8 Convolution

Objective

The convolution neural network is a popular network for image processing applications, including object classification, localization, detection and instance segmentation. It is also used in other applications, such as speech and natural language processing. This chapter describes the basic convolution operation and how it is used in combination with other layers, like pooling and fully connected, to form complete networks. The chapter also explains how to backpropagate through convolution and pooling layers in order to compute the training gradient.

Theory and Examples

Why Use Convolution?

One of the drawbacks of using the standard multilayer network, described in Chapter 2, for image processing is that the size of the weight matrix in the first layer can become very large. This can lead to problems with overfitting, in addition to the increased computation. Instead of using standard matrix multiplication, in which every input element is fully connected to every neuron, it would be helpful to constrain the operation in some way.

Of course, if we constrain the operation, we may limit the possible functions that we can create with the network. We need to choose a structure that is suitable for the problem we want to solve. The process of restricting the form of a network is an example of what is called an *inductive bias*. It is a way of embedding *prior* knowledge about how to solve a particular problem into the neural network. If done correctly, this can make the problem easier for the network to solve.

If we consider operations that transform one image into another, it seems reasonable to consider transformations that are *position invariant*, which means that their effect on a point does not depend on the position of the point. (We can also say that a transformation is position invariant if it commutes with translation – the result will be the same whether the image translation occurs before or after the transformation.) Also, we will consider linear transformations, since the layers in a multilayer network consist of a linear transformation followed by a nonlinear activation function.

Recall from Chapter 6 of NND2 that linear transformations can be represented as matrix multiplications. It turns out that any linear, *position invariant* transformation can also be represented by the convolution between the impulse response of the operation (sometimes called the point spread function) and the original image. (See [Rosenfeld, 1969].) This can be shown to be matrix multiplication with a matrix of a particular structure. (See Eq. 8.12.)

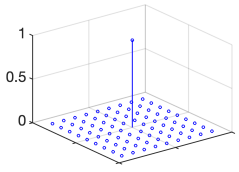
The convolution operation is ubiquitous throughout math, science and engineering. One of the earliest applications was by d'Alembert in 1754 to derive the Taylor series expansion. (See [le Rond d'Alembert et al., 1754].)

Deriving the Convolution Operation

To illustrate that convolution can represent linear, position invariant transformations, we need to define a couple of functions. First,

consider an image that consists of a single pixel with a value of 1, with the remaining pixels equal to zero:

$$\delta_{i,j} = \begin{cases} 1 & i = j = 0 \\ 0 & \text{else} \end{cases} \tag{8.1}$$



This δ is called an *impulse* function or a one-point image.

Next we define the *impulse response*, or point spread function. This is the response to a given linear, position invariant transformation when it is applied to the impulse function. For example, if we have a transformation $\mathcal{W}$, the impulse response would be

$$w_{i,j} = \mathcal{W}(\delta_{i,j}) \tag{8.2}$$

(Note that in this equation $\delta_{i,j}$ and $w_{i,j}$ represent the entire images. The transformation $\mathcal{W}$ does not simply use one point in the input image to produce one point in the output image. It uses a set of points in the input image to produce each point in the ouput image.)

Now we can put these two ideas together. First, write a given image as a sum of impulses (one point images) located at each pixel:

$$p_{m,n} = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} p_{i,j} \delta_{m-i,n-j} \tag{8.3}$$

(Note that, although we show infinite sums, the pixel values will be zero outside some range.)

Now we perform the linear, position invariant transformation $\mathcal{W}$:

$$z_{m,n} = \mathcal{W}(\mathbf{P})_{m,n} = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} p_{i,j} \mathcal{W}(\delta_{m-i,n-j}) \tag{8.4}$$

$$= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} p_{i,j} w_{m-i,n-j} \tag{8.5}$$

where we have used the linearity and position invariant properties of $\mathcal{W}$ and the definition of impulse response in Eq. 8.2. The expression in Eq. 8.5 is the convolution between the image and the impulse response. The convolution operation is cummutative, and the order of the functions being convolved is not important, so we can also write

$$z_{m,n} = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} w_{i,j} p_{m-i,n-j} \tag{8.6}$$

We can write the result in abbreviated notation as follows:

$$\mathbf{Z} = \mathbf{P} * \mathbf{W} = \mathbf{W} * \mathbf{P} \tag{8.7}$$

where $*$ represents the convolution operator.

Convolution and Cross-Correlation

In addition to assumptions about the linearity and position invariance of the transformation, it also seems reasonable to choose transformations such that the value at an output pixel z_{i^*,j^*} should be determined only by pixels in the neighborhood of the corresponding input pixel $p_{i,j}$. (See [Montgomery and Broome, 1962] for an early discussion of this idea, which they called the neighborhood modification process and which we now refer to as a local receptive field.) This would mean that the impulse response of the transformation would be nonzero only over a limited region. In this case the limits of the summation in Eq. 8.6 would be limited to the extent of the impulse response.

$$z_{m,n} = \sum_{i=0}^{r-1} \sum_{j=0}^{c-1} w_{i,j} p_{m-i,n-j} \tag{8.8}$$

where r and c represent the number of rows and columns in the matrix $\mathbf{W}$. We often refer to $\mathbf{W}$ (which represents the impulse response of the desired transformation) as a *kernel* or a *filter*. (We have indicated that the nonzero region of the kernel starts at an index of 0 for both row and column. This is for simplicity of the exposition. We will have more to say about the details of the operation in the next section.)

The convolution operation is very closely related to the *cross-correlation* operation, which is defined as follows:

$$z_{m,n} = \sum_{i=0}^{r-1} \sum_{j=0}^{c-1} w_{i,j} p_{m+i,n+j} \tag{8.9}$$

which we can write in abbreviated notation as

$$\mathbf{Z} = \mathbf{W} \otimes \mathbf{P} \tag{8.10}$$

The only difference between convolution and cross-correlation is in the meaning of the kernel. It is flipped horizontally and vertically between the two operations. (See Solved Problem P8.1 for

a discussion of the relationship between convolution and correlation kernels.) Since we will be training the kernels in convolution networks, the orientation of the kernel is not usually of critical importance, since during training the network will learn the appropriate orientation. In most deep learning frameworks, convolution layers are actually implemented as cross-correlations, and most papers use the term convolution to refer to a cross-correlation operation. We will follow this current convention in the remainder of this text. We will use the cross-correlation operation, while referring to it as convolution, unless specifically noted.

Weight Sharing

The cross-correlation (or convolution) operation can be represented as a matrix multiplication. For example, if we have a 3×3 image and a 2×2 kernel, the operation can be represented as

$$\begin{bmatrix} z_{1,1} & z_{1,2} \\ z_{2,1} & z_{2,2} \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix} \otimes \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,1} & p_{3,2} & p_{3,3} \end{bmatrix} \tag{8.11}$$

$$\begin{bmatrix} z_{1,1} \\ z_{2,1} \\ z_{1,2} \\ z_{2,2} \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{2,1} & 0 & w_{1,2} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & w_{1,1} & w_{2,1} & 0 & w_{1,2} & w_{2,2} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{1,1} & w_{2,1} & 0 & w_{1,2} & w_{2,2} & 0 \\ 0 & 0 & 0 & 0 & w_{1,1} & w_{2,1} & 0 & w_{1,2} & w_{2,2} \end{bmatrix} \begin{bmatrix} p_{1,1} \\ p_{2,1} \\ p_{3,1} \\ p_{1,2} \\ p_{2,2} \\ p_{3,2} \\ p_{1,3} \\ p_{2,3} \\ p_{3,3} \end{bmatrix} \tag{8.12}$$

where Eq, 8.11 shows the operation as a cross-correlation and Eq. 8.12 shows it as standard matrix multiplication. (See Figure 8.2 for a graphical representation of the cross-correlation.)

Note that the same weights appear in each row of the matrix. This is often referred to as *weight sharing*, since each output neuron uses the same weights. It occurs because of the inductive bias we imposed on the network by saying the transformation should be position invariant, and there are many zeros since we specified that the transformation should have a local receptive field. The concept of inductive bias is especially important in problems where the

input space is high dimensional. Clever uses of inductive bias will surely continue to expand the use of neural networks to seemingly intractable problems.

