

Chapter 9

Convolutional Networks

Convolutional networks (LeCun, 1989), also known as **convolutional neural networks** or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels. Convolutional networks have been tremendously successful in practical applications. The name “convolutional neural network” indicates that the network employs a mathematical operation called **convolution**. Convolution is a specialized kind of linear operation. *Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.*

In this chapter, we will first describe what convolution is. Next, we will explain the motivation behind using convolution in a neural network. We will then describe an operation called **pooling**, which almost all convolutional networks employ. Usually, the operation used in a convolutional neural network does not correspond precisely to the definition of convolution as used in other fields such as engineering or pure mathematics. We will describe several variants on the convolution function that are widely used in practice for neural networks. We will also show how convolution may be applied to many kinds of data, with different numbers of dimensions. We then discuss means of making convolution more efficient. Convolutional networks stand out as an example of neuroscientific principles influencing deep learning. We will discuss these neuroscientific principles, then conclude with comments about the role convolutional networks have played in the history of deep learning. One topic this chapter does not address is how to choose the architecture of your convolutional network. The goal of this chapter is to describe the kinds of tools that convolutional networks provide, while chapter 11

describes general guidelines for choosing which tools to use in which circumstances. Research into convolutional network architectures proceeds so rapidly that a new best architecture for a given benchmark is announced every few weeks to months, rendering it impractical to describe the best architecture in print. However, the best architectures have consistently been composed of the building blocks described here.

9.1 The Convolution Operation

In its most general form, convolution is an operation on two functions of a real-valued argument. To motivate the definition of convolution, we start with examples of two functions we might use.

Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output $x(t)$, the position of the spaceship at time t . Both x and t are real-valued, i.e., we can get a different reading from the laser sensor at any instant in time.

Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average together several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function $w(a)$, where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int x(a)w(t-a)da \tag{9.1}$$

This operation is called **convolution**. The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t) \tag{9.2}$$

In our example, w needs to be a valid probability density function, or the output is not a weighted average. Also, w needs to be 0 for all negative arguments, or it will look into the future, which is presumably beyond our capabilities. These limitations are particular to our example though. In general, convolution is defined for any functions for which the above integral is defined, and may be used for other purposes besides taking weighted averages.

In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the **input** and the second

argument (in this example, the function w) as the **kernel**. The output is sometimes referred to as the **feature map**.

In our example, the idea of a laser sensor that can provide measurements at every instant in time is not realistic. Usually, when we work with data on a computer, time will be discretized, and our sensor will provide data at regular intervals. In our example, it might be more realistic to assume that our laser provides a measurement once per second. The time index t can then take on only integer values. If we now assume that x and w are defined only on integer t , we can define the discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \quad (9.3)$$

In machine learning applications, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as tensors. Because each element of the input and kernel must be explicitly stored separately, we usually assume that these functions are zero everywhere but the finite set of points for which we store the values. This means that in practice we can implement the infinite summation as a summation over a finite number of array elements.

Finally, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel K :

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n). \quad (9.4)$$

Convolution is commutative, meaning we can equivalently write:

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n). \quad (9.5)$$

Usually the latter formula is more straightforward to implement in a machine learning library, because there is less variation in the range of valid values of m and n .

The commutative property of convolution arises because we have **flipped** the kernel relative to the input, in the sense that as m increases, the index into the input increases, but the index into the kernel decreases. The only reason to flip the kernel is to obtain the commutative property. While the commutative property

is useful for writing proofs, it is not usually an important property of a neural network implementation. Instead, many neural network libraries implement a related function called the **cross-correlation**, which is the same as convolution but without flipping the kernel:

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n). \quad (9.6)$$

Many machine learning libraries implement cross-correlation but call it convolution. In this text we will follow this convention of calling both operations convolution, and specify whether we mean to flip the kernel or not in contexts where kernel flipping is relevant. In the context of machine learning, the learning algorithm will learn the appropriate values of the kernel in the appropriate place, so an algorithm based on convolution with kernel flipping will learn a kernel that is flipped relative to the kernel learned by an algorithm without the flipping. It is also rare for convolution to be used alone in machine learning; instead convolution is used simultaneously with other functions, and the combination of these functions does not commute regardless of whether the convolution operation flips its kernel or not.

