# Image Processing and Recognition

Dr. Călin-Adrian POPA

## Lecture 5

November 1st, 2022

- for image data, each example consists of a two-dimensional *grid of pixels*
- depending on whether we are handling black-and-white or color images, each pixel location might be associated with either one or multiple numerical values, respectively
- one way of dealing with this rich structure is to simply discard each image's spatial structure by flattening them into one-dimensional vectors, and feeding them through a fully-connected MLP, which is deeply unsatisfying
- because these networks are invariant to the order of the features, we could get similar results regardless of whether we preserve an order corresponding to the spatial structure of the pixels, or if we permute the columns of our design matrix before fitting the MLP's parameters
- preferably, we would leverage our prior knowledge that nearby pixels are typically related to each other, to build efficient models for learning from image data

# 3 Convolutional neural networks

- in this chapter, we introduce *convolutional neural networks* (*CNNs*), a powerful family of neural networks that are designed for precisely this purpose
- CNN-based architectures are now ubiquitous in the field of computer vision, and have become so dominant that hardly anyone today would develop a commercial application or enter a competition related to image recognition, object detection, or semantic segmentation, without building off of this approach

- modern CNNs owe their design to inspirations from biology, group theory, and much experimentation
- in addition to their sample efficiency in achieving accurate models, CNNs tend to be computationally efficient, both because they require fewer parameters than fully-connected architectures, and because convolutions are easy to parallelize across GPU cores
- consequently, CNNs are often applied whenever possible, and increasingly they have emerged as good competitors even on tasks with a one-dimensional sequence structure, such as audio, text, and time series analysis, where recurrent neural networks are conventionally used
- some clever adaptations of CNNs have also brought applications on graph-structured data and in recommender systems

- first, we will walk through the basic operations that comprise the backbone of all convolutional networks
- these include the convolutional layers themselves, details including padding and stride, the pooling layers used to aggregate information across adjacent spatial regions, the use of multiple channels at each layer, and a careful discussion of the structure of modern architectures
- we will discuss the example of LeNet, the first convolutional network successfully deployed, long before the rise of modern deep learning
- then, we will dive into full details of some popular and comparatively recent CNN architectures, whose designs represent most of the techniques commonly used today

# 3.1 From fully-connected layers to convolutions

- to this day, the models that we have discussed so far remain appropriate options when we are dealing with tabular data
- by tabular, we mean that the data consist of rows corresponding to examples and columns corresponding to features
- with tabular data, we might anticipate that the patterns we seek could involve interactions among the features, but we do not assume any structure *a priori* concerning how the features interact
- sometimes, we truly lack knowledge to guide the construction of better architectures; in these cases, an MLP may be the best that we can do
- however, for high-dimensional perceptual data, such structure-less networks may not be suited

# 3.1 From fully-connected layers to convolutions

- for instance, suppose we want to distinguish cats from dogs
- say that we do a thorough job in data collection, collecting an annotated dataset of one-megapixel photographs
- this means that each input to the network has one million dimensions
- according to our discussions of parameterization cost of fully-connected layers, even an aggressive reduction to one thousand hidden dimensions would require a fully-connected layer characterized by $10^6 \times 10^3 = 10^9$ parameters
- learning the parameters of this network may turn out to be infeasible

# 3.1 From fully-connected layers to convolutions

- we might object to this argument on the basis that one megapixel resolution may not be necessary
- however, while we might be able to get away with one hundred thousand pixels, our hidden layer of size 1000 grossly underestimates the number of hidden units that it takes to learn good representations of images, so a practical system will still require billions of parameters
- moreover, learning a classifier by fitting so many parameters might require collecting an enormous dataset
- and yet, today, both humans and computers are able to distinguish cats from dogs quite well, seemingly contradicting these intuitions
- that is because images exhibit rich structure that can be exploited by humans and machine learning models alike
- convolutional neural networks (CNNs) are one creative way that machine learning has embraced for exploiting some of the known structure in natural images

# 3.1 From fully-connected layers to convolutions

- imagine that we want to detect an object in an image
- it seems reasonable that whatever method we use to recognize objects should not be overly concerned with the precise location of the object in the image; ideally, our system should exploit this knowledge
- we can draw some inspiration here from the children's game "Where's Waldo" (depicted in Figure 1)
- the game consists of a number of chaotic scenes with many activities

# 3.1 From fully-connected layers to convolutions

- Waldo shows up somewhere in each, typically in some unlikely location; the player's goal is to locate him
- despite his characteristic outfit, this can be surprisingly difficult, due to the large number of distractions
- however, *what Waldo looks like* does not depend upon *where Waldo is located*
- we could sweep the image with a Waldo detector that could assign a score to each patch, indicating the likelihood that the patch contains Waldo
- CNNs systematize this idea of *spatial invariance*, exploiting it to learn useful representations with fewer parameters

Figure 1: An image of the "Where's Waldo" game.