Details of the Convolution Operation

Eq. 8.6 describes the general convolution operation in theory, however, many practical details are left out. For example, the input image and the kernel are normally defined over a finite range of indices. In that case, for what values of m and n should $z_{m,n}$ be computed? In this section we want to describe the details of the operation in terms of specific image and kernel sizes.

Consider first the input image $\mathbf{P}$. For network inputs we generally use the letter R to represent the number of elements in the input vector. When dealing with images, we have two input dimensions, which we will represent by R_r and R_c, for the row and column sizes. (We will later add an additional dimension for number of channels [colors], and a depth dimension can be added for volumes in 3D convolution. In this section we want to concentrate on just the basic convolution operation on images.)

The output of the convolution will also be an image. For standard multilayer networks, we indicated the number of neurons by S. Now we will need a row and column size: S_r and S_c.

The dimension of the convolution weight matrix $\mathbf{W}$ (also called the kernel) is not tied to the size of the input, unlike in the multilayer network. We will indicate the number of rows and columns by r and c.

These dimensions are illustrated in Figure 8.1. Note that the bias in this diagram has the same dimension as the output image, since there is one element of the bias for each neuron. This is called an *untied bias*, since the bias is not tied to the kernel, but to the neuron. For a *tied bias*, the bias in this figure would be a scalar, and there would be only one for each kernel. As we will see in a later section, there can be multiple kernels in a layer.

Next, we should discuss the registration between the pixel locations in the input image and the corresponding pixels in the output image. In Eq. 8.6, the limits on the summation run from $-\infty$ to ∞, but the kernel or the image will be equal to zero, or undefined, outside some range. When the convolution operation is used in neural networks, the computation of a given output pixel $z_{m,n}$ will not

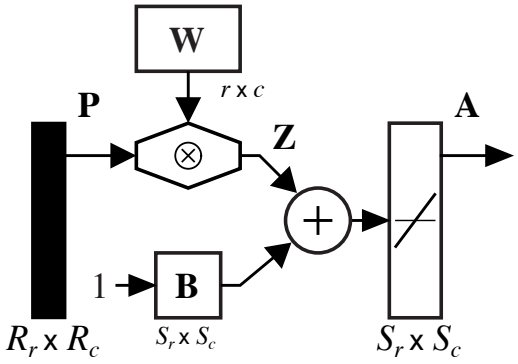


Figure 8.1: Basic Convolution Layer

be performed unless the kernel fits completely within the defined region of the input image. This is illustrated in Figure 8.2. Here we have a 3×3 input image $\mathbf{P}$, a 2×2 kernel $\mathbf{W}$, and $\mathbf{Z} = \mathbf{W} \otimes \mathbf{P}$.

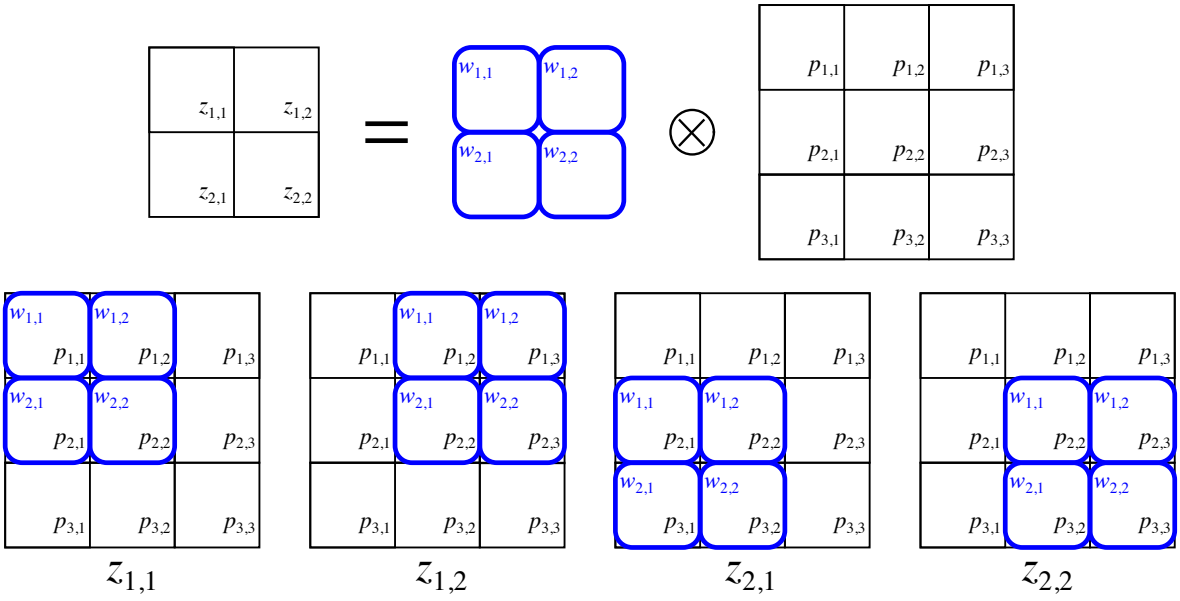


Figure 8.2: Example Convolution Layout

Here we use the convention from NND2, where indices begin at 1 for all vectors and matrices. For this reason we modify the indices in Eq. 8.9 to start at 1, instead of 0, and adjust the offsets so that in Eq. 8.13 $w_{1,1}$ multiplies the same pixel of $\mathbf{P}$ as $w_{0,0}$ did in Eq. 8.9. This allows us to start the computation of Eq. 8.13 at $m = 1$ and $n = 1$.

$$z_{m,n} = \sum_{i=1}^{r} \sum_{j=1}^{c} w_{i,j} p_{m+i-1,n+j-1} \tag{8.13}$$

The final value of m to be computed in Eq. 8.13 will be when $m + r - 1 = R_r$, or $m = R_r - r + 1$. In a similar way, we can find that the final value for n will be $n = R_c - c + 1$. (For the example in Figure 8.2, the maximum m and n are $m = 3 - 2 + 1 = 2$ and $n = 3 - 2 + 1 = 2$.) Therefore, the number of neurons in the convolution layer (the size of the output image) is determined by the input size and the kernel size:

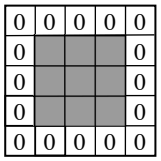
It is worth noting that in a convolution layer the input size could be changed, which would change the output size, but the kernel size could remain the same.

$$S_r = R_r - r + 1 \tag{8.14}$$

$$S_c = R_c - c + 1 \tag{8.15}$$

Padding and Stride

The basic convolution operation is often modified in two ways that can change the size of the output image. The first change is called *padding*. From Equations 8.14 and 8.15, we can see that the output image will be smaller than the input image. Padding artificially increases the size of the input image by adding pixels of value 0 around the edge of the image. This can be done in such a way that the output of the convolution will be the same size as the original input image. If we add P^d pixels around the outside of the original image, this will add $2P^d$ to both the row size and the column size. The new sizes of the output image will be



$$S_r = R_r + 2P^d - r + 1 \tag{8.16}$$

$$S_c = R_c + 2P^d - c + 1 \tag{8.17}$$

The other size-modifying change to the convolution operation is the adjustment of *stride*. The output image can be made smaller by taking larger strides, or kernel movements. Normally, the kernel is moved one step at a time when performing the convolution, as shown in Figure 8.2. If the stride is increased to 2, for example, the output image size is reduced by a factor of 2. Figure 8.3 shows an example with a 2×2 kernel, a 4×4 input image and a stride of 2. The output image is 2×2.