See figure 9.1 for an example of convolution (without kernel flipping) applied to a 2-D tensor.

Discrete convolution can be viewed as multiplication by a matrix. However, the matrix has several entries constrained to be equal to other entries. For example, for univariate discrete convolution, each row of the matrix is constrained to be equal to the row above shifted by one element. This is known as a **Toeplitz matrix**. In two dimensions, a **doubly block circulant matrix** corresponds to convolution. In addition to these constraints that several elements be equal to each other, convolution usually corresponds to a very sparse matrix (a matrix whose entries are mostly equal to zero). This is because the kernel is usually much smaller than the input image. Any neural network algorithm that works with matrix multiplication and does not depend on specific properties of the matrix structure should work with convolution, without requiring any further changes to the neural network. Typical convolutional neural networks do make use of further specializations in order to deal with large inputs efficiently, but these are not strictly necessary from a theoretical perspective.

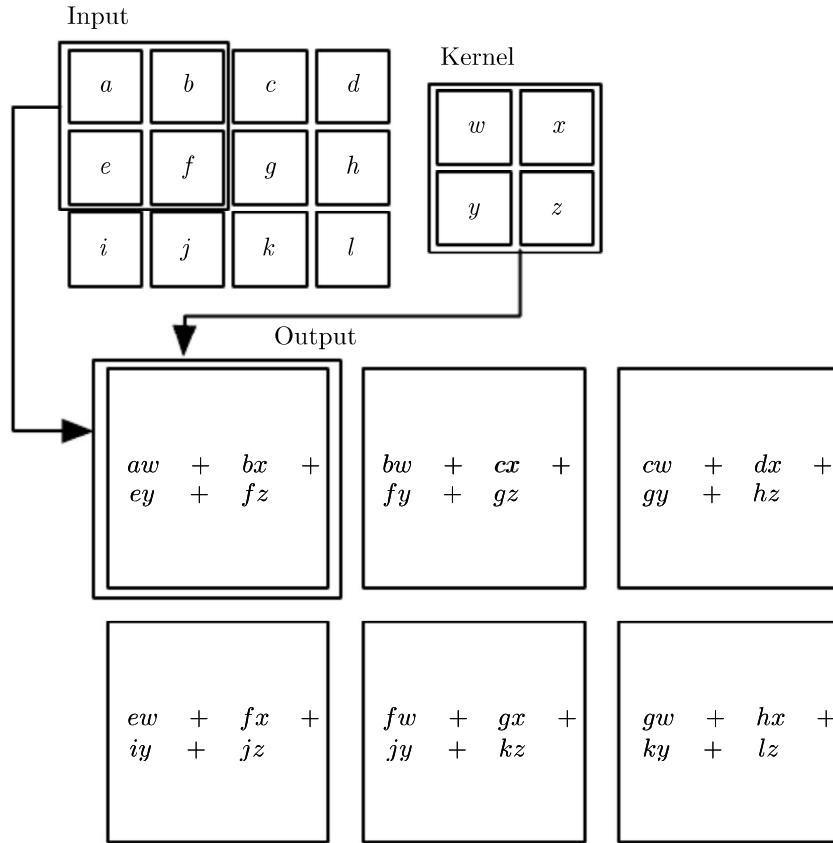


Figure 9.1: An example of 2-D convolution without kernel-flipping. In this case we restrict the output to only positions where the kernel lies entirely within the image, called “valid” convolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor.

9.2 Motivation

Convolution leverages three important ideas that can help improve a machine learning system: **sparse interactions**, **parameter sharing** and **equivariant representations**. Moreover, convolution provides a means for working with inputs of variable size. We now describe each of these ideas in turn.

Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means every output unit interacts with every input unit. Convolutional networks, however, typically have **sparse interactions** (also referred to as **sparse connectivity** or **sparse weights**). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations. These improvements in efficiency are usually quite large. If there are m inputs and n outputs, then matrix multiplication requires $m \times n$ parameters and the algorithms used in practice have $O(m \times n)$ runtime (per example). If we limit the number of connections each output may have to k , then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime. For many practical applications, it is possible to obtain good performance on the machine learning task while keeping k several orders of magnitude smaller than m . For graphical demonstrations of sparse connectivity, see figure 9.2 and figure 9.3. In a deep convolutional network, units in the deeper layers may *indirectly* interact with a larger portion of the input, as shown in figure 9.4. This allows the network to efficiently describe complicated interactions between many variables by constructing such interactions from simple building blocks that each describe only sparse interactions.

Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has **tied weights**, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere. In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters

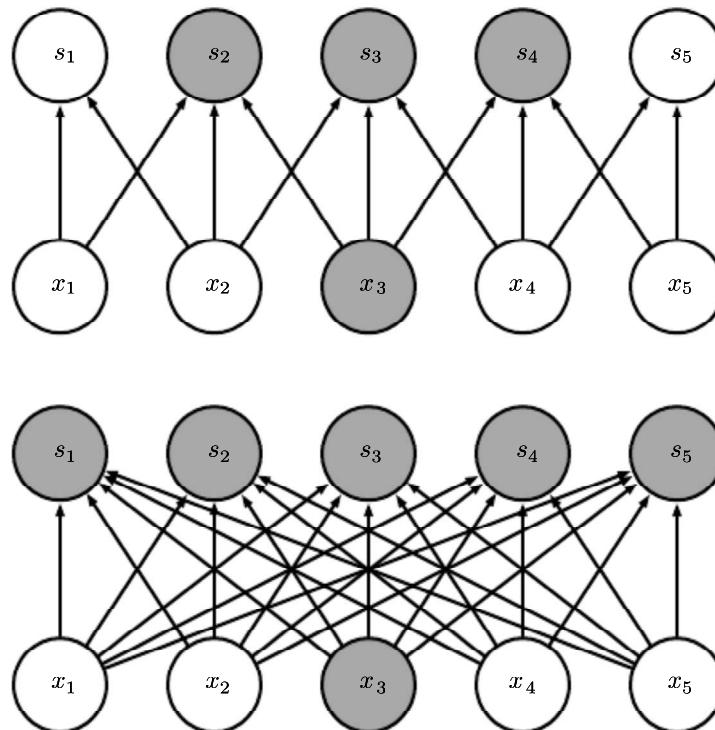


Figure 9.2: *Sparse connectivity, viewed from below:* We highlight one input unit, x_3 , and also highlight the output units in \mathbf{s} that are affected by this unit. (Top) When \mathbf{s} is formed by convolution with a kernel of width 3, only three outputs are affected by \mathbf{x} . (Bottom) When \mathbf{s} is formed by matrix multiplication, connectivity is no longer sparse, so all of the outputs are affected by x_3 .