# 3.1 From fully-connected layers to convolutions

- we can now make these intuitions more concrete by enumerating a few desiderata to guide our design of a neural network architecture suitable for computer vision:
  1. In the earliest layers, our network should respond similarly to the same patch, regardless of where it appears in the image. This principle is called *translation invariance*.
  2. The earliest layers of the network should focus on local regions, without regard for the contents of the image in distant regions. This is the *locality* principle. Eventually, these local representations can be *aggregated* to make predictions at the whole image level.
- let us see how this translates into mathematics; first, we need to discuss the notion of tensors
- just as vectors generalize scalars, and matrices generalize vectors, we can build data structures with even more axes
- *tensors* give us a generic way of describing *n*-dimensional arrays with an arbitrary number of axes
- vectors, for example, are first-order tensors, and matrices are second-order tensors
- tensors are denoted with plain capital letters (e.g., X, Y, and Z) and their indexing mechanism (e.g., $x_{ijk}$ and $[\mathsf{X}]_{i,j,k}$) is similar to that of matrices

# 3.1 From fully-connected layers to convolutions

- to start off, we can consider an MLP with two-dimensional images $\mathbf{X}$ as inputs and their immediate hidden representations $\mathbf{H}$ similarly represented as matrices, where both $\mathbf{X}$ and $\mathbf{H}$ have the same shape
- we now conceive of not only the inputs, but also the hidden representations as possessing spatial structure
- let $[\mathbf{X}]_{i,j}$ and $[\mathbf{H}]_{i,j}$ denote the pixel at location $(i, j)$ in the input image and hidden representation, respectively
- consequently, to have each of the hidden units receive input from each of the input pixels, we would switch from using weight matrices (as we did previously in MLPs) to representing our parameters as fourth-order weight tensors W

- suppose that **U** contains biases, we could formally express the fully-connected layer as:

$$\begin{aligned}[\mathbf{H}]_{i,j} &= [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathrm{W}]_{i,j,k,l}[\mathbf{X}]_{k,l} \\ &= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathrm{V}]_{i,j,a,b}[\mathbf{X}]_{i+a,j+b},\end{aligned}$$

  where the switch from W to V is entirely cosmetic for now, since there is a one-to-one correspondence between coefficients in both fourth-order tensors
- we simply re-index the subscripts $(k, l)$, such that $k = i + a$ and $l = j + b$; in other words, we set $[\mathrm{V}]_{i,j,a,b} = [\mathrm{W}]_{i,j,k,l}$
- the indices $a$ and $b$ run over both positive and negative offsets, covering the entire image
- for any given location $(i, j)$ in the hidden representation $[\mathbf{H}]_{i,j}$, we compute its value by summing over pixels in **X**, centered around $(i, j)$, and weighted by $[\mathrm{V}]_{i,j,a,b}$

- now, let us invoke the first principle established above: *translation invariance*
- this implies that a shift in the input **X** should simply lead to a shift in the hidden representation **H**
- this is only possible if V and **U** do not actually depend on $(i, j)$, i.e., we have $[\mathrm{V}]_{i,j,a,b} = [\mathbf{V}]_{a,b}$ and **U** is a constant, say $u$
- as a result, we can simplify the definition for **H**:

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}.$$

- this is a *convolution*: we are effectively weighting pixels at $(i + a, j + b)$, in the vicinity of location $(i, j)$, with coefficients $[\mathbf{V}]_{a,b}$ to obtain the value $[\mathbf{H}]_{i,j}$
- note that $[\mathbf{V}]_{a,b}$ needs many fewer coefficients than $[\mathrm{V}]_{i,j,a,b}$, since it no longer depends on the location within the image, which is a significant progress

- now, let us invoke the second principle: *locality*
- as motivated above, we believe that we should not have to look very far away from location $(i, j)$ in order to gather relevant information to assess what is going on at $[\mathbf{H}]_{i,j}$
- this means that, outside some range $|a| > \Delta$ or $|b| > \Delta$, we should set $[\mathbf{V}]_{a,b} = 0$
- equivalently, we can rewrite $[\mathbf{H}]_{i,j}$ as:

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{1}$$

# 3.1 From fully-connected layers to convolutions

- note that (1), in a nutshell, is a *convolutional layer*
- *convolutional neural networks* (*CNNs*) are a special family of neural networks that contain convolutional layers
- in the deep learning research community, **V** is referred to as a *convolution kernel*, a *filter*, or simply the layer's *weights*, which are often learnable parameters
- when the local region is small, the difference as compared with a fully-connected network can be dramatic
- while, previously, we might have required billions of parameters to represent just a single layer in an image-processing network, we now typically need just a few hundred, without altering the dimensionality of either the inputs or the hidden representations
- the price paid for this drastic reduction in parameters is that our features are now translation invariant and that our layer can only incorporate local information, when determining the value of each hidden activation

- before going further, we should briefly review why the above operation is called a *convolution*
- in mathematics, the convolution between two functions, say $f, g : \mathbb{R}^d \to \mathbb{R}$ is defined as:

$$(f * g)(\boldsymbol{x}) = \int f(\boldsymbol{z}) g(\boldsymbol{x} - \boldsymbol{z}) d\boldsymbol{z}.$$

- that is, we measure the *overlap* between $f$ and $g$ when one function is "flipped" and shifted by $\boldsymbol{x}$
- whenever we have discrete objects, the integral turns into a sum
- for instance, for vectors from the set of square summable infinite dimensional vectors with index running over $\mathbb{Z}$, we obtain the following definition:

$$(f * g)(i) = \sum_a f(a) g(i - a).$$

- for two-dimensional tensors, we have a corresponding sum with indices $(a, b)$ for $f$ and $(i - a, j - b)$ for $g$, respectively:

$$(f * g)(i, j) = \sum_a \sum_b f(a, b) g(i - a, j - b). \tag{2}$$

- this looks similar to (1), with one major difference
- rather than using $(i + a, j + b)$, we are using the difference instead
- note, though, that this distinction is mostly cosmetic, since we can always match the notation between (1) and (2)
- our original definition in (1) more properly describes a *cross-correlation*; we will come back to this in the following section

# 3.1 From fully-connected layers to convolutions

- returning to our Waldo detector, let us see what this looks like
- the convolutional layer picks windows of a given size and weighs intensities according to the filter **V**, as demonstrated in Figure 2
- we might aim to learn a model so that wherever the “Waldoness” is highest, we should find a peak in the hidden layer representations

Figure 2: Detecting Waldo.

# 3.1 From fully-connected layers to convolutions

- there is just one problem with this approach; so far, we ignored that images consist of 3 channels: red, green, and blue
- in reality, images are not two-dimensional objects but rather third-order tensors, characterized by a height, width, and channel, e.g., with shape $1024 \times 1024 \times 3$ pixels
- while the first two of these axes concern spatial relationships, the third can be regarded as assigning a multidimensional representation to each pixel location
- we thus index X as $[\mathsf{X}]_{i,j,k}$
- the convolutional filter has to adapt accordingly; instead of $[\mathbf{V}]_{a,b}$, we now have $[\mathsf{V}]_{a,b,c}$

# 3.1 From fully-connected layers to convolutions

- moreover, just as our input consists of a third-order tensor, it turns out to be a good idea to similarly formulate our hidden representations as third-order tensors H
- in other words, rather than just having a single hidden representation corresponding to each spatial location, we want an *entire vector* of *hidden representations* corresponding to each spatial location
- we could think of the hidden representations as comprising a number of two-dimensional grids, stacked on top of each other
- as in the inputs, these are sometimes called *channels*
- they are also sometimes called *feature maps*, as each provides a spatialized set of learned features to the subsequent layer
- intuitively, we might imagine that, at lower layers that are closer to inputs, some channels could become specialized to recognize edges, while others could recognize textures

- to support multiple channels in both inputs (X) and hidden representations (H), we can add a fourth coordinate to V: $[\mathrm{V}]_{a,b,c,d}$
- correspondingly, the bias $u$ becomes a vector $\mathbf{u}$
- putting everything together, we have:

$$[\mathrm{H}]_{i,j,d} = [\mathbf{u}]_d + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} [\mathrm{V}]_{a,b,c,d} [\mathrm{X}]_{i+a,j+b,c}, \tag{3}$$

  where $d$ indexes the *output channels* in the hidden representations H
- the subsequent convolutional layer will go on to take a third-order tensor, H, as the input
- being more general, (3) is the definition of a *convolutional layer* for *multiple channels*, where V is a kernel or filter of the layer

# 3.1 From fully-connected layers to convolutions

- there are still many operations that we need to address
- for instance, we need to figure out how to combine all the hidden representations to a single output, e.g., whether there is a Waldo *anywhere* in the image
- we also need to decide how to compute things efficiently, how to combine multiple layers, appropriate activation functions, and how to make reasonable design choices to yield networks that are effective in practice
- we turn to these issues in the next sections

