current volume to *all* neurons in the previous volume – there are simply too many connections and too many weights, making it impossible to train deep networks on images with large spatial dimensions. Instead, when utilizing CNNs, we choose to connect each neuron to only a *local region* of the input volume – we call the size of this local region the **receptive field** (or simply, the variable F) of the neuron.

To make this point clear, let’s return to our CIFAR-10 dataset where the input volume has an input size of $32 \times 32 \times 3$. Each image thus has a width of 32 pixels, a height of 32 pixels, and a depth of 3 (one for each RGB channel). If our receptive field is of size 3×3 , then each neuron in the CONV layer will connect to a 3×3 local region of the image for a total of $3 \times 3 \times 3 = 27$ weights (remember, the depth of the filters is three because they extend through the full depth of the input image, in this case, three channels).

Now, let’s assume that the spatial dimensions of our input volume have been reduced to a smaller size, but our depth is now larger, due to utilizing more filters deeper in the network, such that the volume size is now $16 \times 16 \times 94$. Again, if we assume a receptive field of size 3×3 , then every neuron in the CONV layer will have a total of $3 \times 3 \times 94 = 846$ connections to the input volume. Simply put, the receptive field F is the *size* of the filter, yielding an $F \times F$ kernel that is convolved with the input volume.

At this point we have explained the connectivity of neurons in the input volume, but not the arrangement or size of the output volume. There are three parameters that control the size of an output volume: the **depth**, **stride**, and **zero-padding** size, each of which we’ll review below.

Depth

The *depth* of an output volume controls the number of neurons (i.e., filters) in the CONV layer that connect to a local region of the input volume. Each filter produces an activation map that “activates” in the presence of oriented edges or blobs or color.

For a given CONV layer, the depth of the activation map will be K , or simply the number of filters we are learning in the current layer. The set of filters that are “looking at” the same (x, y) location of the input is called the **depth column**.

Stride

Consider Figure 11.1 earlier in this chapter where we described a convolution operation as “sliding” a small matrix across a large matrix, stopping at each coordinate, computing an element-wise multiplication and sum, then storing the output. This description is similar to a *sliding window* (<http://pyimg.co/0yizo>) that slides from left-to-right and top-to-bottom across an image.

In the context of Section 11.1.5 on convolution above, we only took a step of one pixel each time. In the context of CNNs, the same principle can be applied – for each step, we create a new depth column around the local region of the image where we convolve each of the K filters with the region and store the output in a 3D volume. When creating our CONV layers we normally use a stride step size S of either $S = 1$ or $S = 2$.

Smaller strides will lead to overlapping receptive fields and larger output volumes. Conversely, larger strides will result in less overlapping receptive fields and smaller output volumes. To make the concept of convolutional stride more concrete, consider the Table 11.1 where we have a 5×5 input image (*left*) along with a 3×3 Laplacian kernel (*right*).

Using $S = 1$, our kernel slides from left-to-right and top-to-bottom, one pixel at a time, producing the following output (Figure 11.2, *left*). However, if we were to apply the same operation, only

95	242	186	152	39			
39	14	220	153	180	0	1	0
5	247	212	54	46	1	-4	1
46	77	133	110	74	0	1	0
156	35	74	93	116			

Table 11.1: Our input 5×5 image (*left*) that we are going to convolve with a Laplacian kernel (*right*).

692	-315	-6	692	-6
-680	-194	305	153	-86
153	-59	-86		

Table 11.2: **Left:** Output of convolution with 1×1 stride. **Right:** Output of convolution with 2×2 stride. Notice how a larger stride can reduce the spatial dimensions of the input.

this time with a stride of $S = 2$, we skip *two pixels at a time* (two pixels along the x -axis and two pixels along the y -axis), producing a smaller output volume (*right*).

Thus, we can see how convolution layers can be used to reduce the spatial dimensions of the input volumes simply by changing the stride of the kernel. As we'll see later in this section, convolutional layers and pooling layers are the primary methods to reduce spatial input size. The pooling layers section will also provide a more visual example of how vary stride sizes will affect output size.

Zero-padding

As we know from Section 11.1.5, we need to “pad” the borders of an image to retain the *original image size* when applying a convolution – the same is true for filters inside of a CNN. Using zero-padding, we can “pad” our input along the borders such that our output volume size matches our input volume size. The amount of padding we apply is controlled by the parameter P .

This technique is *especially critical* when we start looking at deep CNN architectures that apply multiple CONV filters on top of each other. To visualize zero-padding, again refer to Table 11.1 where we applied a 3×3 Laplacian kernel to a 5×5 input image with a stride of $S = 1$.

