THE CROSS-CORRELATION OPERATION

CONVOLUTIONAL NEURAL NETWORKS 0

DATA SCIENCE IN ELECTRON MICROSCOPY

PHILIPP PELZ

2024

HTTPS://GITHUB.COM/ECLIPSE-LAB/WS24_DATASCIENCEFOREM

FROM FULLY CONNECTED LAYERS TO CONVOLUTIONS

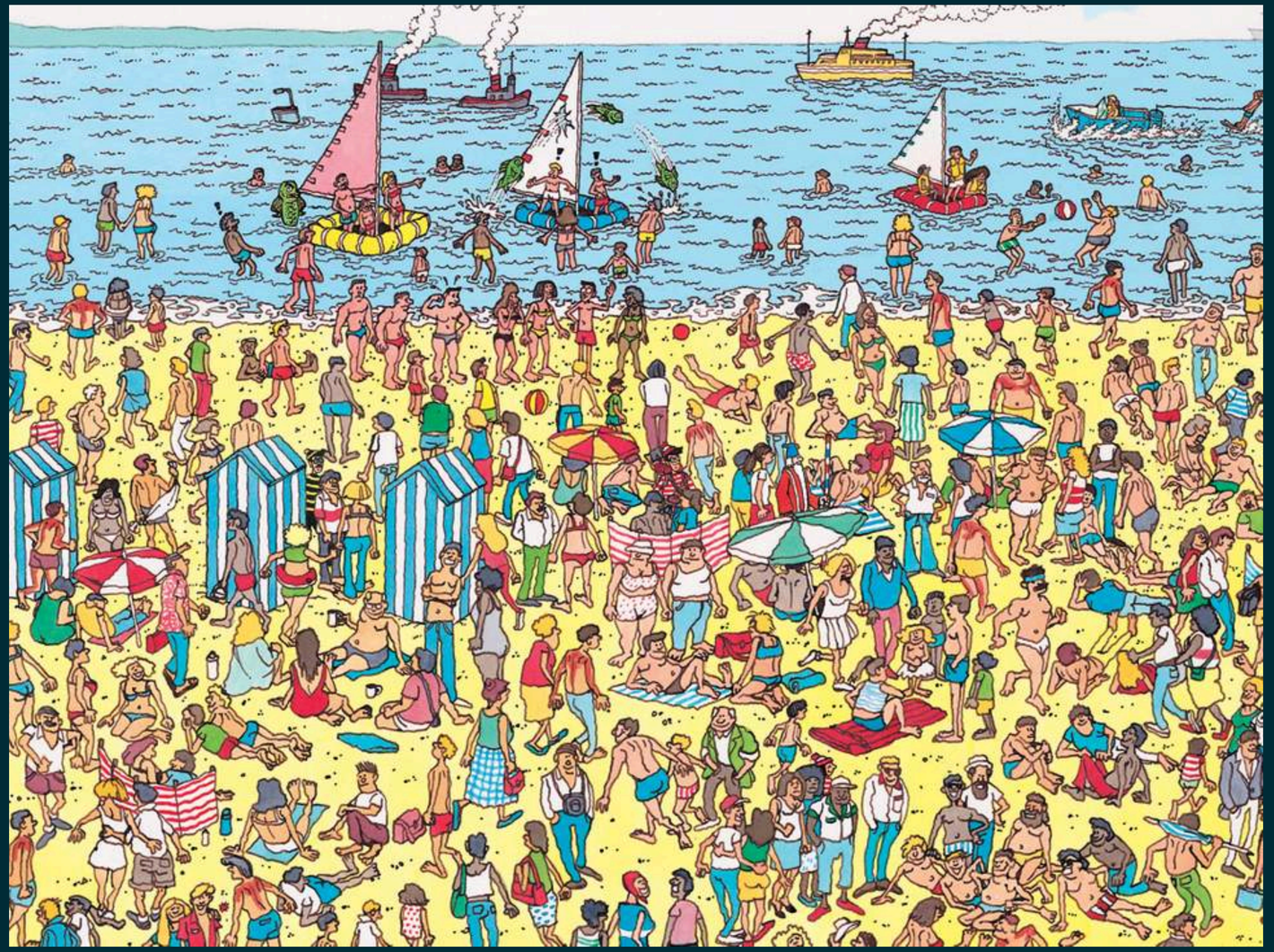
:label:`sec_why-conv`

- MLPs work well for tabular data but become impractical for high-dimensional perceptual data like images
- For a one-megapixel image, even with dimension reduction to 1000 hidden units, a fully connected layer would require $10^6 \times 10^3 = 10^9$ parameters
- Images have rich structure that can be exploited through Convolutional Neural Networks (CNNs)