Equations 8.18 and 8.19 show how the output image size is determined when both padding and non-unity strides are used

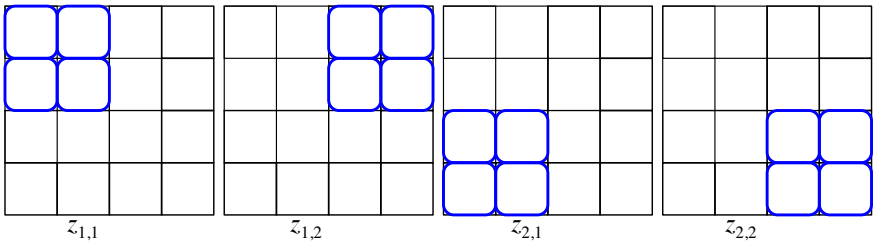


Figure 8.3: Example Convolution with Stride of 2

together. S^t is the stride length.

$$S_r = (R_r + 2P^d - r)/S^t + 1 \tag{8.18}$$

$$S_c = (R_c + 2P^d - c)/S^t + 1 \tag{8.19}$$

Template Matching–Matched Filter

As we showed earlier, the convolution operation can be used to implement any linear, position-invariant transformation on an image. It has been used in image processing since the 1950s for edge detection, noise removal, contrast enhancement, image restoration, etc. (See [Kovásznay and Joseph, 1955] and [Rosenfeld, 1969].) In this section we want to explore its power for determining when two images match. This process is called *template matching*, or *matched filtering*. (See [Montgomery and Broome, 1962] for an early discussion of matched filters.)

We know from Chapter 5 of NND2 that a good way to determine the similarity between two vectors is to compute their inner product. We can see from Eq. 8.13 and Figure 8.2 that each step of the convolution is an inner product between the kernel and a segment of the input image. Therefore, if we construct the kernel in the form of a template that we would like to match in the input image, the magnitude of the convolution at each output pixel will tell us how close the subimage matches the template.

To see how template matching can work, consider a simple problem in which we are trying to distinguish between squares and diamonds. In Figure 8.4, we see two 15×15 black and white images representing examples of our two categories.

If we use 3×3 kernels, what types of templates could we use that would help us distinguish between squares and diamonds? These are very simple objects, so we can select the features manually.

(a) Square (b) Diamond

Figure 8.4: Sample Images to be Recognized

Squares consist of horizontal and vertical lines, and diamonds consist of slanted lines. Some possible kernels are shown in Figure 8.5.

(a) Horizontal (b) Vertical (c) Slash (d) Backslash

Figure 8.5: Sample Kernels

Figure 8.6 illustrates the results of convolving each of the kernels with the square image. We can see the larger magnitudes in locations where horizontal and vertical lines match segments of the image. The slanted kernels don't match with any segments of the image.

(a) Horizontal (b) Vertical (c) Slash (d) Backslash

Figure 8.6: Filtered Versions of the Square

Figure 8.7 shows the results of convolving the kernels with the diamond image. In this case there are no matches when convolving with the horizontal and vertical kernels, but there are matches with the slanted kernels in appropriate segments of the diamond image.

To experiment with the convolution operation with different kernels, use the Deep Learning Demonstration Convolution (**`dl3gd`**).

(a) Horizontal (b) Vertical (c) Slash (d) Backslash

Figure 8.7: Filtered Versions of the Diamond

Hierachical Template Matching

We now see that the convolution operation can be used to detect elementary features in images. Before deep learning, such matched filters were hand-designed. Now we want to train convolution networks to learn which features to extract. In addition, natural images have complex features that can be formed with combinations of multiple simple elements. We want to build a complete architecture that can be trained to recognize realistic objects, using the convolution operation as a building block.

The idea of stacking convolution layers in a hierarchical architecture was proposed by Kunihiko Fukushima in 1980 (see [Fukushima, 1980]), inspired by the hierarchical model of the cat visual cortex developed by David Hubel and Torsten Wiesel in 1962 (see [Hubel and Wiesel, 1962]). What they found was that neurons in the early visual cortex are organized in a hierarchical fashion, where the first cells connected to the cat's retina are responsible for detecting simple patterns like edges and bars, followed by later layers responding to more complex patterns by combining the earlier neuron outputs. (See Chapter 18 of NND2 for a diagram of the human visual system and discussions of various types of receptive fields.) Their work earned them the Nobel Prize for Physiology or Medicine in 1981.

Yann LeCun, in the late 1980s refined the architecture of Fukushima and demonstrated how all of the network weights could be trained simultaneously using gradient-based algorithms, with backpropagation used to compute the gradient. The components of the convolution network architectures in current use are in large part the same ones first proposed by LeCun.

Multiple Kernels

The first step in creating the hierarchical architecture is to have multiple kernels in each convolution layer. The idea is to build up complex features by combining many simple features from previous layers. LeCun called the output of one kernel operation a *feature map*

(FM). (This is not to be confused with the Self-Organizing Feature Maps of Kohonen, described in Chapter 16 of NND2.) Since each convolution layer will have multiple FMs, this adds an additional dimension to the layer output. This idea can be extended to the input image. For example, a color image consists of red, green and blue planes. Therefore, a convolution layer takes a set of FMs as an input and produces another set of FMs as an output.

We will use the letter H^m to represent the number of FMs in Layer m, with $m = 0$ for the input. If there are H^0 FMs (planes) in the input, and H^1 FMs in the first convolution layer, then there will be $H^0 \times H^1$ kernels in Layer 1. To obtain one FM at the output of Layer 1, H^0 kernels are convolved with the H^0 FMs of the input, and the results are added together. This is done H^1 times to produce all H^1 FMs at the output. The general equation for an input coming from Layer n and connecting to layer m would be the following:

$$z_{i,j}^{m,n,h} = \sum_{l=1}^{H^n} \sum_{u=1}^{r^m} \sum_{v=1}^{c^m} w_{u,v}^{m,n,h,l} a_{i+u-1,j+v-1}^{n,l} \tag{8.20}$$

In abbreviated notation, this would be

$$\mathbf{Z}^{m,n,h} = \sum_{l=1}^{H^n} \mathbf{W}^{m,n,h,l} \otimes \mathbf{A}^{n,l} \tag{8.21}$$

where, in $\mathbf{Z}^{m,n,h}$, m represents the current layer number, n represents the previous layer number and h represents the FM number. In most convolution networks the layers are connected sequentially, so that $m = n + 1$. However, new and more complex architectures are being invented daily.

Pooling

In a standard convolution network, a convolution layer is followed by a *pooling* layer, which changes the resolution of the FMs. It performs a type of subsampling, aggregation, consolidation or coarsening. This allows the different layers of the network to operate at different scales, producing a multi-scale operation. Pooling also reduces the size of the output FM and reduces the number of parameters in the network. It is another example of inductive bias, where we assume that a certain structure in the network is appropriate for the data we are working with.

There are several different types of pooling, but they all generally

consolidate several pixels in an input FM to produce one pixel in the output FM. For example, the following equation represents average pooling.

$$z_{i,j} = \left\{ \sum_{k=1}^{r} \sum_{l=1}^{c} v_{r(i-1)+k,c(j-1)+l} \right\} w \tag{8.22}$$

In abbreviated notation, we write average pooling like this.

$$\mathbf{Z} = w \boxplus_{r,c}^{ave} \mathbf{V} \tag{8.23}$$

The average pooling operation in Eq. 8.22 looks similar to the convolution operation in Eq. 8.13. The difference is that all of the elements of the kernel are equal to w. However, the stride used in the pooling operation is almost always equal to the size of the kernel. In other words, the stride would equal r in the vertical direction and c in the horizontal direction. Pooling therefore consolidates all of the $r \times c$ pixels into one pixel, reducing the image size by a corresponding amount.

Another type of pooling is max pooling. This also consolidates multiple pixels into one, but it uses the maximum of the input pixels, rather than the sum.

$$z_{i,j} = max \left\{ v_{r(i-1)+k,c(j-1)+l} | k = 1, ..., r; l = 1, ..., c \right\} \tag{8.24}$$

Note that there is no trainable parameter in the max pooling function.

The abbreviated notation for max pooling is

$$\mathbf{Z} = \boxplus_{r,c}^{max} \mathbf{V} \tag{8.25}$$

Theoretically, any size consolidation window and any stride could be used for pooling. However, in most cases the consolidation window is square, and the stride is equal to the window size, so there is no overlap in the consolidations. The most common choice is $r = 2$, $c = 2$ and $S^t = 2$.