We can see in Table 11.3 (*left*) how the output volume is *smaller* (3×3) than the input volume (5×5) due to the nature of the convolution operation. If we instead set $P = 1$, we can pad our input volume with zeros (*right*) to create a 7×7 volume and then apply the convolution operation, leading to an output volume size that matches the original input volume size of 5×5 (*bottom*).

Without zero padding, the spatial dimensions of the input volume would decrease too quickly, and we wouldn't be able to train deep networks (as the input volumes would be too tiny to learn any useful patterns from).

Putting all these parameters together, we can compute the size of an output volume as a function of the input volume size (W , assuming the input images are square, which they nearly always are), the receptive field size F , the stride S , and the amount of zero-padding P . To construct a valid CONV layer, we need to ensure the following equation is an integer:

$$((W - F + 2P)/S) + 1 \quad (11.6)$$

If it is *not* an integer, then the strides are set incorrectly, and the neurons cannot be tiled such that they fit across the input volume in a symmetric way.

692	-315	-6	0	0	0	0	0	0
-680	-194	305	0	95	242	186	152	39
153	-59	-86	0	39	14	220	153	180
			0	5	247	212	54	46
			0	46	77	133	110	74
			0	156	35	74	93	116
			0	0	0	0	0	0

-99	-673	-130	-230	176
-42	692	-315	-6	-482
312	-680	-194	305	124
54	153	-59	-86	-24
-543	167	-35	-72	-297

Table 11.3: **Left:** The output of applying a 3×3 convolution to a 5×5 output (i.e., the spatial dimensions decrease). **Right:** Applying zero-padding to the original input with $P = 1$ increases the spatial dimensions to 7×7 . **Bottom:** After applying the 3×3 convolution to the padded input, our output volume times matches the *original* input volume size of 5×5 , thus zero-padding helps us preserve spatial dimensions.

As an example, consider the first layer of the AlexNet architecture which won the 2012 ImageNet classification challenge and is *hugely* responsible for the current boom of deep learning applied to image classification. Inside their paper, Krizhevsky et al. [99] documented their CNN architecture according to Figure 11.8.

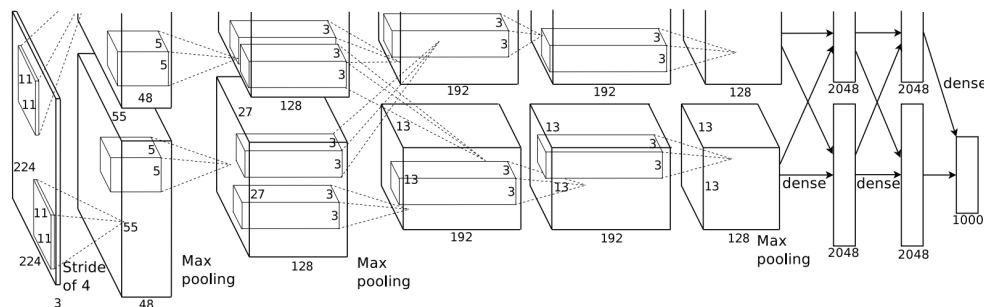


Figure 11.8: The original AlexNet architecture diagram provided by Krizhevsky et al. [99] Notice how the input image is documented to be $224 \times 224 \times 3$, although this cannot be possible due to Equation 11.6. It’s also worth noting that we are unsure why the top-half of this figure is cutoff from the original publication.

Notice how the first layer claims that the input image size is 224×224 pixels. However, this can't possibly be correct if we apply our equation above using 11×11 filters, a stride of four, and no padding:

$$((224 - 11 + 2(0))/4) + 1 = 54.25 \quad (11.7)$$

Which is certainly not an integer.

For novice readers just getting started in deep learning and CNNs, this small error in such a seminal paper has caused countless errors of confusion and frustration. It's unknown why this typo occurred, but it's likely that Krizhevsky et al. used 227×227 input images, since:

$$((227 - 11 + 2(0))/4) + 1 = 55 \quad (11.8)$$

Errors like these are more common than you might think, so when implementing CNNs from publications, be sure to *check the parameters yourself* rather than simply assuming the parameters listed are correct. Due to the vast number of parameters in a CNN, it's quite easy to make a typographical mistake when documenting an architecture (I've done it myself many times).

To summarize, the CONV layer in the same, elegant manner as Karpathy [126]:

- Accepts an input volume of size $W_{input} \times H_{input} \times D_{input}$ (the input sizes are normally square, so it's common to see $W_{input} = H_{input}$).
- Requires four parameters:
 1. The number of filters K (which controls the *depth* of the output volume).
 2. The receptive field size F (the size of the K kernels used for convolution and is nearly always *square*, yielding an $F \times F$ kernel).
 3. The stride S .
 4. The amount of zero-padding P .
- The output of the CONV layer is then $W_{output} \times H_{output} \times D_{output}$, where:
 - $W_{output} = ((W_{input} - F + 2P)/S) + 1$
 - $H_{output} = ((H_{input} - F + 2P)/S) + 1$
 - $D_{output} = K$

We'll review common settings for these parameters in Section 11.3.1 below.