Three key principles guide CNN design: 1. **Translation Invariance**: The network should respond similarly to patterns regardless of their location 2. **Locality**: Early layers should focus on local regions, not the entire image 3. **Hierarchy**: Deeper layers should capture longer-range features

These principles allow CNNs to efficiently process visual information while dramatically reducing the number of parameters compared to fully connected networks. ## Invariance

- When detecting objects in images, the recognition method should not be overly dependent on the precise location of the object
- This concept can be illustrated through the children's game "Where's Waldo" (depicted in :numref:`img_waldo`)
- *What Waldo looks like* does not depend upon *where Waldo is located*
- We could sweep the image with a Waldo detector that assigns a score to each patch, indicating the likelihood of containing Waldo
- Many object detection and segmentation algorithms are based on this approach :cite:`Long.Shelhamer.Darrell.2015`
- CNNs systematize this idea of *spatial invariance*, exploiting it to learn useful representations with fewer parameters



An image of the “Where’s Waldo” game.

- 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* (or *translation equivariance*).
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.
3. As we proceed, deeper layers should be able to capture longer-range features of the image, in a way similar to higher level vision in nature.

- Let’s see how this translates into mathematics.

CONSTRAINING THE MLP

- 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
- 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
- To have each hidden unit receive input from each input pixel, we use fourth-order weight tensors W
- With biases $\mathbf{U}$, we can express the fully connected layer as:

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

- For a 1000×1000 image mapped to a 1000×1000 hidden representation, this requires 10^{12} parameters, far beyond current computational capabilities

TRANSLATION INVARIANCE

- invoke the first principle established above: translation invariance :cite:`Zhang.ea.1988`.
- This implies that a shift in the input $\mathbf{X}$ should simply lead to a shift in the hidden representation $\mathbf{H}$.
- This is only possible if $\mathbf{V}$ and $\mathbf{U}$ do not actually depend on (i, j). As such, we have $[\mathbf{V}]_{i,j,a,b} = [\mathbf{V}]_{a,b}$ and $\mathbf{U}$ is a constant, say u.
- As a result, we can simplify the definition for $\mathbf{H}$:

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

- 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 $[\mathbf{V}]_{i,j,a,b}$ since it no longer depends on the location within the image. The number of parameters required is reduced from 10^{12} to $4 \cdot 10^6$.

LOCALITY

- Now let's invoke the second principle: locality. We believe that we should not have to look very far away from location (i, j) 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

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

:eqlabel:`eq_conv-layer`

- This reduces the number of parameters from $4 \cdot 10^6$ to $4\Delta^2$, where Δ is typically smaller than 10. We've reduced the number of parameters by another 4 orders of magnitude.
- This equation, in essence, is what we call a *convolutional layer*. *Convolutional neural networks* (CNNs) are neural networks that contain convolutional layers.

CONVOLUTIONS

- Let's briefly review why :eqref:`eq_conv-layer` is called a convolution.
- In mathematics, the *convolution* between two functions :cite:`Rudin.1973`, say $f, g : \mathbb{R}^d \to \mathbb{R}$ is defined as

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

- For discrete objects, the integral becomes a sum. For two-dimensional tensors:

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

:eqlabel:`eq_2d-conv-discrete`

CHANNELS

- Returning to Waldo detector, let's see what this looks like.
- Convolutional layer picks windows of a given size and weighs intensities according to the filter V, as demonstrated in :numref:`fig_waldo_mask`.
- Aim to learn a model so that wherever the "waldoness" is highest, we should find a peak in the hidden layer representations.

Detect Waldo.

- Images consist of 3 channels: red, green, and blue.
- Images are third-order tensors, characterized by a height, width, and channel, e.g., with shape $1024 \times 1024 \times 3$ pixels.
- While the first two 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 $[\mathsf{V}]_{a,b,c}$.
- Hidden representations are also formulated as third-order tensors H.
- These channels are sometimes called *feature maps*, as each provides a spatialized set of learned features.
- To support multiple channels in both inputs (X) and hidden representations (H), we add a fourth coordinate to V: $[\mathsf{V}]_{a,b,c,d}$.