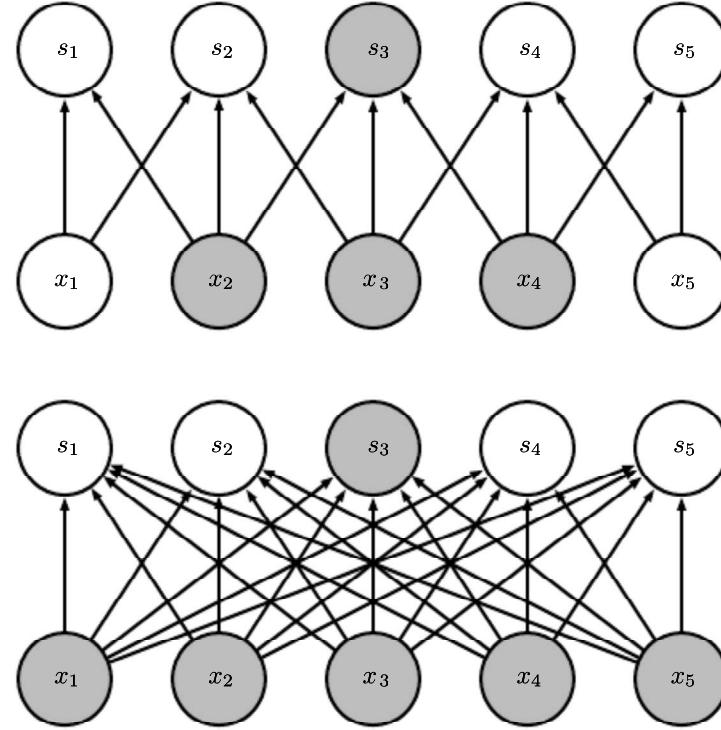


Figure 9.3: *Sparse connectivity, viewed from above:* We highlight one output unit, s_3 , and also highlight the input units in \mathbf{x} that affect this unit. These units are known as the **receptive field** of s_3 . (Top) When \mathbf{s} is formed by convolution with a kernel of width 3, only three inputs affect s_3 . (Bottom) When \mathbf{s} is formed by matrix multiplication, connectivity is no longer sparse, so all of the inputs affect s_3 .

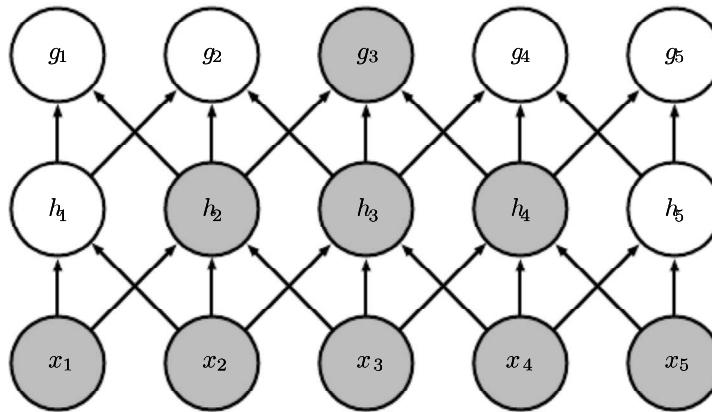


Figure 9.4: The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. This effect increases if the network includes architectural features like strided convolution (figure 9.12) or pooling (section 9.3). This means that even though *direct* connections in a convolutional net are very sparse, units in the deeper layers can be *indirectly* connected to all or most of the input image.

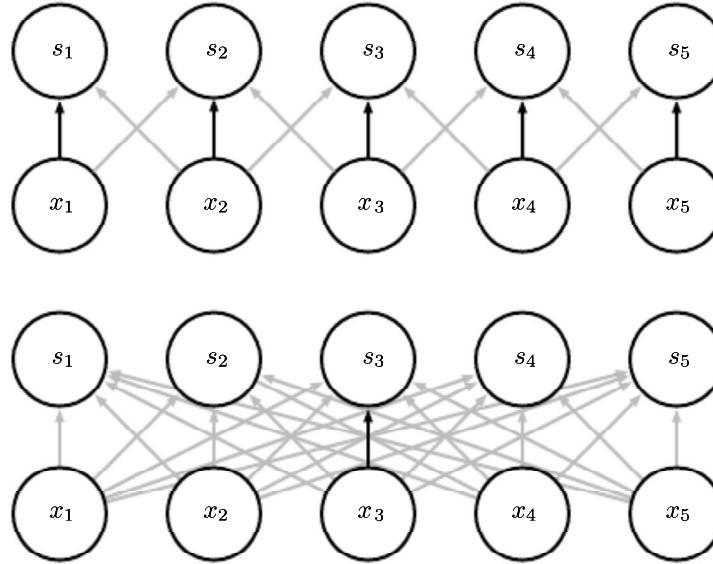


Figure 9.5: Parameter sharing: Black arrows indicate the connections that use a particular parameter in two different models. (*Top*)The black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations. (*Bottom*)The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing so the parameter is used only once.