11.2.3 Activation Layers

After each CONV layer in a CNN, we apply a nonlinear activation function, such as ReLU, ELU, or any of the other Leaky ReLU variants mentioned in Chapter 10. We typically denote activation layers as RELU in network diagrams as since ReLU activations are most commonly used, we may also simply state ACT – in either case, we are making it clear that an activation function is being applied inside the network architecture.

Activation layers are not technically “layers” (due to the fact that no parameters/weights are learned inside an activation layer) and are sometimes omitted from network architecture diagrams as it's *assumed* that an activation *immediately follows* a convolution.

In this case, authors of publications will mention which activation function they are using after each CONV layer somewhere in their paper. As an example, consider the following network architecture: INPUT => CONV => RELU => FC.

To make this diagram more concise, we could simply remove the RELU component since it's assumed that an activation always follows a convolution: INPUT => CONV => FC. I personally do not like this and choose to *explicitly* include the activation layer in a network diagram to make it clear *when* and *what* activation function I am applying in the network.

An activation layer accepts an input volume of size $W_{input} \times H_{input} \times D_{input}$ and then applies the given activation function (Figure 11.9). Since the activation function is applied in an element-wise manner, the output of an activation layer is always the same as the input dimension, $W_{input} = W_{output}$, $H_{input} = H_{output}$, $D_{input} = D_{output}$.

11.2.4 Pooling Layers

There are two methods to reduce the size of an input volume – CONV layers with a stride > 1 (which we've already seen) and POOL layers. It is common to insert POOL layers in-between consecutive

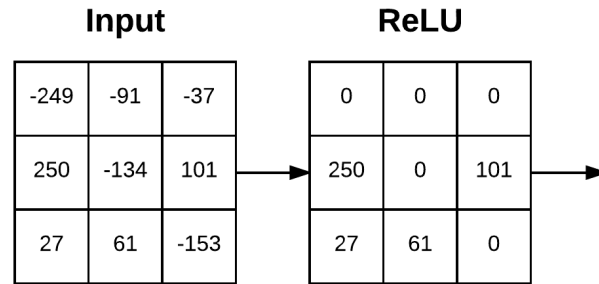


Figure 11.9: An example of an input volume going through a ReLU activation, $\max(0, x)$. Activations are done *in-place* so there is no need to create a separate output volume although it is easy to visualize the flow of the network in this manner.

CONV layers in a CNN architectures:

INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC

The primary function of the POOL layer is to progressively reduce the spatial size (i.e., width and height) of the input volume. Doing this allows us to reduce the amount of parameters and computation in the network – pooling also helps us control overfitting.

POOL layers operate on each of the depth slices of an input *independently* using either the *max* or *average* function. Max pooling is typically done in the middle of the CNN architecture to reduce spatial size, whereas average pooling is normally used as the final layer of the network (e.x., GoogLeNet, SqueezeNet, ResNet) where we wish to avoid using FC layers entirely. The most common type of POOL layer is max pooling, although this trend is changing with the introduction of more exotic micro-architectures.

Typically we'll use a pool size of 2×2 , although deeper CNNs that use larger input images (> 200 pixels) may use a 3×3 pool size early in the network architecture. We also commonly set the stride to either $S = 1$ or $S = 2$. Figure 11.10 (heavily inspired by Karpathy et al. [126]) follows an example of applying max pooling with 2×2 pool size and a stride of $S = 1$. Notice for every 2×2 block, we keep only the largest value, take a single step (like a sliding window), and apply the operation again – thus producing an output volume size of 3×3 .

We can further decrease the size of our output volume by increasing the stride – here we apply $S = 2$ to the same input (Figure 11.10, *bottom*). For every 2×2 block in the input, we keep only the largest value, then take a step of *two pixels*, and apply the operation again. This pooling allows us to reduce the width and height by a factor of two, effectively discarding 75% of activations from the previous layer.

In summary, POOL layers Accept an input volume of size $W_{input} \times H_{input} \times D_{input}$. They then require two parameters:

- The receptive field size F (also called the “pool size”).
- The stride S .

Applying the POOL operation yields an output volume of size $W_{output} \times H_{output} \times D_{output}$, where:

- $W_{output} = ((W_{input} - F)/S) + 1$
- $H_{output} = ((H_{input} - F)/S) + 1$
- $D_{output} = D_{input}$

In practice, we tend to see two types of max pooling variations:

- **Type #1:** $F = 3, S = 2$ which is called *overlapping pooling* and normally applied to images/input volumes with large spatial dimensions.