The complete convolution can be written as:

$$[\mathsf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} [\mathsf{V}]_{a,b,c,d} [\mathsf{X}]_{i+a,j+b,c},$$

where d indexes the output channels in the hidden representations H.

SUMMARY AND DISCUSSION

- derived the structure of convolutional neural networks from first principles.
- Translation invariance in images implies that all patches of an image will be treated in the same manner.
- Locality means that only a small neighborhood of pixels will be used to compute the corresponding hidden representations. Some of the earliest references to CNNs are in the form of the Neocognitron :cite:`Fukushima.1982`.
- Adding channels allowed us to bring back some of the complexity that was lost due to the restrictions imposed on the convolutional kernel by locality and translation invariance.
- Many images have tens to hundreds of channels, generating hyperspectral images instead. They report data on many different wavelengths. In the following we will see how to use convolutions effectively to manipulate the dimensionality of the images they operate on, how to move from location-based to channel-based representations and how to deal with large numbers of categories efficiently.

EXERCISES

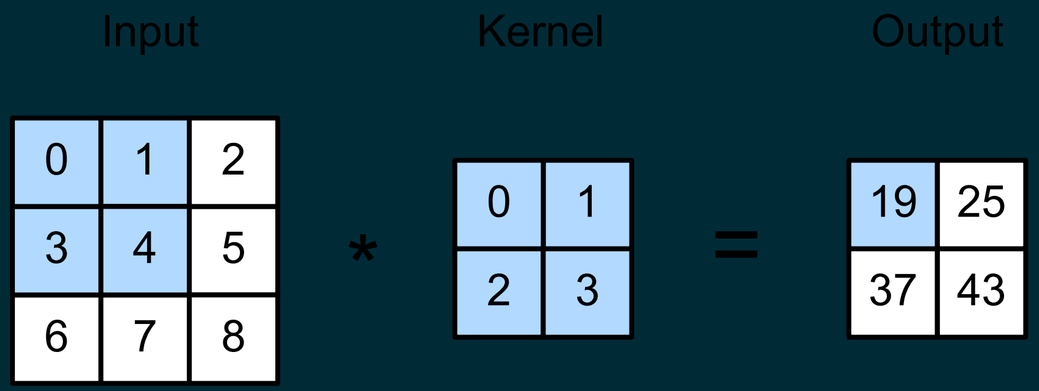
1. Assume that the size of the convolution kernel is $\Delta = 0$. Show that in this case the convolution kernel implements an MLP independently for each set of channels. This leads to the Network in Network architectures :cite:`Lin.Chen.Yan.2013`.
2. Audio data is often represented as a one-dimensional sequence.
    1. When might you want to impose locality and translation invariance for audio?
    2. Derive the convolution operations for audio.
    3. Can you treat audio using the same tools as computer vision? Hint: use the spectrogram.
3. Why might translation invariance not be a good idea after all? Give an example.
4. Do you think that convolutional layers might also be applicable for text data? Which problems might you encounter with language?
5. What happens with convolutions when an object is at the boundary of an image.
6. Prove that the convolution is symmetric, i.e., $f * g = g * f$.

CONVOLUTIONS FOR IMAGES

- 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.

```
from d2l import torch as d2l
import torch
from torch import nn
```

- 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 :numref:`sec_why-conv`, in such a layer, an input tensor and a kernel tensor are combined to produce an output tensor through a cross-correlation operation.



Two-dimensional cross-correlation operation

- 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 elementwise and the resulting tensor is summed up yielding a single scalar value.

The output tensor has a height of 2 and a width of 2 and the four elements are derived from the two-dimensional cross-correlation operation:

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

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:

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

```python
def corr2d(X, K):  #@save
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = d2l.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
```