for every location, we learn only one set. This does not affect the runtime of forward propagation—it is still $O(k \times n)$ —but it does further reduce the storage requirements of the model to k parameters. Recall that k is usually several orders of magnitude less than m . Since m and n are usually roughly the same size, k is practically insignificant compared to $m \times n$. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency. For a graphical depiction of how parameter sharing works, see figure 9.5.

As an example of both of these first two principles in action, figure 9.6 shows how sparse connectivity and parameter sharing can dramatically improve the efficiency of a linear function for detecting edges in an image.

In the case of convolution, the particular form of parameter sharing causes the layer to have a property called **equivariance** to translation. To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translates the input, i.e., shifts it, then the convolution function is equivariant to g . For example, let I be a function giving image brightness at integer coordinates. Let g be a function

mapping one image function to another image function, such that $I' = g(I)$ is the image function with $I'(x, y) = I(x - 1, y)$. This shifts every pixel of I one unit to the right. If we apply this transformation to I , then apply convolution, the result will be the same as if we applied convolution to I' , then applied the transformation g to the output. When processing time series data, this means that convolution produces a sort of timeline that shows when different features appear in the input. If we move an event later in time in the input, the exact same representation of it will appear in the output, just later in time. Similarly with images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network. The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image. In some cases, we may not wish to share parameters across the entire image. For example, if we are processing images that are cropped to be centered on an individual's face, we probably want to extract different features at different locations—the part of the network processing the top of the face needs to look for eyebrows, while the part of the network processing the bottom of the face needs to look for a chin.

Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations.

Finally, some kinds of data cannot be processed by neural networks defined by matrix multiplication with a fixed-shape matrix. Convolution enables processing of some of these kinds of data. We discuss this further in section 9.7.

9.3 Pooling

A typical layer of a convolutional network consists of three stages (see figure 9.7). In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the **detector** stage. In the third stage, we use a **pooling function** to modify the output of the layer further.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the **max pooling** (Zhou



Figure 9.6: *Efficiency of edge detection.* The image on the right was formed by taking each pixel in the original image and subtracting the value of its neighboring pixel on the left. This shows the strength of all of the vertically oriented edges in the input image, which can be a useful operation for object detection. Both images are 280 pixels tall. The input image is 320 pixels wide while the output image is 319 pixels wide. This transformation can be described by a convolution kernel containing two elements, and requires $319 \times 280 \times 3 = 267,960$ floating point operations (two multiplications and one addition per output pixel) to compute using convolution. To describe the same transformation with a matrix multiplication would take $320 \times 280 \times 319 \times 280$, or over eight billion, entries in the matrix, making convolution four billion times more efficient for representing this transformation. The straightforward matrix multiplication algorithm performs over sixteen billion floating point operations, making convolution roughly 60,000 times more efficient computationally. Of course, most of the entries of the matrix would be zero. If we stored only the nonzero entries of the matrix, then both matrix multiplication and convolution would require the same number of floating point operations to compute. The matrix would still need to contain $2 \times 319 \times 280 = 178,640$ entries. Convolution is an extremely efficient way of describing transformations that apply the same linear transformation of a small, local region across the entire input. (Photo credit: Paula Goodfellow)