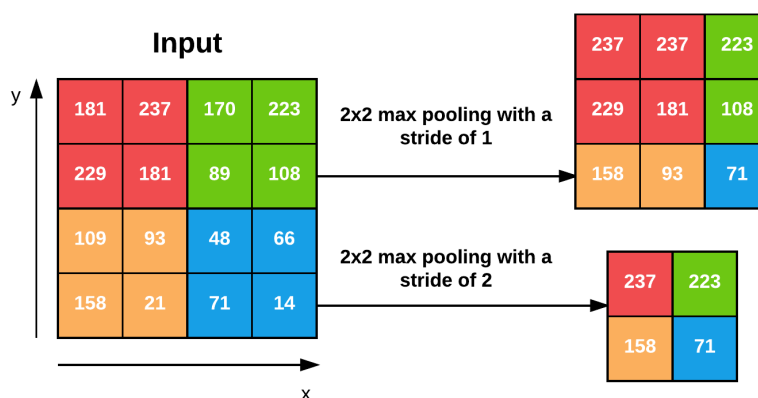


Figure 11.10: **Left:** Our input 4×4 volume. **Right:** Applying 2×2 max pooling with a stride of $S = 1$. **Bottom:** Applying 2×2 max pooling with $S = 2$ – this dramatically reduces the spatial dimensions of our input.

- **Type #2:** $F = 2, S = 2$ which is called *non-overlapping pooling*. This is the most common type of pooling and is applied to images with smaller spatial dimensions.

For network architectures that accept smaller input images (in the range of 32 – 64 pixels) you may also see $F = 2, S = 1$ as well.

To POOL or CONV?

In their 2014 paper, *Striving for Simplicity: The All Convolutional Net*, Springenberg et al. [127] recommend discarding the POOL layer *entirely* and simply relying on CONV layers with a larger stride to handle downsampling the spatial dimensions of the volume. Their work demonstrated this approach works very well on a variety of datasets, including CIFAR-10 (small images, low number of classes) and ImageNet (large input images, 1,000 classes). This trend continues with the ResNet architecture [101] which uses CONV layers for downsampling as well.

It's becoming increasingly more common to *not* use POOL layers in the middle of the network architecture and *only* use average pooling at the end of the network if FC layers are to be avoided. Perhaps in the future there won't be pooling layers in Convolutional Neural Networks – but in the meantime, it's important that we study them, learn how they work, and apply them to our own architectures.

11.2.5 Fully-connected Layers

Neurons in FC layers are fully-connected to all activations in the previous layer, as is the standard for feedforward neural networks that we've been discussing in Chapter 10. FC layers are *always* placed at the end of the network (i.e., we don't apply a CONV layer, then an FC layer, followed by another CONV) layer.

It's common to use one or two FC layers prior to applying the softmax classifier, as the following (simplified) architecture demonstrates:

INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC => FC

Here we apply two fully-connected layers before our (implied) softmax classifier which will compute our final output probabilities for each class.

11.2.6 Batch Normalization

First introduced by Ioffe and Szegedy in their 2015 paper, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* [128], batch normalization layers (or BN for short), as the name suggests, are used to normalize the activations of a given input volume before passing it into the next layer in the network.

If we consider x to be our mini-batch of activations, then we can compute the normalized \hat{x} via the following equation:

$$\hat{x}_i = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \varepsilon}} \quad (11.9)$$

During *training*, we compute the μ_β and σ_β over each mini-batch β , where:

$$\mu_\beta = \frac{1}{M} \sum_{i=1}^m x_i \quad \sigma_\beta^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_\beta)^2 \quad (11.10)$$

We set ε equal to a small positive value such as $1e-7$ to avoid taking the square root of zero. Applying this equation implies that the activations leaving a batch normalization layer will have approximately zero mean and unit variance (i.e., zero-centered).

At *testing* time, we replace the mini-batch μ_β and σ_β with *running averages* of μ_β and σ_β computed during the training process. This ensures that we can pass images through our network and still obtain accurate predictions without being biased by the μ_β and σ_β from the final mini-batch passed through the network at training time.

Batch normalization has been shown to be *extremely effective* at reducing the number of epochs it takes to train a neural network. Batch normalization also has the added benefit of helping “stabilize” training, allowing for a larger variety of learning rates and regularization strengths. Using batch normalization doesn’t alleviate the need to tune these parameters of course, but it *will* make your life easier by making learning rate and regularization less volatile and more straightforward to tune. You’ll also tend to notice *lower final loss* and a *more stable loss curve* when using batch normalization in your networks.

The biggest drawback of batch normalization is that it can actually slow down the wall time it takes to train your network (even though you’ll need fewer epochs to obtain reasonable accuracy) by 2-3x due to the computation of per-batch statistics and normalization.