The max pooling operation is illustrated in Figure 8.8, where the window size and stride are equal to 3. The image size goes from 15×15 to 5×5.

Figure 8.9 shows a network with a convolution layer followed by a max pooling layer. Note that in the convolution layer we are using a tied bias, so there is one scalar bias for each kernel. The number of kernels is $H^1 \times H^0$, since there are H^0 planes in the input image and H^1 FMs in the first layer. The dimensions of the two terms

(a) Filtered Diamond

(b) Pooled Filtered Diamond

(c) Filtered Square

(d) Pooled Filtered Square

Figure 8.8: Examples of Max Pooling

coming into the summation in the first layer do not have the same dimension, but we will assume that broadcasting will be done to ensure compatibility, as in the Python language (see Chapter 5).

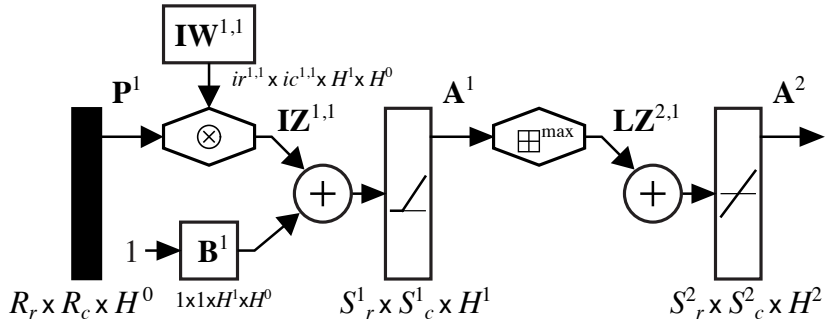


Figure 8.9: Network with Convolution and Pooling Layers

Unlike the convolution layer, in the pooling layer the number of FMs at the input will always be equal to the number of FMs at the output. In Figure 8.9, this means that $H^2 = H^1$. The pooling operation is applied to each FM individually to produce one output FM. In the convolution layer, separate kernels are applied to each input FM, and the results are summed to produce one output FM. The number of input FMs in that case do not constrain the number of output FMs.

Simplified Notation

Since convolution networks often have a large number of layers, it is useful to have a shorthand notation. We can build on the notation introduced in Solved Problem P7.1. Figure 8.10 shows the same network as Figure 8.9, but in the shorthand notation.

Convolution and pooling layers take images as input and produce images as output. In order to create a convolution network that performs classification, we need to convert the images to vectors, so that they can be input to a standard dot product (matrix

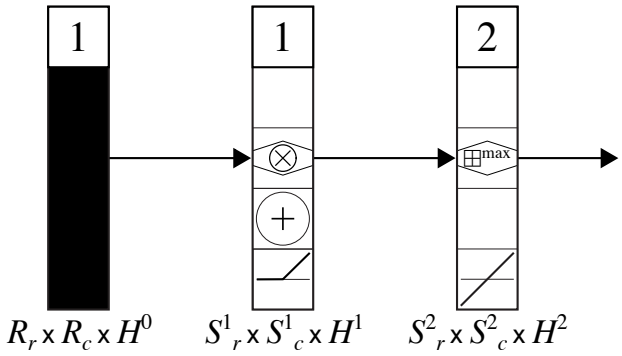
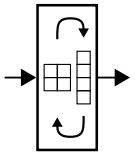


Figure 8.10: Shorthand Notation for Figure 8.9



multiplication) layer. In our diagrams, we show this as a conversion block. It converts the output of a layer from a set of FMs to a single vector. It stacks the columns of the FMs on top of each other, starting from the first FM to the last.

$$vec(\mathbf{A}^m) = \left[\left(\mathbf{a}_1^{m,1}\right)^T \left(\mathbf{a}_2^{m,1}\right)^T \cdots \left(\mathbf{a}_{S_c^m}^{m,1}\right)^T \left(\mathbf{a}_1^{m,2}\right)^T \cdots \left(\mathbf{a}_{S_c^m}^{m,H^m}\right)^T \right]^T \tag{8.26}$$

where $\mathbf{a}_i$ indicates the i^{th} column of matrix $\mathbf{A}$.

Figure 8.11 shows a full five layer convolution network classifier. There are two convolution-pooling groups, followed by a conversion to vector and then a fully connected layer (as in the multilayer network) with a softmax activation function to perform the classification.

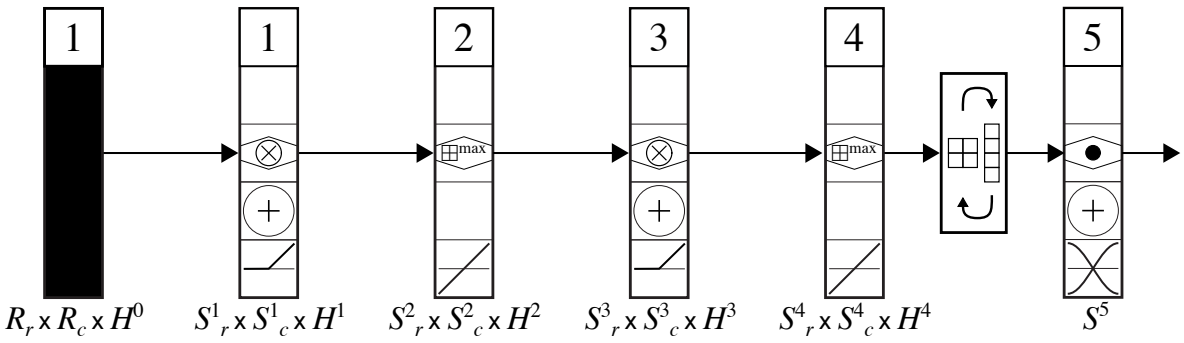


Figure 8.11: Full Convolution Network

Gradient Calculation

Convolution networks are trained using the same gradient-based algorithms we described in Chapter 3, and the gradient is computed

using the same backpropagation (chain rule) process we discussed in several chapters (e.g., see Eq. 7.30). In that process, we computed a sensitivity (defined in Eq. 3.38 [and Eq. 11.29 of NND2] and repeated here), which is the derivative of the performance (loss) function with respect to the net input at a given layer.

$$\mathbf{s}^m \triangleq \frac{\partial F(\mathbf{x})}{\partial \mathbf{n}^m} \tag{8.27}$$

In Eq. 7.30 we compute the sensitivities by backpropagating across the layer in stages: across the net input function, across the weight function and across the transfer function. To illustrate this process with convolution networks, it is helpful to perform one step at a time. For this reason, we will introduce a new intermediate variable at each step. We will also change the symbol for the sensitivity to be consistent with the other intermediate variables. The definitions are as follows:

$$\mathbf{dN} \equiv \frac{\partial F}{\partial \mathbf{N}}, \mathbf{dA} \equiv \frac{\partial F}{\partial \mathbf{A}}, \mathbf{dZ} \equiv \frac{\partial F}{\partial \mathbf{Z}}, \mathbf{dW} \equiv \frac{\partial F}{\partial \mathbf{W}}, \mathbf{db} \equiv \frac{\partial F}{\partial \mathbf{B}} \tag{8.28}$$

where **dN** is the original sensitivity, although it is a matrix in the case of the convolution network, since the net input is a matrix.

The process of backpropagating across a layer is shown in Figure 8.12. (For simplicity, this figure shows just one FM in each layer.) In this figure, the process of going backward from $\mathbf{dN}^m$ to $\mathbf{dN}^n$ is analogous to going from $\mathbf{s}^{m+1}$ to $\mathbf{s}^m$ in Eq. 3.49. (See also Figure 3.5.)