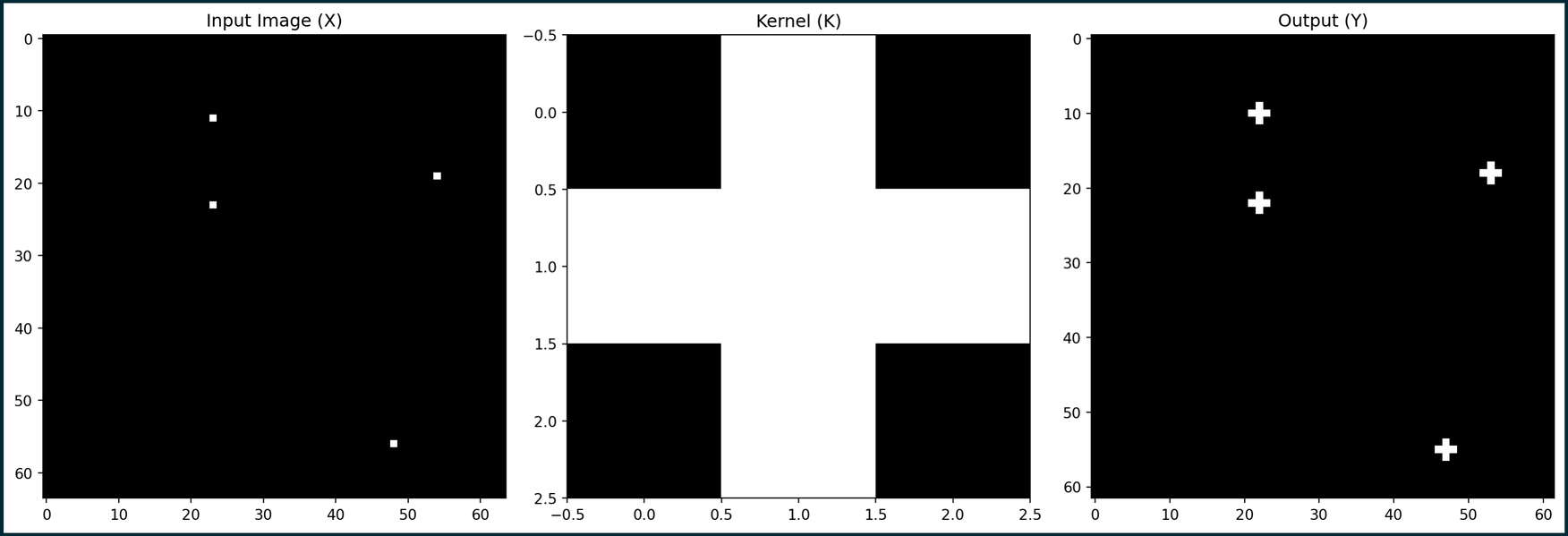
```python
# Create a 256x256 sparse image with a few 1s
X = torch.zeros((64, 64))
# Add some 1s at specific positions (this is just an example pattern)
X[11, 23] = 1.0
X[23, 23] = 1.0
X[56, 48] = 1.0
X[19, 54] = 1.0

# Create a 3x3 cross-shaped kernel
K = torch.tensor([[0.0, 1.0, 0.0],
                  [1.0, 1.0, 1.0],
                  [0.0, 1.0, 0.0]])

# Apply the correlation
Y = corr2d(X, K)

import matplotlib.pyplot as plt

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5))
```



CONVOLUTIONAL LAYERS

- 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.

```python
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(torch.rand(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

- 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.

OBJECT EDGE DETECTION IN IMAGES

- Let's examine a simple application of a convolutional layer: detecting the edge of an object in an image by finding the location of pixel changes.
- First, we construct an "image" of 6×8 pixels. The middle four columns are black (0) and the rest are white (1).

```
X = d2l.ones((6, 8))
X[:, 2:6] = 0
X
```

```
tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.]])
```

- Next we construct a kernel K with a height of 1 and a width of 2. When performing the cross-correlation operation, if the horizontally adjacent elements are the same, the output is 0. Otherwise, the output is non-zero.
- This kernel is a special case of a finite difference operator. At location (i, j) it computes $x_{i,j} - x_{(i+1),j}$, approximating the first derivative in the horizontal direction.

```
K = d2l.tensor([[1.0, -1.0]])
```

- As we can see below, [**we detect 1 for the edge from white to black and -1 for the edge from black to white.**] All other outputs take value 0.