That said, I recommend using batch normalization in *nearly every situation* as it does make a significant difference. As we’ll see later in this book, applying batch normalization to our network architectures can help us prevent overfitting and allows us to obtain significantly higher classification accuracy in fewer epochs compared to the same network architecture *without* batch normalization.

So, Where Do the Batch Normalization Layers Go?

You’ve probably noticed in my discussion of batch normalization I’ve left out exactly *where* in the network architecture we place the batch normalization layer. According to the original paper by Ioffe and Szegedy [128], they placed their batch normalization (BN) *before* the activation:

"We add the BN transform immediately before the nonlinearity, by normalizing
 $x = Wu + b.$ "

Using this scheme, a network architecture utilizing batch normalization would look like this:

```
INPUT => CONV => BN => RELU ...
```

However, this view of batch normalization doesn't make sense from a statistical point of view. In this context, a BN layer is normalizing the distribution of features coming out of a CONV layer. Some of these features may be negative, in which they will be clamped (i.e., set to zero) by a nonlinear activation function such as ReLU.

If we normalize *before* activation, we are essentially including the negative values inside the normalization. Our zero-centered features are then passed through the ReLU where we kill of any activations less than zero (which include features which may have not been negative *before* the normalization) – this layer ordering entirely defeats the purpose of applying batch normalization in the first place.

Instead, if we place the batch normalization *after* the ReLU we will normalize the positive valued features without statistically biasing them with features that would have otherwise not made it to the next CONV layer. In fact, François Chollet, the creator and maintainer of Keras confirms this point stating that the BN should come after the activation:

"I can guarantee that recent code written by Christian [Szegedy, from the BN paper] applies relu before BN. It is still occasionally a topic of debate, though." [129]

It is unclear why Ioffe and Szegedy suggested placing the BN layer before the activation in their paper, but further experiments [130] as well as anecdotal evidence from other deep learning researchers [131] confirm that placing the batch normalization layer *after* the nonlinear activation yields higher accuracy and lower loss in nearly all situations.

Placing the BN after the activation in a network architecture would look like this:

```
INPUT => CONV => RELU => BN ...
```

I can confirm that in nearly all experiments I've performed with CNNs, placing the BN after the RELU yields slightly higher accuracy and lower loss. That said, take note of the word "*nearly*" – there have been a *very small* number of situations where placing the BN before the activation worked better, which implies that you should default to placing the BN after the activation, but may want to dedicate (at most) one experiment to placing the BN before the activation and noting the results.

After running a few of these experiments, you'll quickly realize that BN after the activation performs better and there are more important parameters to your network to tune to obtain higher classification accuracy. I discuss this in more detail in Section 11.3.2 later in this chapter.

11.2.7 Dropout

The last layer type we are going to discuss is dropout. Dropout is actually a form of *regularization* that aims to help prevent overfitting by increasing testing accuracy, perhaps at the expense of training accuracy. For each mini-batch in our training set, dropout layers, with probability p , randomly disconnect inputs from the preceding layer to the next layer in the network architecture.

Figure 11.11 visualizes this concept where we randomly disconnect with probability $p = 0.5$ the connections between two FC layers for a given mini-batch. Again, notice how half of the connections are severed for this mini-batch. After the forward and backward pass are computed for the mini-batch, we re-connect the dropped connections, and then sample another set of connections to drop.

The reason we apply dropout is to reduce overfitting by *explicitly* altering the network architecture at training time. Randomly dropping connections ensures that no single node in the network is

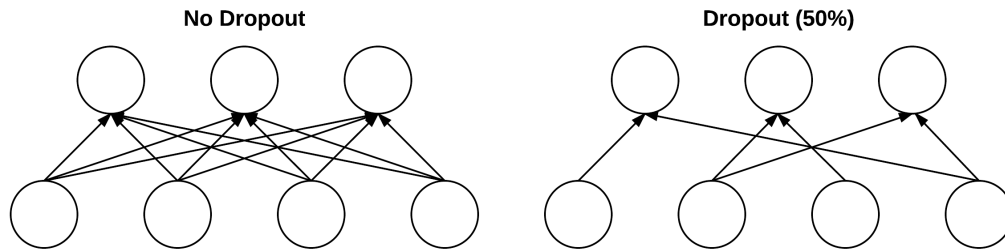


Figure 11.11: **Left:** Two layers of a neural network that are fully-connected with no dropout. **Right:** The same two layers after dropping 50% of the connections.

responsible for “activating” when presented with a given pattern. Instead, dropout ensures there are *multiple, redundant nodes* that will activate when presented with similar inputs – this in turn helps our model to *generalize*.