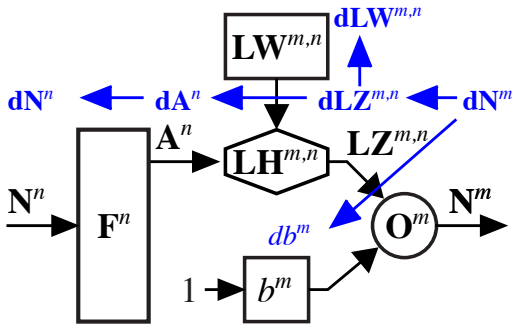


Figure 8.12: Backpropagating Across a Layer

To update the sensitivities, as in Eq. 7.30, we follow these steps:

$$\mathbf{dN}^m \rightarrow \mathbf{dLZ}^{m,n} \rightarrow \mathbf{dA}^n \rightarrow \mathbf{dN}^n \tag{8.29}$$

where we backpropagate across the net input function, the weight function and then the transfer function. Next, to compute the components of the gradient, we perform the following steps:

$$\mathbf{dN}^m \rightarrow db^m \tag{8.30}$$

$$\mathbf{dLZ}^{m,n} \rightarrow \mathbf{dLW}^{m,n} \tag{8.31}$$

where we again backpropagate across the net input function and the weight function, but in a different direction. This corresponds to the steps given in Equations 7.31 through 7.33.

We will consider each of the above backpropagation steps separately and will show the detailed steps for the summation net input function and the convolution weight function. We will consider other specific net input functions and weight functions in the Solved Problems and Exercises.

Across the summation net input function

There are two paths when backpropagating across the net input function: $\mathbf{dN}^m \rightarrow \mathbf{dLZ}^{m,n}$ and $\mathbf{dN}^m \rightarrow db^m$. We begin with $\mathbf{dN}^m \rightarrow \mathbf{dLZ}^{m,n}$. Since

$$\mathbf{N}^m = \mathbf{LZ}^{m,n} + b^m \tag{8.32}$$

or in scalar form

$$n_{i,j}^m = lz_{i,j}^{m,n} + b^m \tag{8.33}$$

we can write

$$dlz_{i,j}^{m,n} \equiv \frac{\partial F}{\partial lz_{i,j}^{m,n}} = \frac{\partial n_{i,j}^m}{\partial lz_{i,j}^{m,n}} \times \frac{\partial F}{\partial n_{i,j}^m} = 1 \times \frac{\partial F}{\partial n_{i,j}^m} \equiv dn_{i,j}^m \tag{8.34}$$

or in matrix form

$$\mathbf{dLZ}^{m,n} = \mathbf{dN}^m \tag{8.35}$$

(Note that this equation would be true, even if multiple layers were connecting into Layer m – i.e., there could be more than one Layer n, and the derivative would be the same for all of them.)

Now we consider the $\mathbf{dN}^m \rightarrow db^m$ path.

$$db^m \equiv \frac{\partial F}{\partial b^m} = \sum_{i=1}^{S_r^m} \sum_{j=1}^{S_c^m} \frac{\partial F}{\partial n_{i,j}^m} \times \frac{\partial n_{i,j}^m}{\partial b^m} = \sum_{i=1}^{S_r^m} \sum_{j=1}^{S_c^m} dn_{i,j}^m \tag{8.36}$$

In this case, when we use the chain rule, we need to consider all of the ways in which b^m affects $\mathbf{N}^m$. Since b^m affects each element of $\mathbf{N}^m$, we need to sum over all of these elements in Eq. 8.36. Notice that the resulting operation is the average pooling operation, with the kernel size being equal to the size of the feature map. In abbreviated notation, this can be written

$$db^m = \boxplus_{S_r^m, S_c^m}^{ave} \mathbf{dN}^m \tag{8.37}$$

Across the convolution weight function

There are also two backpropagation paths across the weight function: $\mathbf{dLZ}^{m,n} \rightarrow \mathbf{dA}^n$ and $\mathbf{dLZ}^{m,n} \rightarrow \mathbf{dLW}^{m,n}$. We begin with $\mathbf{dLZ}^{m,n} \rightarrow \mathbf{dA}^n$. The convolution operation with multiple kernels is given by Eq. 8.20, repeated here:

$$lz_{i,j}^{m,n,h} = \sum_{l=1}^{H^n} \sum_{u=1}^{r^m} \sum_{v=1}^{c^m} lw_{u,v}^{m,n,h,l} a_{i+u-1,j+v-1}^{n,l} \tag{8.38}$$

Therefore, we can write

$$da_{u',v'}^{n,l} \equiv \frac{\partial F}{\partial a_{u',v'}^{n,l}} = \sum_{h=1}^{H^n} \sum_{i=1}^{S_r^m} \sum_{j=1}^{S_c^m} \frac{\partial F}{\partial lz_{i,j}^{m,n,h}} \times \frac{\partial lz_{i,j}^{m,n,h}}{\partial a_{u',v'}^{n,l}} \tag{8.39}$$

From Eq. 8.38 we can show that

$$\frac{\partial lz_{i,j}^{m,n,h}}{\partial a_{u',v'}^{n,l}} = \begin{cases} lw_{u'-i+1,v'-j+1}^{m,n,h,l} & \text{if } i \leq u' \leq i + r^m - 1 \text{ and } j \leq v' \leq j + c^m - 1 \\ 0 & else \end{cases} \tag{8.40}$$

Therefore we can simplify Eq. 8.39 to

$$da_{u',v'}^{n,l} = \sum_{h=1}^{H^n} \sum_{i=u'-r^m+1}^{u'} \sum_{j=v'-c^m+1}^{v'} dlz_{i,j}^{m,n,h} lw_{u'-i+1,v'-j+1}^{m,n,h,l} \tag{8.41}$$

If we define new variables

$$i' = i + r^m - u' \tag{8.42}$$

$$j' = j + c^m - v' \tag{8.43}$$

we can rewrite Eq. 8.41 as

$$da^{n,l}_{u',v'} = \sum_{h=1}^{H^n} \sum_{i'=1}^{r^m} \sum_{j'=1}^{c^m} lw^{m,n,h,l}_{r^m-i'+1,c^m-j'+1} dlz^{m,n,h}_{i'+u'-r^m,j'+v'-c^m} \tag{8.44}$$

Compare Eq. 8.44 with Eq. 8.38. We can see that they are both performing convolution operations with the same weights. In Eq. 8.38, $\mathbf{LW}^{m,n,h,l}$ is being convolved with $\mathbf{A}^{n,l}$ to produce $\mathbf{LZ}^{m,n,h}$. In Eq. 8.44, $\mathbf{LW}^{m,n,h,l}$ is being convolved with $\mathbf{dLZ}^{m,n,h}$ to produce $\mathbf{dA}^{n,l}$. However, in Eq. 8.44 the indexing on the weight is different. Here is a comparison:

$$\begin{aligned}
\text{Forward} &\Leftrightarrow \text{Backpropagation} \\
\text{Indexes: } u, v &\Leftrightarrow \text{Indexes: } i', j' \\
lw^{m,n,h,l}_{u,v} &\Leftrightarrow lw^{m,n,h,l}_{r^m-i'+1,c^m-j'+1} \\
a^{n,l}_{i+u-1,j+v-1} &\Leftrightarrow dlz^{m,n,h}_{i'+u'-r^m,j'+v'-c^m}
\end{aligned}$$

The indexes on the weight in the backpropagation step are incrementing down, where the indices in the forward step are incrementing up. Effectively, the weight matrix has been rotated 180 degrees. We can represent the operation in matrix form as follows:

$$\mathbf{dA}^{n,l} = \sum_{l=1}^{H^n} ((rot180)\mathbf{LW}^{m,n,h,l}) \otimes \mathbf{dLZ}^{m,n,h} \tag{8.45}$$

Note that this result is similar to the process for the standard multilayer network: going forward we multiply by the weight matrix (see Eq. 3.42), and going backward we multiply by the transpose of the weight matrix (see Eq. 3.49).