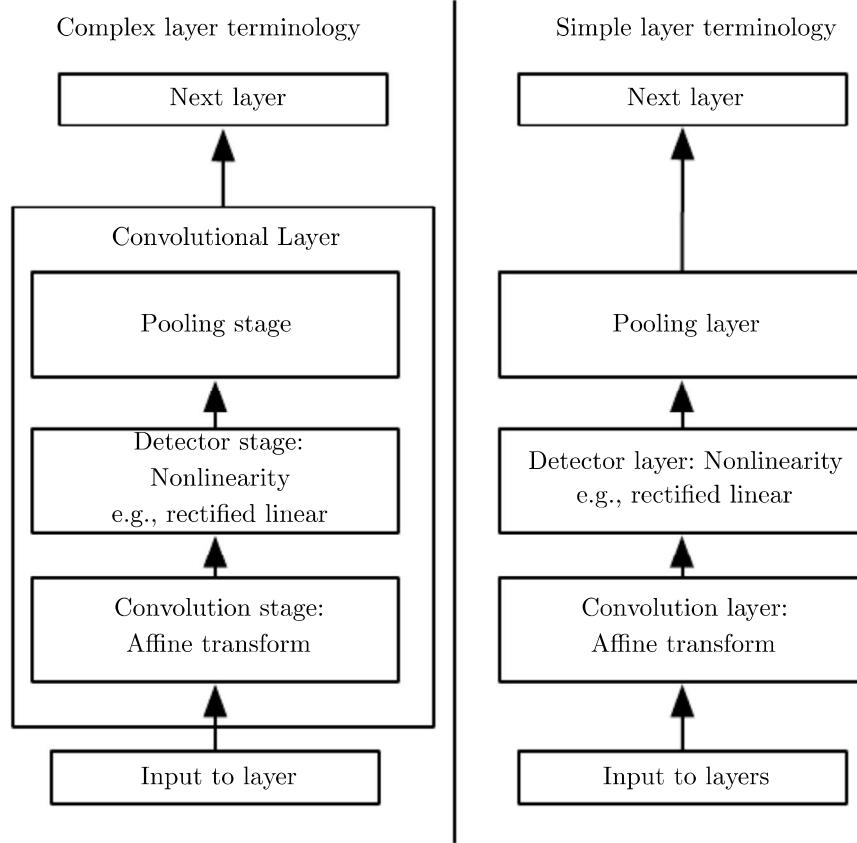


Figure 9.7: The components of a typical convolutional neural network layer. There are two commonly used sets of terminology for describing these layers. (*Left*) In this terminology, the convolutional net is viewed as a small number of relatively complex layers, with each layer having many “stages.” In this terminology, there is a one-to-one mapping between kernel tensors and network layers. In this book we generally use this terminology. (*Right*) In this terminology, the convolutional net is viewed as a larger number of simple layers; every step of processing is regarded as a layer in its own right. This means that not every “layer” has parameters.

and Chellappa, 1988) operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the L^2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel.

In all cases, pooling helps to make the representation become approximately **invariant** to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. See figure 9.8 for an example of how this works. *Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.* For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face. In other contexts, it is more important to preserve the location of a feature. For example, if we want to find a corner defined by two edges meeting at a specific orientation, we need to preserve the location of the edges well enough to test whether they meet.

The use of pooling can be viewed as adding an infinitely strong prior that the function the layer learns must be invariant to small translations. When this assumption is correct, it can greatly improve the statistical efficiency of the network.

Pooling over spatial regions produces invariance to translation, but if we pool over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant to (see figure 9.9).

Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than detector units, by reporting summary statistics for pooling regions spaced k pixels apart rather than 1 pixel apart. See figure 9.10 for an example. This improves the computational efficiency of the network because the next layer has roughly k times fewer inputs to process. When the number of parameters in the next layer is a function of its input size (such as when the next layer is fully connected and based on matrix multiplication) this reduction in the input size can also result in improved statistical efficiency and reduced memory requirements for storing the parameters.

For many tasks, pooling is essential for handling inputs of varying size. For example, if we want to classify images of variable size, the input to the classification layer must have a fixed size. This is usually accomplished by varying the size of an offset between pooling regions so that the classification layer always receives the same number of summary statistics regardless of the input size. For example, the final pooling layer of the network may be defined to output four sets of summary statistics, one for each quadrant of an image, regardless of the image size.