It is most common to place dropout layers with $p = 0.5$ *in-between* FC layers of an architecture where the final FC layer is assumed to be our softmax classifier:

... CONV => RELU => POOL => FC => DO => FC => DO => FC

However, as I discuss in Section 11.3.2, we may also apply dropout with smaller probabilities (i.e., $p = 0.10 - 0.25$) in earlier layers of the network as well (normally following a downsampling operation, either via max pooling or convolution).

11.3 Common Architectures and Training Patterns

As we have seen throughout this chapter, Convolutional Neural Networks are made up of four primary layers: CONV, POOL, RELU, and FC. Taking these layers and stacking them together in a particular pattern yields a *CNN architecture*.

The CONV and FC layers (and BN) are the only layers of the network that actually learn parameters – the other layers are simply responsible for performing a given operation. Activation layers, (ACT) such as RELU and dropout aren’t technically layers, but are often included in the CNN architecture diagrams to make the operation order *explicitly clear* – we’ll adopt the same convention in this section as well.

11.3.1 Layer Patterns

By far, the most common form of CNN architecture is to stack a few CONV and RELU layers, following them with a POOL operation. We repeat this sequence until the volume width and height is small, at which point we apply one or more FC layers. Therefore, we can derive the most common CNN architecture using the following pattern [126]:

INPUT => [[CONV => RELU]*N => POOL?]*M => [FC => RELU]*K => FC

Here the * operator implies one or more and the ? indicates an optional operation.

Common choices for each repetition include:

- $0 \leq N \leq 3$
- $M \geq 0$
- $0 \leq K \leq 2$

Below we can see some examples of CNN architectures that follow this pattern:

- INPUT => FC
- INPUT => [CONV => RELU => POOL] * 2 => FC => RELU => FC
- INPUT => [CONV => RELU => CONV => RELU => POOL] * 3 => [FC => RELU] * 2 => FC

Here is an example of a very shallow CNN with only one CONV layer ($N = M = K = 0$) which we will review in Chapter 12:

```
INPUT => CONV => RELU => FC
```

Below is an example of an AlexNet-like [99] CNN architecture which has multiple CONV => RELU => POOL layer sets, followed by FC layers:

```
INPUT => [CONV => RELU => POOL] * 2 => [CONV => RELU] * 3 => POOL =>
[FC => RELU => DO] * 2 => SOFTMAX
```

For deeper network architectures, such as VGGNet [100], we'll stack two (or more) layers before every POOL layer:

```
INPUT => [CONV => RELU] * 2 => POOL => [CONV => RELU] * 2 => POOL =>
[CONV => RELU] * 3 => POOL => [CONV => RELU] * 3 => POOL =>
[FC => RELU => DO] * 2 => SOFTMAX
```

Generally, we apply deeper network architectures when we (1) have lots of labeled training data and (2) the classification problem is sufficiently challenging. Stacking multiple CONV layers before applying a POOL layer allows the CONV layers to develop more complex features before the destructive pooling operation is performed.

As we'll discover in the *ImageNet Bundle* of this book, there are more “exotic” network architectures that deviate from these patterns and, in turn, have created patterns of their own. Some architectures remove the POOL operation entirely, relying on CONV layers to downsample the volume – then, at the end of the network, average pooling is applied rather than FC layers to obtain the input to the softmax classifiers.

Network architectures such as GoogLeNet, ResNet, and SqueezeNet [101, 102, 132] are great examples of this pattern and demonstrate how removing FC layers leads to less parameters and faster training time.

These types of network architectures also “stack” and concatenate filters across the channel dimension: GoogLeNet applies 1×1 , 3×3 , and 5×5 filters and then concatenates them together across the channel dimension to learn multi-level features. Again, these architectures are considered more “exotic” and considered advanced techniques.

If you're interested in these more advanced CNN architectures, please refer to the *ImageNet Bundle*; otherwise, you'll want to stick with the basic layer stacking patterns until you learn the fundamentals of deep learning.

11.3.2 Rules of Thumb

In this section, I'll review common rules of thumb when constructing your own CNNs. To start, the images presented to the **input layer** should be *square*. Using square inputs allows us to take advantage of linear algebra optimization libraries. Common input layer sizes include 32×32 ,

64×64 , 96×96 , 224×224 , 227×227 and 229×229 (leaving out the number of channels for notational convenience).

Secondly, the input layer should also be *divisible by two multiple times* after the first CONV operation is applied. You can do this by tweaking your filter size and stride. The “divisible by two rule” enables the spatial inputs in our network to be conveniently down sampled via POOL operation in an efficient manner.