# 3.2 Convolutions for images

- now that we understand how convolutional layers work in theory, we are ready to see how they work in practice
- building on our motivation of convolutional neural networks as efficient architectures for exploring structure in image data, we stick with images as our running example
- recall that, strictly speaking, convolutional layers are a misnomer, since the operations they express are more accurately described as cross-correlations
- based on our descriptions of convolutional layers, in such a layer, an input tensor and a kernel tensor are combined to produce an output tensor through a cross-correlation operation

- let us ignore channels for now, and see how this works with two-dimensional data and hidden representations
- in Figure 3, the input is a two-dimensional tensor with a height of 3 and width of 3
- we mark the shape of the tensor as $3 \times 3$ or $(3, 3)$
- the height and width of the kernel are both 2
- the shape of the *kernel window* (or *convolution window*) is given by the height and width of the kernel (here, it is $2 \times 2$)

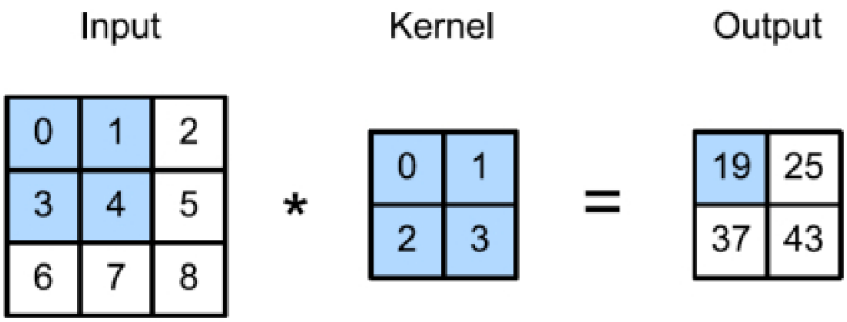


Figure 3: Two-dimensional cross-correlation operation. The shaded portions are the first output element, as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

- in the two-dimensional cross-correlation operation, we begin with the convolution window positioned at the upper-left corner of the input tensor, and slide it across the input tensor, both from left to right and top to bottom
- when the convolution window slides to a certain position, the input subtensor contained in that window and the kernel tensor are multiplied element-wise, and the resulting tensor is summed up, yielding a single scalar value
- this result gives the value of the output tensor at the corresponding location
- here, the output tensor has a height of 2 and width of 2, and the four elements are derived from the two-dimensional cross-correlation operation:

$$
\begin{aligned}
0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 &= 19, \\
1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 &= 25, \\
3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 &= 37, \\
4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 &= 43.
\end{aligned}
$$

- note that, along each axis, the output size is slightly smaller than the input size
- because the kernel has width and height greater than one, we can only properly compute the cross-correlation for locations where the kernel fits wholly within the image, the output size is given by the input size $n_h \times n_w$ minus the size of the convolution kernel $k_h \times k_w$ via:

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

- this is the case, since we need enough space to "shift" the convolution kernel across the image
- later, we will see how to keep the size unchanged by padding the image with zeros around its boundary, so that there is enough space to shift the kernel

# 3.2 Convolutions for images

- a convolutional layer cross-correlates the input and kernel, and adds a scalar bias to produce an output
- the two parameters of a convolutional layer are the *kernel* and the scalar *bias*
- when training models based on convolutional layers, we typically initialize the kernels randomly, just as we would with a fully-connected layer
- in $h \times w$ convolution or a $h \times w$ convolution kernel, the height and width of the convolution kernel are $h$ and $w$, respectively
- we also refer to a convolutional layer with a $h \times w$ convolution kernel simply as a $h \times w$ convolutional layer

# 3.2 Convolutions for images

- let us take a moment to discuss a simple application of a convolutional layer: detecting the edge of an object in an image by finding the location of the pixel change
- in this case, the kernel can be constructed as $[1, -1]$, i.e., it has a height of 1 and a width of 2
- designing an edge detector by finite differences $[1, -1]$ is good, if we know this is precisely what we are looking for
- however, as we look at larger kernels, and consider successive layers of convolutions, it might be impossible to specify precisely what each filter should be doing, manually
- thus, we need to *learn* these kernels

- recall our observation of the correspondence between the cross-correlation and convolution operations
- here, let us continue to consider two-dimensional convolutional layers
- what if such layers perform strict convolution operations as defined in (2), instead of cross-correlations?
- in order to obtain the output of the strict *convolution* operation, we only need to flip the two-dimensional kernel tensor both horizontally and vertically, and then perform the *cross-correlation* operation with the input tensor
- it is noteworthy that, since kernels are learned from data in deep learning, the outputs of convolutional layers remain unaffected, no matter if such layers perform either the strict convolution operations, or the cross-correlation operations