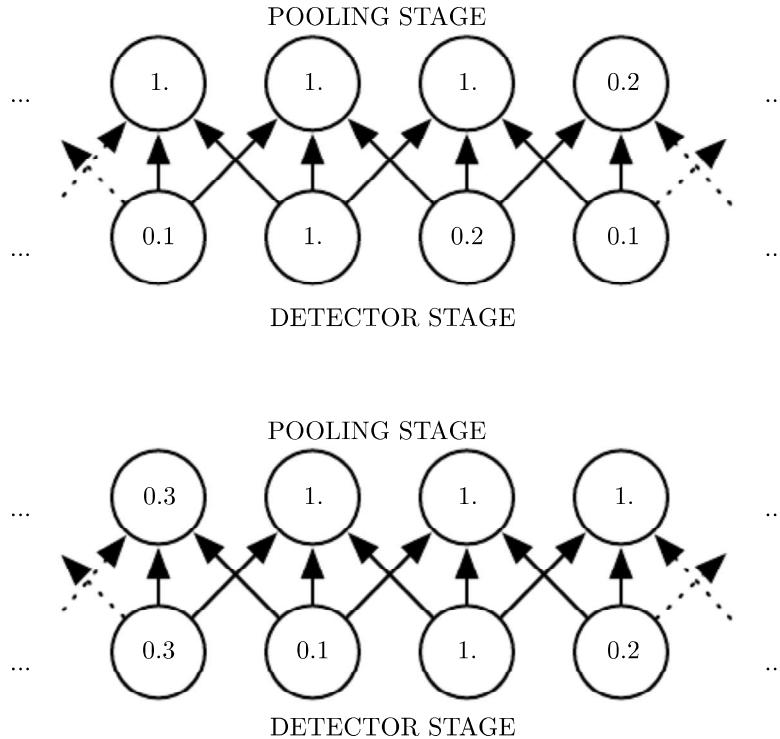


Figure 9.8: Max pooling introduces invariance. (*Top*)A view of the middle of the output of a convolutional layer. The bottom row shows outputs of the nonlinearity. The top row shows the outputs of max pooling, with a stride of one pixel between pooling regions and a pooling region width of three pixels. (*Bottom*)A view of the same network, after the input has been shifted to the right by one pixel. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are only sensitive to the maximum value in the neighborhood, not its exact location.