In general, your CONV layers should use smaller filter sizes such as 3×3 and 5×5 . Tiny 1×1 filters are used to learn local features, but only in your more advanced network architectures. Larger filter sizes such as 7×7 and 11×11 may be used as the *first* CONV layer in the network (to reduce spatial input size, provided your images are sufficiently larger than $> 200 \times 200$ pixels); however, after this initial CONV layer the filter size should drop dramatically, otherwise you will reduce the spatial dimensions of your volume too quickly.

You’ll also commonly use a stride of $S = 1$ for CONV layers, at least for smaller spatial input volumes (networks that accept larger input volumes use a stride $S \geq 2$ in the first CONV layer to help reduce spatial dimensions). Using a stride of $S = 1$ enables our CONV layers to learn filters while the POOL layer is responsible for downsampling. However, keep in mind that not *all* network architectures follow this pattern – some architectures skip max pooling altogether and rely on the CONV stride to reduce volume size.

My personal preference is to apply **zero-padding** to my CONV layers to ensure the output dimension size matches the input dimension size – the only exception to this rule is if I want to *purposely* reduce spatial dimensions via convolution. Applying zero-padding when *stacking* multiple CONV layers on top of each other has also demonstrated to increase classification accuracy in practice. As we’ll see later in this book, libraries such as Keras can automatically compute zero-padding for you, making it even easier to build CNN architectures.

A second personal recommendation is to use POOL layers (rather than CONV layers) reduce the spatial dimensions of your input, at least until you become more experienced constructing your own CNN architectures. Once you reach that point, you should start experimenting with using CONV layers to reduce spatial input size and try removing max pooling layers from your architecture.

Most commonly, you’ll see max pooling applied over a 2×2 receptive field size and a stride of $S = 2$. You might also see a 3×3 receptive field early in the network architecture to help reduce image size. It is **highly uncommon** to see receptive fields larger than three since these operations are very destructive to their inputs.

Batch normalization is an expensive operation which can *double* or *triple* the amount of time it takes to train your CNN; however, I recommend using BN in *nearly all situations*. While BN does indeed slow down the training time, it also tends to “stabilize” training, making it easier to tune other hyperparameters (there are some exceptions, of course – I detail a few of these “exception architectures” inside the *ImageNet Bundle*).

I also place the batch normalization *after* the activation, as has become commonplace in the deep learning community even though it goes against the original Ioffe and Szegedy paper [128].

Inserting BN into the common layer architectures above, they become:

- INPUT => CONV => RELU => BN => FC
- INPUT => [CONV => RELU => BN => POOL] * 2 => FC => RELU => BN => FC
- INPUT => [CONV => RELU => BN => CONV => RELU => BN => POOL] * 3 => [FC => RELU => BN] * 2 => FC

You *do not* apply batch normalization before the softmax classifier as at this point we assume our network has learned its discriminative features earlier in the architecture.

Dropout (DO) is typically applied in between FC layers with a dropout probability of 50% – you should consider applying dropout in nearly *every* architecture you build. While not always performed, I also like to include dropout layers (with a very small probability, 10-25%) between

POOL and CONV layers. Due to the local connectivity of CONV layers, dropout is less effective here, but I’ve often found it helpful when battling overfitting.

By keeping these rules of thumb in mind, you’ll be able to reduce your headaches when constructing CNN architectures since your CONV layers will preserve input sizes while the POOL layers take care of reducing spatial dimensions of the volumes, eventually leading to FC layers and the final output classifications.

Once you master this “traditional” method of building Convolutional Neural Networks, you should then start exploring leaving max pooling operations out *entirely* and using *just* CONV layers to reduce spatial dimensions, eventually leading to *average pooling* rather than an FC layer – these types of more advanced architecture techniques are covered inside the *ImageNet Bundle*.

11.4 Are CNNs Invariant to Translation, Rotation, and Scaling?

A common question I get asked is:

“Are Convolutional Neural Networks invariant to changes in translation, rotation, and scaling? Is that why they are such powerful image classifiers?”

To answer this question, we first need to discriminate between the *individual filters* in the network along with the *final trained network*. Individual filters in a CNN are *not* invariant to changes in how an image is rotated – we demonstrate this in Chapter 12 of the *ImageNet Bundle* where we use features extracted from a CNN to determine how an image is oriented.