Now let's consider the $\mathbf{dLZ}^{m,n} \rightarrow \mathbf{dLW}^{m,n}$ path, which computes a portion of the gradient.

$$dlw^{m,n,h,l}_{u,v} \equiv \frac{\partial F}{\partial lw^{m,n,h,l}_{u,v}} = \sum_{i=1}^{S^m_r} \sum_{j=1}^{S^m_c} \frac{\partial F}{\partial lz^{m,n,h}_{i,j}} \times \frac{\partial lz^{m,n,h}_{i,j}}{\partial lw^{m,n,h,l}_{u,v}} \tag{8.46}$$

From Eq. 8.38, we can see that

$$dlw^{m,n,h,l}_{u,v} = \sum_{i=1}^{S^m_r} \sum_{j=1}^{S^m_c} dlz^{m,n,h}_{i,j} \times a^{n,l}_{i+u-1,j+v-1} \tag{8.47}$$

This is a convolution of $\mathbf{dLZ}^{m,n,h}$ with $\mathbf{A}^{n,l}$. In abbreviated notation, we can write

$$\mathbf{dLW}^{m,n,h,l} = \mathbf{dLZ}^{m,n,h} \otimes \mathbf{A}^{n,l} \tag{8.48}$$

Across Activation Function

The final step of backpropagating across a layer is crossing the activation function – $\mathbf{dA}^n \rightarrow \mathbf{dN}^n$.

$$dn_{i,j}^{n,h} \equiv \frac{\partial F}{\partial n_{i,j}^{n,h}} = \frac{\partial a_{i,j}^{n,h}}{\partial n_{i,j}^{n,h}} \times \frac{\partial F}{\partial a_{i,j}^{n,h}} \tag{8.49}$$

By definition,

$$\frac{\partial F}{\partial a_{i,j}^{n,h}} \equiv da_{i,j}^{n,h} \tag{8.50}$$

and

$$\frac{\partial a_{i,j}^{n,h}}{\partial n_{i,j}^{n,h}} = \dot{f}^{n,h}(n_{i,j}^{n,h}) \tag{8.51}$$

Therefore, we can write

$$dn_{i,j}^{n,h} = \dot{f}^{n,h}(n_{i,j}^{n,h}) \times da_{i,j}^{n,h} \tag{8.52}$$

In abbreviated form, this is

$$\mathbf{dN}^{n,h} = \dot{\mathbf{F}}^{n,h} \circ \mathbf{dA}^{n,h} \tag{8.53}$$

where $\circ$ is the Hadamard product, which is an element by element matrix multiplication.

Epilogue

The convolution operation is ubiquitous in science, engineering and mathematics, and it has been in use for centuries. For image processing, it can implement any linear, position invariant transformation. It has been used in this way for template matching (matched filtering) in image processing since the 1950s. The trainable, hierachical convolution network developed in the 1980s by Yann LeCun has become, with the advent of modern computation resources, one of the most popular neural network architectures for image classification, segmentation and object detection.

These convolution networks are trained using the same gradient-based algorithms that we use for multilayer (fully connected) networks, and the gradient is computed using the same backpropagation algorithm. The difference is only in the specifics of propagating across more general net input functions, weight functions and transfer functions.

One difficulty in using any deep learning model, like a convolution network, is explainablity. The model is something of a black box, in that it does not explain why it made a certain decision. To develop confidence in the model, it can be very helpful to be able to explain why a decision is made. In the next chapter we will discuss neural network explainability.

Further Reading

[Rosenfeld, 1969] Azriel Rosenfeld. Picture processing by computer. *ACM Computing Surveys (CSUR)*, 1(3):147–176, 1969

An early survey of digital image processing. Covers convolution, cross-correlation and template matching (matched filtering). Gives a nice review of the classical methods for pattern recognition.

[Duda et al., 1973] Richard O Duda, Peter E Hart, et al. *Pattern classification and scene analysis*, volume 3. Wiley New York, 1973

This was one of the preeminent textbooks on pattern recognition during the 1970s and 1980s. It covers classical methods of pattern recognition, including template matching. An updated version of the book, titled just Pattern Classification, replaces the classical sections with neural network material. This demonstrates how the field has changed since the introduction of CNNs.

[Hubel and Wiesel, 1962] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962

This paper shows that neurons in the early visual cortex are organized in a hierarchical fashion, where the first cells connected to the cat's retinas are responsible for detecting simple patterns like edges and bars, followed by later layers responding to more complex patterns by combining the earlier neuronal activities. This work inspired Fukushima and, later, LeCun to develop hierarchical convolutional neural networks.

[Fukushima, 1980] Kunihiko Fukushima. A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biol. Cybern.*, 36:193–202, 1980

In this paper Fukushima introduced the first artificial convolutional neural network. It was based on the work of Hubel and Weisel describing the receptive fields in the cat's visual cortex. The network training was somewhat specialized and problem specific. It did not use a backpropagation-based algorithm.

[LeCun et al., 1998] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86 (11):2278–2324, 1998

This article gives the best detailed description of LeCun's original LeNet CNN architectures. The networks are similar in structure to those of Fukushima, but they are trained with standard gradient-based algorithms, with the gradient computed with backpropogation.

Summary of Results

Basic Convolution Operation

$$z_{m,n} = \sum_{i=1}^{r} \sum_{j=1}^{c} w_{i,j} p_{m-i+1,n-j+1}$$

Abbreviated Notation

$$\mathbf{Z} = \mathbf{W} * \mathbf{P}$$

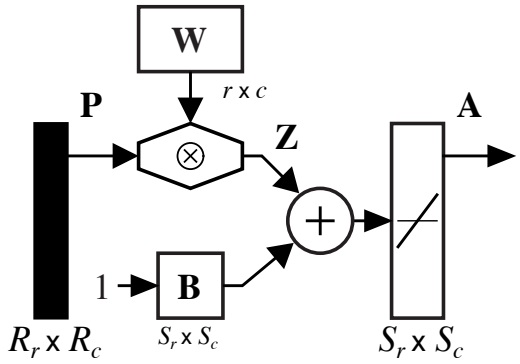
Basic Correlation Operation

$$z_{m,n} = \sum_{i=1}^{r} \sum_{j=1}^{c} w_{i,j} p_{m+i-1,n+j-1}$$

Abbreviated Notation

$$\mathbf{Z} = \mathbf{W} \otimes \mathbf{P}$$

Basic Convolution Layer



Affect of Stride and Padding on Output Size

$$S_r = (R_r + 2P^d - r)/S^t + 1$$
$$S_c = (R_c + 2P^d - c)/S^t + 1$$

Convolution with Multiple Feature Maps

$$z_{i,j}^{m,n,h} = \sum_{l=1}^{H^n} \sum_{u=1}^{r^m} \sum_{v=1}^{c^m} w_{u,v}^{m,n,h,l} a_{i+u-1,j+v-1}^{n,l}$$

Abbreviated Notation

$$\mathbf{Z}^{m,n,h} = \sum_{l=1}^{H^n} \mathbf{W}^{m,n,h,l} \otimes \mathbf{A}^{n,l}$$

Average Pooling

$$z_{i,j} = \left\{ \sum_{k=1}^{r} \sum_{l=1}^{c} v_{r(i-1)+k,c(j-1)+l} \right\} w$$

Abbreviated Notation

$$\mathbf{Z} = w \boxplus_{r,c}^{ave} \mathbf{V}$$

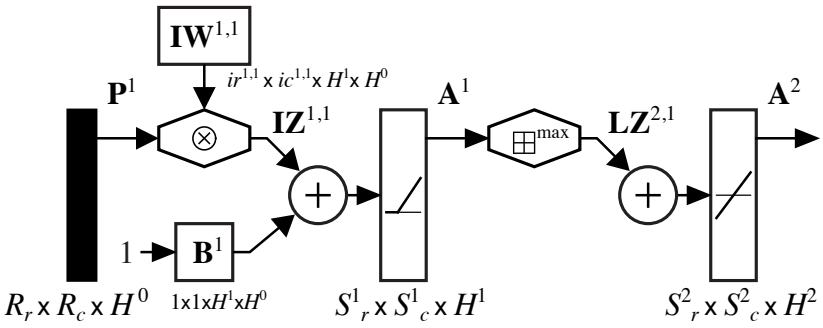
Max Pooling

$$z_{i,j} = max \left\{ v_{r(i-1)+k,c(j-1)+l} | k = 1, ..., r; l = 1, ..., c \right\}$$