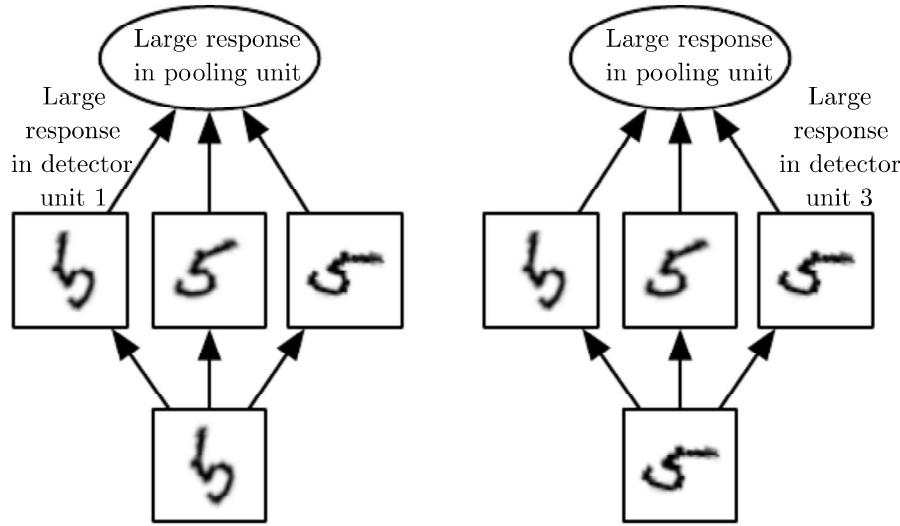


Figure 9.9: *Example of learned invariances*: A pooling unit that pools over multiple features that are learned with separate parameters can learn to be invariant to transformations of the input. Here we show how a set of three learned filters and a max pooling unit can learn to become invariant to rotation. All three filters are intended to detect a hand-written 5. Each filter attempts to match a slightly different orientation of the 5. When a 5 appears in the input, the corresponding filter will match it and cause a large activation in a detector unit. The max pooling unit then has a large activation regardless of which detector unit was activated. We show here how the network processes two different inputs, resulting in two different detector units being activated. The effect on the pooling unit is roughly the same either way. This principle is leveraged by maxout networks (Goodfellow *et al.*, 2013a) and other convolutional networks. Max pooling over spatial positions is naturally invariant to translation; this multi-channel approach is only necessary for learning other transformations.

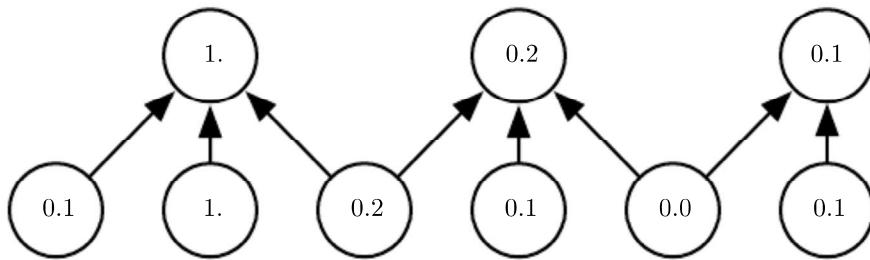


Figure 9.10: *Pooling with downsampling*. Here we use max-pooling with a pool width of three and a stride between pools of two. This reduces the representation size by a factor of two, which reduces the computational and statistical burden on the next layer. Note that the rightmost pooling region has a smaller size, but must be included if we do not want to ignore some of the detector units.

Some theoretical work gives guidance as to which kinds of pooling one should use in various situations (Boureau *et al.*, 2010). It is also possible to dynamically pool features together, for example, by running a clustering algorithm on the locations of interesting features (Boureau *et al.*, 2011). This approach yields a different set of pooling regions for each image. Another approach is to *learn* a single pooling structure that is then applied to all images (Jia *et al.*, 2012).

Pooling can complicate some kinds of neural network architectures that use top-down information, such as Boltzmann machines and autoencoders. These issues will be discussed further when we present these types of networks in part III. Pooling in convolutional Boltzmann machines is presented in section 20.6. The inverse-like operations on pooling units needed in some differentiable networks will be covered in section 20.10.6.

Some examples of complete convolutional network architectures for classification using convolution and pooling are shown in figure 9.11.

9.4 Convolution and Pooling as an Infinitely Strong Prior

Recall the concept of a **prior probability distribution** from section 5.2. This is a probability distribution over the parameters of a model that encodes our beliefs about what models are reasonable, before we have seen any data.

Priors can be considered weak or strong depending on how concentrated the probability density in the prior is. A weak prior is a prior distribution with high entropy, such as a Gaussian distribution with high variance. Such a prior allows the data to move the parameters more or less freely. A strong prior has very low entropy, such as a Gaussian distribution with low variance. Such a prior plays a more active role in determining where the parameters end up.

An infinitely strong prior places zero probability on some parameters and says that these parameter values are completely forbidden, regardless of how much support the data gives to those values.

We can imagine a convolutional net as being similar to a fully connected net, but with an infinitely strong prior over its weights. This infinitely strong prior says that the weights for one hidden unit must be identical to the weights of its neighbor, but shifted in space. The prior also says that the weights must be zero, except for in the small, spatially contiguous receptive field assigned to that hidden unit. Overall, we can think of the use of convolution as introducing an infinitely strong prior probability distribution over the parameters of a layer. This prior