- to illustrate this, suppose that a convolutional layer performs *cross-correlation* and learns the kernel in Figure 3, which is denoted as the matrix **K** here
- assuming that other conditions remain unchanged, when this layer performs strict *convolution* instead, the learned kernel $\mathbf{K}'$ will be the same as **K** after $\mathbf{K}'$ is flipped both horizontally and vertically
- that is to say, when the convolutional layer performs strict *convolution* for the input in Figure 3 and $\mathbf{K}'$, the same output in Figure 3 (cross-correlation of the input and **K**) will be obtained
- in keeping with standard terminology with deep learning literature, we will continue to refer to the cross-correlation operation as a convolution even though, strictly-speaking, it is slightly different
- besides, we use the term *element* to refer to an entry (or component) of any tensor representing a layer representation or a convolution kernel

- as described before, the convolutional layer output in Figure 3 is sometimes called a *feature map*, as it can be regarded as the learned representations (features) in the spatial dimensions (e.g., width and height) to the subsequent layer
- in CNNs, for any element $x$ of some layer, its *receptive field* refers to all the elements (from all the previous layers) that may affect the calculation of $x$ during the forward propagation
- note that the receptive field may be larger than the actual size of the input

- let us continue to use Figure 3 to explain the receptive field
- given the $2 \times 2$ convolution kernel, the *receptive field* of the shaded output element (of value 19) is the four elements in the shaded portion of the input
- now, let us denote the $2 \times 2$ output as **Y** and consider a deeper CNN with an additional $2 \times 2$ convolutional layer that takes **Y** as its input, outputting a single element $z$
- in this case, the receptive field of $z$ on **Y** includes all the four elements of **Y**, while the receptive field on the input includes all the nine input elements
- thus, when any element in a feature map needs a larger receptive field to detect input features over a broader area, we can build a deeper network

- in the previous example of Figure 3, our input had both a height and width of 3 and our convolution kernel had both a height and width of 2, yielding an output representation with dimension $2 \times 2$
- as we generalized in Section 3.2, assuming that the input shape is $n_h \times n_w$ and the convolution kernel shape is $k_h \times k_w$, then the output shape will be $(n_h - k_h + 1) \times (n_w - k_w + 1)$
- therefore, the output shape of the convolutional layer is determined by the shape of the input and the shape of the convolution kernel

# 3.3 Padding and stride

- in several cases, we incorporate techniques, including *padding* and *strided convolutions*, that affect the size of the output
- as motivation, note that, since kernels generally have width and height greater than 1, after applying many successive convolutions, we tend to wind up with outputs that are considerably smaller than our input
- if we start with a $240 \times 240$ pixel image, 10 layers of $5 \times 5$ convolutions reduce the image to $200 \times 200$ pixels, slicing off 30% of the image, and with it ignoring any interesting information on the boundaries of the original image
- *padding* is the most popular tool for handling this issue
- in other cases, we may want to reduce the dimensionality drastically, e.g., if we find the original input resolution to be too big
- *strided convolutions* are a popular technique that can help in these instances

- as described above, one tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image
- since we typically use small kernels, for any given convolution, we might only lose a few pixels, but this can add up as we apply many successive convolutional layers
- one straightforward solution to this problem is to add extra pixels around the boundary of our input image, thus increasing the effective size of the image
- typically, we set the values of the extra pixels to zero
- in Figure 4, we pad a $3 \times 3$ input, increasing its size to $5 \times 5$; the corresponding output then increases to a $4 \times 4$ matrix
- the shaded portions are the first output element, as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$