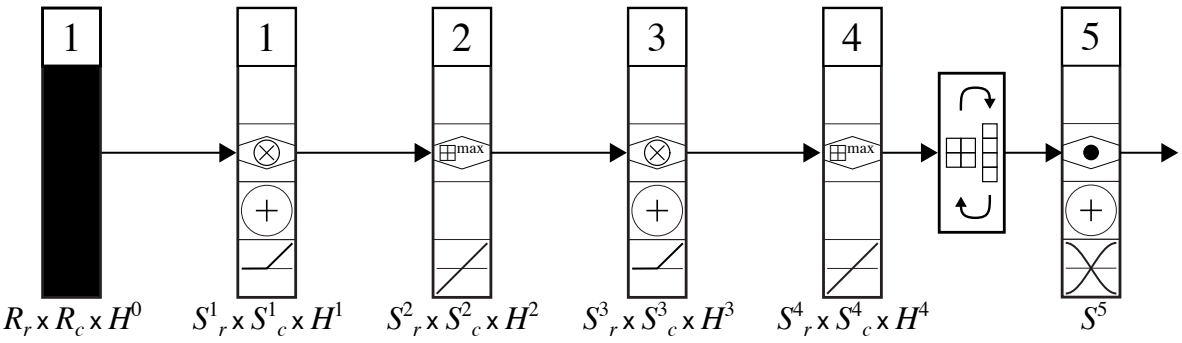
Abbreviated Notation

$$\mathbf{Z} = \boxplus_{r,c}^{max} \mathbf{V}$$

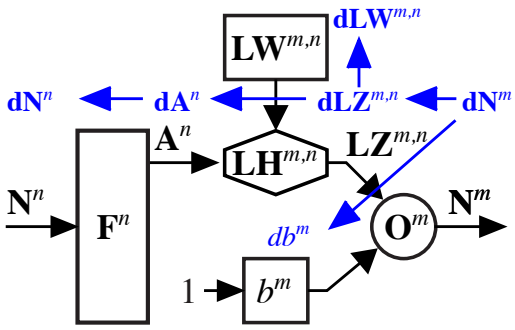
Network with Convolution and Pooling Layers



Full Convolution Network, Simplified Notation



Backpropagating Across a Layer



Backpropagation Path $\mathbf{dN}^m \rightarrow \mathbf{dLZ}^{m,n}$ (Summation)

$$dlz^{m,n}_{i,j} = dn^m_{i,j}$$

$$\mathbf{dLZ}^{m,n} = \mathbf{dN}^m$$

Backpropagation Path $\mathbf{dN}^m \rightarrow db^m$ (Summation)

$$db^m = \sum_{i=1}^{S^m_r} \sum_{j=1}^{S^m_c} dn^m_{i,j}$$

$$db^m = \boxplus^{ave}_{S^m_r, S^m_c} \mathbf{dN}^m$$

Backpropagation Path $\mathbf{dLZ}^{m,n} \rightarrow \mathbf{dA}^n$ (Convolution)

$$da^{n,l}_{u',v'} = \sum_{h=1}^{H^n} \sum_{i'=1}^{r^m} \sum_{j'=1}^{c^m} lw^{m,n,h,l}_{r^m - i' + 1, c^m - j' + 1} dlz^{m,n,h}_{i'+u'-r^m, j'+v'-c^m}$$

$$\mathbf{dA}^{n,l} = \sum_{l=1}^{H^n} ((rot180)\mathbf{LW}^{m,n,h,l}) \otimes \mathbf{dLZ}^{m,n,h}$$

Backpropagation Path $\mathbf{dLZ}^{m,n} \rightarrow \mathbf{dLW}^{m,n}$ (Convolution)

$$dlw_{u,v}^{m,n,h,l} = \sum_{i=1}^{S_r^m} \sum_{j=1}^{S_c^m} dlz_{i,j}^{m,n,h} \times a_{i+u-1,j+v-1}^{n,l}$$

$$\mathbf{dLW}^{m,n,h,l} = \mathbf{dLZ}^{m,n,h} \otimes \mathbf{A}^{n,l}$$

Backpropagation Path $\mathbf{dA}^{n} \rightarrow \mathbf{dN}^{n}$ (Activation Function)

$$dn_{i,j}^{n,h} = \dot{f}^{n,h}(n_{i,j}^{n,h}) \times da_{i,j}^{n,h}$$

$$\mathbf{dN}^{n,h} = \dot{\mathbf{F}}^{n,h} \circ \mathbf{dA}^{n,h}$$

Solved Problems

P8.1 **Demonstrate how the kernel in the convolution operation is flipped vertically and horizontally relative to the kernel in the correlation operation.**

The convolution operation can be written

$$z_{m,n} = \sum_i \sum_j w_{i,j} p_{m-i,n-j}$$

or, equivalently as

$$z_{m,n} = \sum_i \sum_j p_{i,j} w_{m-i,n-j}$$

where the summations range over all indices where both w and p are nonzero. We will use the second implementation in this demonstration.

Consider the 3x3 kernel $w_{i,j}$ shown in the following figure.

j \ i	-2	-1	0	1	2
2	0	0	7	8	9
1	0	0	4	5	6
0	0	0	1	2	3
-1	0	0	0	0	0
-2	0	0	0	0	0

Let's demonstrate how to go from $w_{i,j}$ to $w_{m-i,n-j}$, which is needed in the equation above. We will follow the steps

$$w_{i,j} \rightarrow w_{-i,j} \rightarrow w_{-i,-j} \rightarrow w_{m-i,n-j}$$

To go from $w_{i,j}$ to $w_{-i,j}$, we need to flip the kernel from left to right, as shown in the following figure.

2	9	8	7	0	0
1	6	5	4	0	0
j 0	3	2	1	0	0
-1	0	0	0	0	0
-2	0	0	0	0	0
	-2	-1	0	1	2

i

To go from $w_{-i,j}$ to $w_{-i,-j}$, we need to flip the kernel from top to bottom, as shown in the following figure.

2	0	0	0	0	0
1	0	0	0	0	0
j 0	3	2	1	0	0
-1	6	5	4	0	0
-2	9	8	7	0	0
	-2	-1	0	1	2

i

Finally, to go from $w_{-i,-j}$ to $w_{m-i,n-j}$, we need to shift the kernel by m pixels on the x-axis and n pixels on the y-axis, as shown in the following figure.

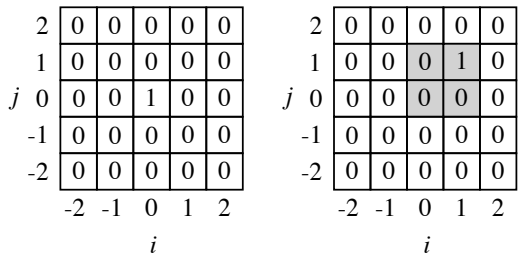
n+1	0	0	0	0	0
n	0	3	2	1	0
j *n*-1	0	6	5	4	0
n-2	0	9	8	7	0
n-3	0	0	0	0	0
	m-3	*m*-2	*m*-1	*m*	*m*+1

i

Therefore, as we calculate $z_{m,n}$ for different values of m and n, the convolution kernel is flipped vertically and horizontally and then moved across the image. At each value of m and n, an inner product is computed between the flipped and shifted kernel and the input image. This operation is equivalent to a correlation operation, if we use the flipped convolution kernel as the correlation kernel.

P8.2 **As discussed in this chapter, any linear, position invariant transformation can be represented by a convolution operation, where the convolution kernel is the impulse response of the transformation. Consider the transformation of shifting the input image one pixel to the right and one pixel up. Find the convolution kernel for this transformation by finding the impulse response.**

To find the impulse response, we need to apply the transformation to the impulse function (one-point image). In this case, that means that we shift the impulse function one pixel to the left and one pixel up. This process is shown in the following figure.