```
Y = corr2d(X, K)
Y
```

```
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

- We can now apply the kernel to the transposed image. As expected, it vanishes. [**The kernel K only detects vertical edges.**]

```
corr2d(d2l.transpose(X), K)
```

```
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
```

LEARNING A KERNEL

- Now let’s see whether we can [**learn the kernel that generated Y from X**] by looking at the input–output pairs only.
- We first construct a convolutional layer and initialize its kernel as a random tensor. Next, in each iteration, we will use the squared error to compare Y with the output of the convolutional layer.

```
## Construct a two-dimensional convolutional layer with 1 output channel and a
## kernel of shape (1, 2). For the sake of simplicity, we ignore the bias here
conv2d = nn.LazyConv2d(1, kernel_size=(1, 2), bias=False)

## The two-dimensional convolutional layer uses four-dimensional input and
## output in the format of (example, channel, height, width), where the batch
## size (number of examples in the batch) and the number of channels are both 1
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))
lr = 3e-2  ## Learning rate

for i in range(10):
    Y_hat = conv2d(X)
    l = (Y_hat - Y) ** 2
    conv2d.zero_grad()
    l.sum().backward()
    ## Update the kernel
    conv2d.weight.data[:] -= lr * conv2d.weight.grad
    if (i + 1) % 2 == 0:
```

```
epoch 2, loss 1.746
epoch 4, loss 0.319
epoch 6, loss 0.064
epoch 8, loss 0.015
epoch 10, loss 0.004
```

- Let's **take a look at the kernel tensor we learned.**

```
d2l.reshape(conv2d.weight.data, (1, 2))
```

```
tensor([[ 0.9877, -0.9990]])
```

- Indeed, the learned kernel tensor is remarkably close to the kernel tensor K we defined earlier.

CROSS-CORRELATION AND CONVOLUTION

- Recall our observation from :numref:`sec_why-conv` of the correspondence between the cross-correlation and convolution operations.
- For 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.
- Since kernels are learned from data in deep learning, outputs of convolutional layers remain unaffected no matter such layers perform either the strict convolution operations or the cross-correlation operations.
- When the convolutional layer performs strict *convolution* for the input and the flipped kernel, we get the same output as the cross-correlation of the input and original kernel.
- We continue to refer to the cross-correlation operation as a convolution even though, strictly-speaking, it is slightly different.
- We use the term *element* to refer to an entry (or component) of any tensor representing a layer representation or a convolution kernel.

FEATURE MAP AND RECEPTIVE FIELD

- As described in *Why Con Channels*, the convolutional layer output is sometimes called a *feature map*, as it can be regarded as the learned representations in the spatial dimensions 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.
- The receptive field may be larger than the actual size of the input.
- 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.