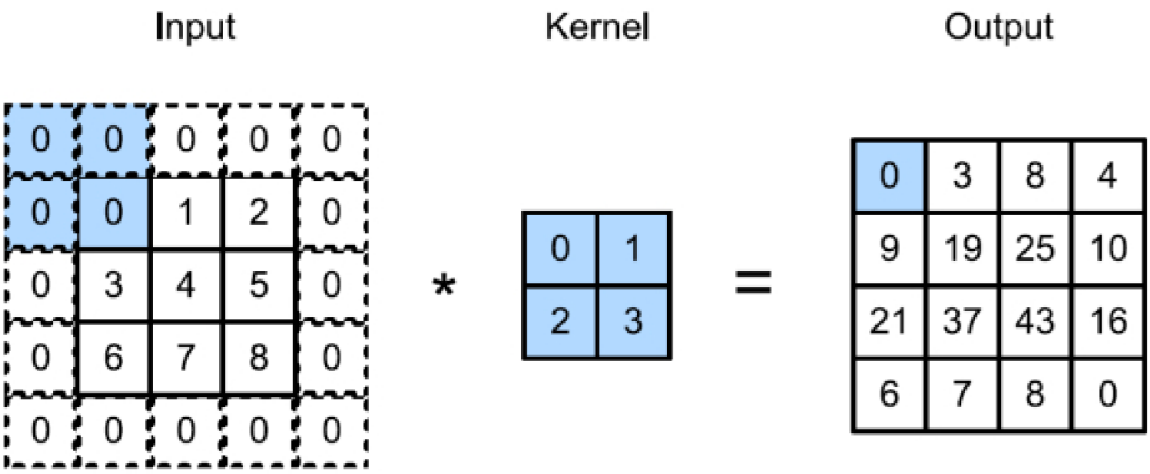


Figure 4: Two-dimensional cross-correlation with padding.

- in general, if we add a total of $p_h$ rows of padding (roughly half on top and half on bottom) and a total of $p_w$ columns of padding (roughly half on the left and half on the right), the output shape will be:

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

- this means that the height and width of the output will increase by $p_h$ and $p_w$, respectively
- in many cases, we will want to set $p_h = k_h - 1$ and $p_w = k_w - 1$ to give the input and output the same height and width
- this will make it easier to predict the output shape of each layer, when constructing the network
- assuming that $k_h$ is odd here, we will pad $p_h/2$ rows on both sides of the height
- if $k_h$ is even, one possibility is to pad $\lceil p_h/2 \rceil$ rows on the top of the input and $\lfloor p_h/2 \rfloor$ rows on the bottom
- we will proceed in the same way for the width

- CNNs commonly use convolution kernels with odd height and width values, such as 1, 3, 5, or 7
- choosing odd kernel sizes has the benefit that we can preserve the spatial dimensionality while padding with the same number of rows on top and bottom, and the same number of columns on left and right
- moreover, this practice of using odd kernels and padding to precisely preserve dimensionality offers a clear benefit
- for any two-dimensional tensor $\mathbf{X}$, when the kernel's size is odd and the number of padding rows and columns on all sides are the same, producing an output with the same height and width as the input, we know that the output $[\mathbf{Y}]_{i,j}$ is calculated by cross-correlation of the input and convolution kernel with the window centered on $[\mathbf{X}]_{i,j}$

# 3.3 Padding and stride

- when computing the cross-correlation, we start with the convolution window at the upper-left corner of the input tensor, and then slide it over all locations both down and to the right
- in previous examples, we default to sliding one element at a time
- however, sometimes, either for computational efficiency, or because we wish to downsample, we move our window more than one element at a time, skipping the intermediate locations
- we refer to the number of rows and columns traversed per slide as the *stride*
- so far, we have used strides of 1, both for height and width; sometimes, we may want to use a larger stride

- Figure 5 shows a two-dimensional cross-correlation operation with a stride of 3 vertically and 2 horizontally
- the shaded portions are the output elements, as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$, $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$
- we can see that, when the second element of the first column is outputted, the convolution window slides down three rows
- the convolution window slides two columns to the right when the second element of the first row is outputted
- when the convolution window continues to slide two columns to the right on the input, there is no output, because the input element cannot fill the window (unless we add another column of padding)

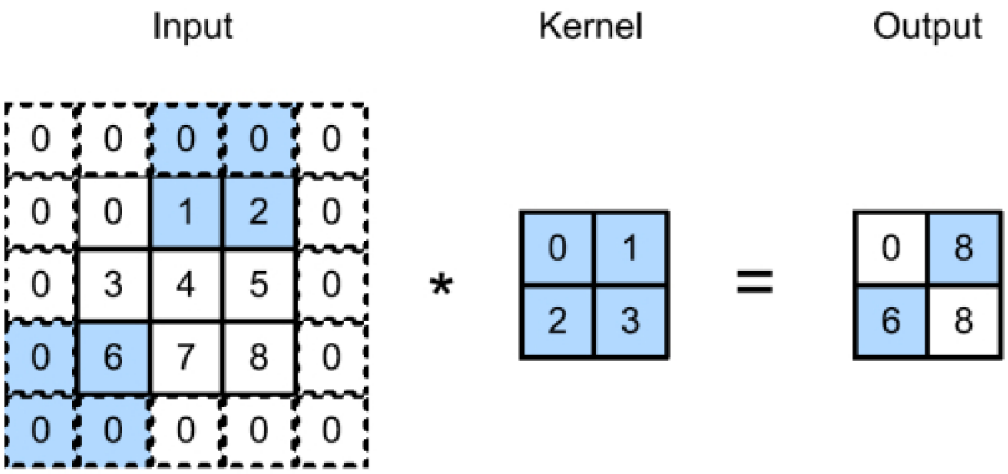


Figure 5: Cross-correlation with strides of 3 and 2 for height and width, respectively.