Based on this result, the convolution kernel will be

$$\mathbf{W} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

P8.3 **Find the backpropagation path $\mathbf{dLZ}^{m,n} \rightarrow \mathbf{dA}^{n}$ for the average pooling function.**

As with the convolution weight function, for the $\mathbf{dLZ}^{m,n} \rightarrow \mathbf{dA}^{n}$ path, we use Eq. 8.39. However, each $a^{n,l}_{u',v'}$ affects only one $lz^{m,n,h}_{i,j}$, since the stride of the pooling operation is equal to the size of the kernel. Therefore there is no summation required. The equation reduces to

$$da^{n,h}_{u',v'} \equiv \frac{\partial F}{\partial a^{n,h}_{u',v'}} = \frac{\partial F}{\partial lz^{m,n,h}_{i,j}} \times \frac{\partial lz^{m,n,h}_{i,j}}{\partial a^{n,h}_{u',v'}} = dlz^{m,n,h}_{i,j} \times \frac{\partial lz^{m,n,h}_{i,j}}{\partial a^{n,h}_{u',v'}}$$

It remains to find

$$\frac{\partial lz^{m,n,h}_{i,j}}{\partial a^{n,l}_{u',v'}}$$

The average pooling operation is defined in Eq. 8.22, and repeated here:

$$lz_{i,j}^{m,n,h} = \left\{ \sum_{u=1}^{r^m} \sum_{v=1}^{c^m} a_{r^m(i-1)+u,c^m(j-1)+v}^{n,h} \right\} lw^{m,n,h}$$

(Notice that for the average pooling weight function, in contrast to the convolution weight function, each input feature map contributes to one output feature map.) From this equation, we can see that

$$\frac{\partial lz_{i,j}^{m,n,h}}{\partial a_{u',v'}^{n,l}} = lw^{m,n,h} \text{ for } \begin{cases} r^m(i-1)+1 \le u' \le r^m(i-1)+r^m \\ c^m(j-1)+1 \le v' \le c^m(j-1)+c^m \end{cases}$$

From these equations we can see that each element of $\mathbf{dLZ}^{m,n,h}$ will produce $r^m \times c^m$ elements of $\mathbf{dA}^{n,h}$. We can represent this using an abbreviated notation as

$$\mathbf{dA}^{n,h} = (lw^{m,n,h}) \boxtimes_{r^m,c^m}^{ave} (\mathbf{dLZ}^{m,n,h})$$

where $\boxtimes_{j,k}^{ave}\mathbf{A}$ takes each element of the matrix $\mathbf{A}$ and expands to j rows and k columns (reverse of $\boxplus_{j,k}^{ave}\mathbf{A}$).

P8.4 Find the backpropagation path $\mathbf{dLZ}^{m,n} \rightarrow \mathbf{dLW}^{m,n}$ for the average pooling function.

The $\mathbf{dLZ}^{m,n} \rightarrow \mathbf{dLW}^{m,n}$ backpropagation path computes the portion of the gradient for the weight $lw^{m,n,h}$. We can use Eq. 8.46, but it can be simplified, since the weight is a scalar, and each feature map operates separately.

$$dlw^{m,n,h} \equiv \frac{\partial F}{\partial lw^{m,n,h}} = \sum_{i=1}^{S_r^m} \sum_{j=1}^{S_c^m} \frac{\partial F}{\partial lz_{i,j}^{m,n,h}} \times \frac{\partial lz_{i,j}^{m,n,h}}{\partial lw^{m,n,h}}$$

From the average pooling equation we can convert this to

$$dlw^{m,n,h} = \sum_{i=1}^{S_r^m} \sum_{j=1}^{S_c^m} dlz_{i,j}^{m,n,h} \left\{ \sum_{u=1}^{r^m} \sum_{v=1}^{c^m} a_{r^m(i-1)+u,c^m(j-1)+v}^{n,h} \right\}$$

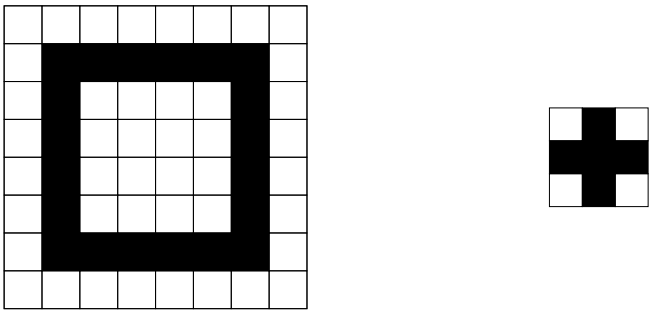
In abbreviated notation, this can be written

$$dlw^{m,n,h} = \boxplus_{S_r^m,S_c^m}^{ave} (\mathbf{dLZ}^{m,n,h} \circ (\boxplus_{r^m,c^m}^{ave} \mathbf{A}^{n,h}))$$

First it performs a reduction operation that produces a matrix of size S_r^m by S_c^m, and then, after a Hadamard matrix multiplication, it performs another reduction to produce the scalar gradient.

Exercises

E8.1 Consider the 8x8 input image below and the corresponding 3x3 kernel. The black squares correspond to values of 1, and the white squares correspond to values of zero.



i. Manually perform a convolution of the kernel on the image, with a stride of 1 and a zero padding of 1.

ii. What types of objects is this kernel good at detecting?

E8.2 For your output from Exercise E8.1, manually perform a 2x2 max pooling operation, with a stride of 2.

E8.3 Design a minimal set of convolution kernels that could be useful in identifying characters from the following font family.

E8.4 Demonstrate that the convolution kernel found in Solved Problem P8.2 does perform the required shifting operation by applying it manually to the image below.

0	0	0	0	0
0	0	0	0	0
0	0	1	0	0
0	1	1	0	0
0	0	0	0	0

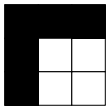
E8.5 We want to perform a linear, position invariant transformation, which is to average adjacent pixels in an input image to produce the pixels in an output image.

i. Using the concepts from Solved Problem P8.2, find the convolution kernel that will perform this transformation.

ii. Test you kernel on the input image in Exercise E8.1.

iii. What is the corresponding correlation kernel that will perform the same operation? Use the ideas from Solved Problem P8.1 to find it.

E8.6 The objective of this exercise is to investigate the difference between convolution and correlation. Consider the following 3x3 convolution kernel.



i. Manually perform a convolution (not correlation) of this kernel with the image in Exercise E8.1, with a stride of 1 and no padding.

ii. Perform a correlation with the same kernel, and compare the results with part i.

iii. Find the appropriate correlation kernel that will produce the same output that you obtained in part i. In other words, find the kernel that will produce the same output when used for correlation that the original kernel produces when used for convolution. (See Solved Problem P8.1 for a discussion of the relationship between convolution and correlation kernels.)

iv. Perform a correlation with your new kernel, and verify that you obtain the same output as in part i.

E8.7 Suppose that we have a 15x15 input image. This is input to a convolution layer with one feature map.

i. If the convolution kernel size is 3x3, the stride is $S^t = 2$ and the zero padding is $P^d = 1$, what is the size of the output feature map?

ii. If the convolution kernel size is 3x3 and the stride is $S^t = 1$, what does the padding have to be so that the size of the output feature map is the same as the size of the input image?

E8.8 Find the backpropagation path $\mathbf{dLZ}^{m,n} \rightarrow \mathbf{dA}^n$ for the max pooling operation.

E8.9 Consider a layer for which the net input function is the product between a scalar bias and the output of the weight function, instead of the standard summation. Find the two backpropagation paths across this net input function: $\mathbf{dN}^m \rightarrow \mathbf{dLZ}^{m,n}$ and $\mathbf{dN}^m \rightarrow db^m$

E8.10 Using the kernels in Figure 8.5, design a convolution network to distinguish between squares and diamonds. The network should have a convolution layer, a max pooling layer and a fully connected layer with a softmax activation. The input to the network will be a 15x15 image, like the images in Figure 8.4. Specify all of the weights and biases in the network. Remember that the network is to be designed, not trained.