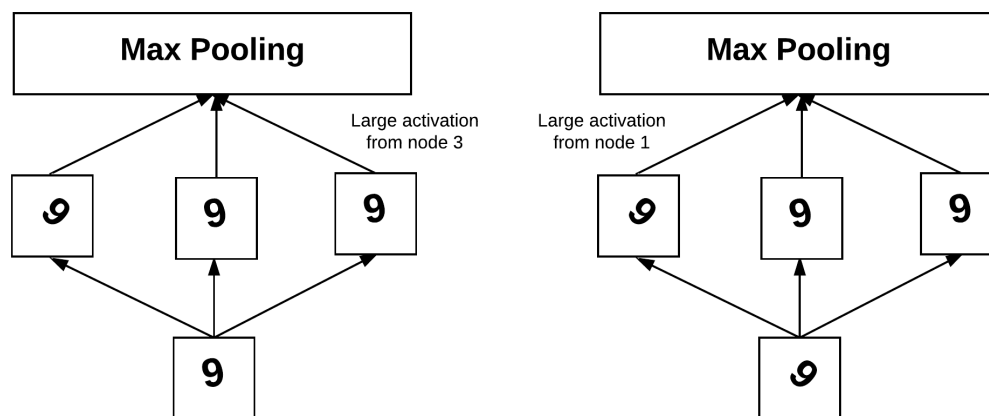


Figure 11.12: CNN as a whole learns filters that will fire when a pattern is presented at a particular orientation. On the *left* the, the digit 9 has been rotated $\approx 10^\circ$. This rotation is similar to node three which has learned what the digit 9 looks like when rotated in this manner. This node will have a higher activation than the other two nodes – the max pooling operation will detect this. On the *right* we have a second example, only this time the 9 has been rotated $\approx -45^\circ$, causing the first node to have the highest activation (Figure heavily inspired by Goodfellow et al. [10]).

However, a CNN *as a whole* can *learn* filters that fire when a pattern is presented at a particular orientation. For example, consider Figure 11.12, adapted and inspired from *Deep Learning* by Goodfellow et al. [10].

Here we see the digit “9” (bottom) presented to the CNN along with a set of filters the CNN has learned (middle). Since there is a filter inside the CNN that has “learned” what a “9” looks

like, rotated by 10 degrees, it fires and emits a strong activation. This large activation is captured during the pooling stage and ultimately reported as the final classification.

The same is true for the second example (Figure 11.12, *left*). Here we see the “9” rotated by -45 degrees, and since there is a filter in the CNN that has learned what a “9” looks like when it is rotated by -45 degrees, the neuron activates and fires. Again, these filters themselves are *not* rotation invariant – it’s just that the CNN has learned what a “9” looks like under *small rotations* that exist in the training set.

Unless your training data includes digits that are rotated across the full 360-degree spectrum, your CNN is *not* truly rotation invariant (again, this point is demonstrated in Chapter 12 of the *ImageNet Bundle*).

The same can be said about scaling – the filters themselves are *not* scale invariant, but it is highly likely that your CNN has learned a set of filters that *fire when patterns exist at varying scales*. We can also “help” our CNNs to be scale invariant by presenting our example image to them at testing time under varying scales and crops, then averaging the results together (see Chapter 10 of the *Practitioner Bundle* for more details on crop averaging to increase classification accuracy).

Translation invariance; however, is something that a CNN excels at. Keep in mind that a filter slides from left-to-right and top-to-bottom across an input, and will activate when it comes across a particular edge-like region, corner, or color blob. During the pooling operation, this large response is found and thus “beats” all its neighbors by having a larger activation. Therefore, CNNs can be seen as “not caring” exactly where an activation fires, simply that it *does fire* – and, in this way, we naturally handle translation inside a CNN.

11.5 Summary

In this chapter we took a tour of Convolutional Neural Network concepts (CNNs). We started by discussing what *convolution* and *cross-correlation* are and how the terms are used interchangeably in the deep learning literature.

To understand convolution at a more intimate level, we implemented it by hand using Python and OpenCV. However, traditional image processing operations require us to hand-define our kernels and are *specific* to a given image processing task (e.x., smoothing, edge detection, etc.). Using deep learning we can instead *learn* these types of filters which are then stacked on top of each other to automatically discover high-level concepts. We call this stacking and learning of higher-level features based on lower-level inputs the *compositionality* of Convolutional Neural Networks.

CNNs are built by stacking a sequence of layers where each layer is responsible for a given task. CONV layers will learn a set of K convolutional filters, each of which are size $F \times F$ pixels. We then apply activation layers on top of the CONV layers to obtain a nonlinear transformation. POOL layers help reduce the spatial dimensions of the input volume as it flows through the network.

Once the input volume is sufficiently small, we can apply FC layers which are our traditional dot product layers from Chapter 12, eventually feeding into a softmax classifier for our final output predictions.

Batch normalization layers are used to standardize inputs to a CONV or activation layer by computing the mean and standard deviation across a mini-batch. A dropout layer can then be applied to randomly disconnect nodes from a given input to an output, helping to reduce overfitting.

Finally, we wrapped up the chapter by reviewing common CNN architectures that you can use to implement your own networks. In our next chapter, we’ll implement your first CNN in Keras, ShallowNet, based on the layer patterns we mentioned above. Future chapters will discuss deeper network architectures such as the seminal LeNet architecture [19] and variants of the VGGNet architecture [100].