- in general, when the stride for the height is $s_h$ and the stride for the width is $s_w$, the output shape is:
$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor.$$
- if we set $p_h = k_h - 1$ and $p_w = k_w - 1$, then the output shape will be simplified to $\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$
- going a step further, if the input height and width are divisible by the strides on the height and width, then the output shape will be $(n_h/s_h) \times (n_w/s_w)$

# 3.3 Padding and stride

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

# 3.4 Multiple input and multiple output channels

- while we have described the multiple channels that comprise each image (e.g., color images have the standard RGB channels to indicate the amount of red, green, and blue) and convolutional layers for multiple channels in Section 3.1, until now, we simplified all of our numerical examples by working with just a single input and a single output channel
- this has allowed us to think of our inputs, convolution kernels, and outputs each as two-dimensional tensors
- when we add channels into the mix, our inputs and hidden representations both become three-dimensional tensors; for example, each RGB input image has shape $3 \times h \times w$
- we refer to this axis, with a size of 3, as the *channel dimension*
- in this section, we will take a deeper look at convolution kernels with multiple input and multiple output channels

# 3.4 Multiple input and multiple output channels

- when the input data contain multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-correlation with the input data
- assuming that the number of channels for the input data is $c_{in}$, the number of input channels of the convolution kernel also needs to be $c_{in}$
- if our convolution kernel's window shape is $k_h \times k_w$, then, when $c_{in} = 1$, we can think of our convolution kernel as just a two-dimensional tensor of shape $k_h \times k_w$
- however, when $c_{in} > 1$, we need a kernel that contains a tensor of shape $k_h \times k_w$ for every input channel
- concatenating these $c_{in}$ tensors together yields a convolution kernel of shape $c_{in} \times k_h \times k_w$

# 3.4 Multiple input and multiple output channels

- since the input and convolution kernel each have $c_{in}$ channels, we can perform a cross-correlation operation on the two-dimensional tensor of the input and the two-dimensional tensor of the convolution kernel for each channel, adding the $c_{in}$ results together (summing over the channels) to yield a two-dimensional tensor
- this is the result of a two-dimensional cross-correlation between a multi-channel input and a multi-input-channel convolution kernel
- in Figure 6, we demonstrate an example of a two-dimensional cross-correlation with two input channels
- the shaded portions are the first output element, as well as the input and kernel tensor elements used for the output computation:
  $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56.$

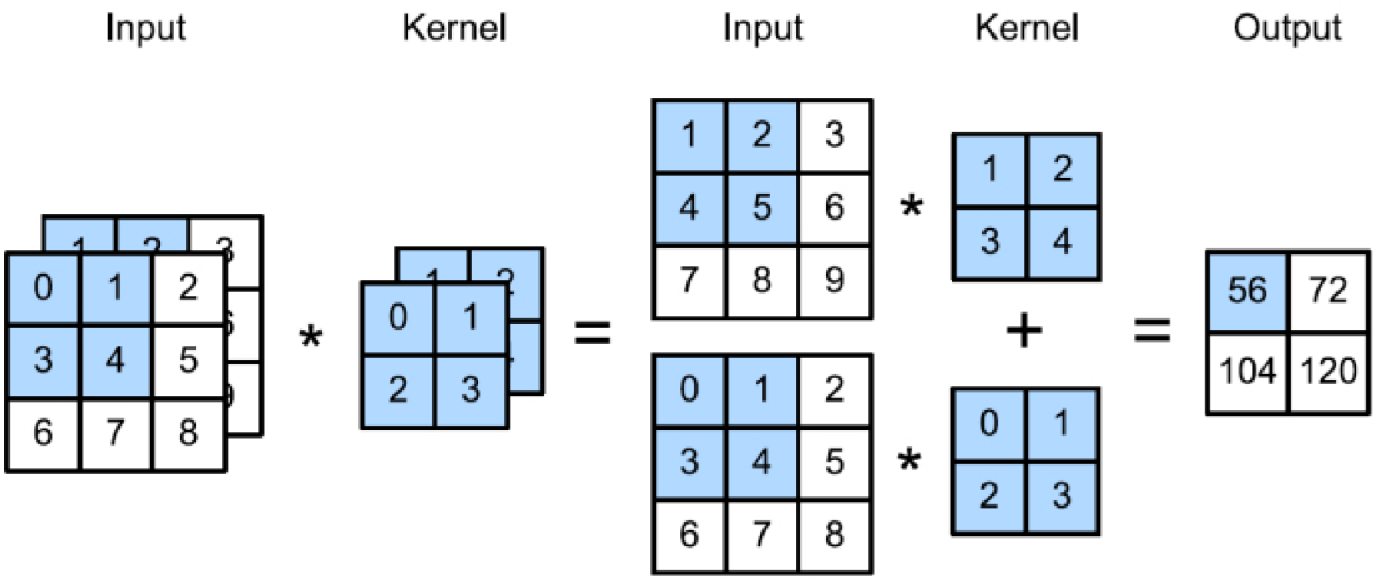


Figure 6: Cross-correlation computation with 2 input channels.

# 3.4 Multiple input and multiple output channels

- regardless of the number of input channels, so far we always ended up with one output channel
- however, as we discussed in Section 3.1, it turns out to be essential to have multiple channels at each layer
- in the most popular neural network architectures, we actually increase the channel dimension as we go higher up in the neural network, typically downsampling to trade off spatial resolution for greater *channel depth*
- intuitively, we could think of each channel as responding to some different set of features
- reality is a bit more complicated than the most naive interpretations of this intuition, since representations are not learned independently, but are rather optimized to be jointly useful
- so it may not be that a single channel learns an edge detector, but rather that some direction in channel space corresponds to detecting edges

# 3.4 Multiple input and multiple output channels

- denote by $c_{in}$ and $c_{out}$ the number of input and output channels, respectively, and let $k_h$ and $k_w$ be the height and width of the kernel
- to get an output with multiple channels, we can create a kernel tensor of shape $c_{in} \times k_h \times k_w$ for *every* output channel
- we concatenate them on the output channel dimension, so that the shape of the convolution kernel is $c_{out} \times c_{in} \times k_h \times k_w$
- in cross-correlation operations, the result on each output channel is calculated from the convolution kernel corresponding to that output channel, and takes input from all channels in the input tensor

# 3.4 Multiple input and multiple output channels

- at first, a $1 \times 1$ convolution, i.e., $k_h = k_w = 1$, does not seem to make much sense
- after all, a convolution correlates adjacent pixels; a $1 \times 1$ convolution obviously does not
- nonetheless, they are popular operations that are sometimes included in the designs of complex deep networks
- because the minimum window is used, the $1 \times 1$ convolution loses the ability of larger convolutional layers to recognize patterns consisting of interactions among adjacent elements in the height and width dimensions
- the only computation of the $1 \times 1$ convolution occurs on the channel dimension

# 3.4 Multiple input and multiple output channels

- Figure 7 shows the cross-correlation computation using the $1 \times 1$ convolution kernel with 3 input channels and 2 output channels
- note that the inputs and outputs have the same height and width
- each element in the output is derived from a linear combination of elements *at the same position* in the input image
- we could think of the $1 \times 1$ convolutional layer as constituting a fully-connected layer applied at every single pixel location to transform the $c_{in}$ corresponding input values into $c_{out}$ output values
- because this is still a convolutional layer, the weights are tied across pixel location
- thus, the $1 \times 1$ convolutional layer requires $c_{out} \times c_{in}$ weights (plus the bias)

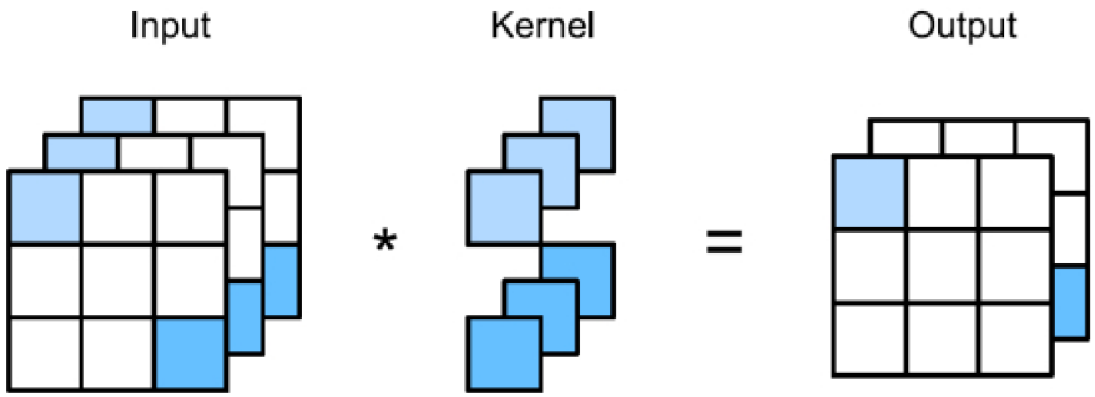


Figure 7: The cross-correlation computation uses the $1 \times 1$ convolution kernel with 3 input channels and 2 output channels. The input and output have the same height and width.

Thank you!