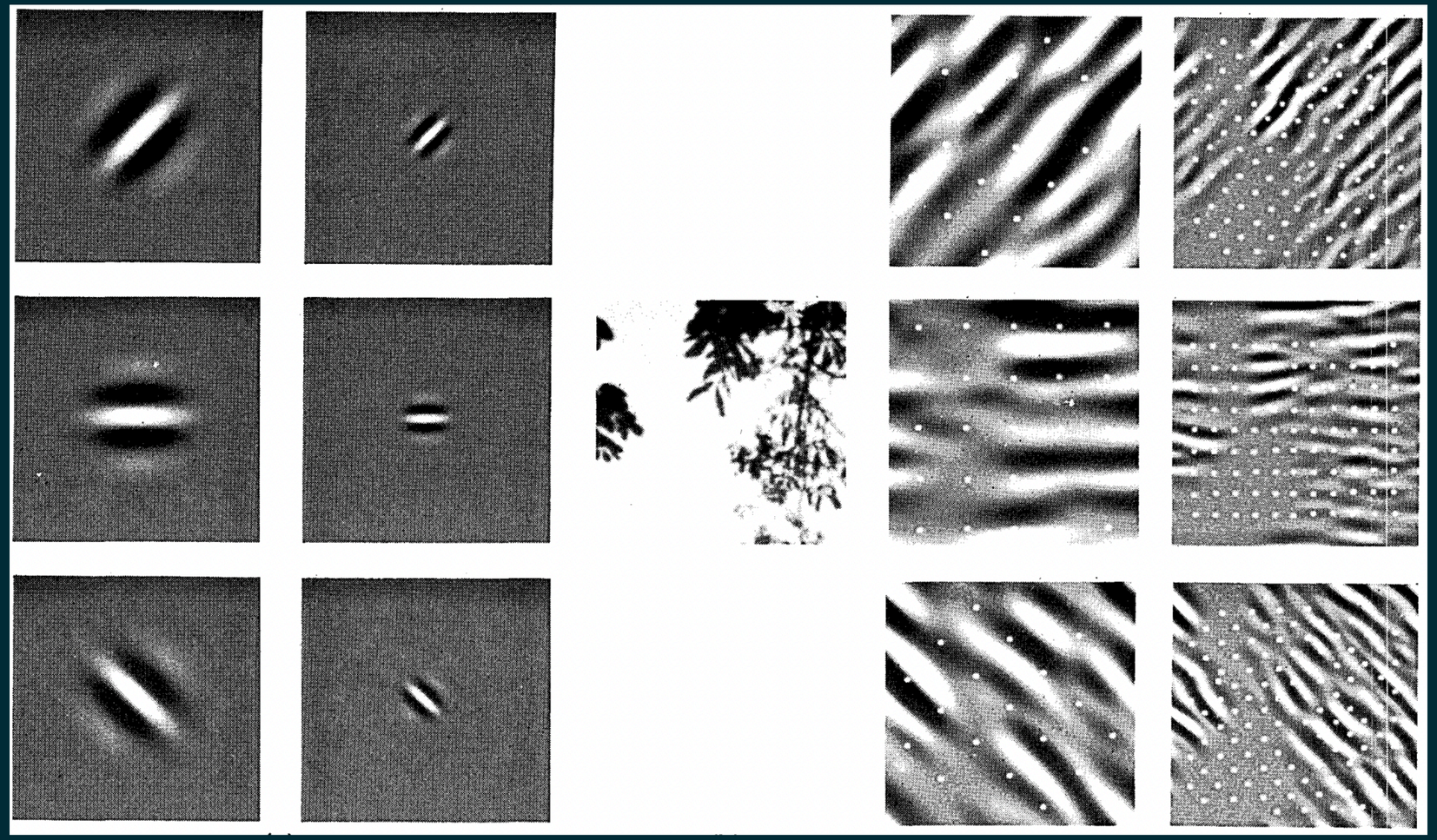


Figure and caption taken from :citet:`Field.1987`: An example of coding with six different channels. (Left) Examples of the six types of sensor associated with each channel. (Right) Convolution of the image in (Middle) with the six sensors shown in (Left). The response of the individual sensors is determined by sampling these filtered images at a distance proportional to the size of the sensor (shown with dots). This diagram shows the response of only the even symmetric sensors.

- this relation holds for features computed by deeper layers of networks trained on image classification tasks. Convolutions have proven to be an incredibly powerful tool for computer vision, both in biology and in code.

SUMMARY

- core computation required for a convolutional layer is a cross-correlation operation. We saw that a simple nested for-loop is all that is required to compute its value.
- If we have multiple input and multiple output channels, we are performing a matrix-matrix operation between channels.
- As can be seen, the computation is straightforward and, most importantly, highly *local*.
- This affords significant hardware optimization and many recent results in computer vision are only possible due to that. After all, it means that chip designers can invest into fast computation rather than memory, when it comes to optimizing for convolutions. While this may not lead to optimal designs for other applications, it opens the door to ubiquitous and affordable computer vision.
- convolutions can be used for many purposes such as to detect edges and lines, to blur images, or to sharpen them.
- Most importantly, it is not necessary that the statistician (or engineer) invents suitable filters.
- Instead, we can simply *learn* them from data. This replaces feature engineering heuristics by evidence-based statistics.
- filters are not just advantageous for building deep networks but also correspond to receptive fields and feature maps in the brain

EXERCISES

1. Construct an image X with diagonal edges.
    1. What happens if you apply the kernel K in this section to it?
    2. What happens if you transpose X?
    3. What happens if you transpose K?
2. Design some kernels manually.
    1. Given a directional vector $\mathbf{v} = (v_1, v_2)$, derive an edge-detection kernel that detects edges orthogonal to $\mathbf{v}$, i.e., edges in the direction $(v_2, -v_1)$.
    2. Derive a finite difference operator for the second derivative. What is the minimum size of the convolutional kernel associate with it? Which structures in images respond most strongly to it?
    3. How would you design a blur kernel? Why might you want to use such a kernel?
    4. What is the minimum size of a kernel to obtain a derivative of order d?
3. When you try to automatically find the gradient for the `Conv2D` class we created, what kind of error message do you see?
4. How do you represent a cross-correlation operation as a matrix multiplication by changing the input and kernel tensors?

PADDING AND STRIDE

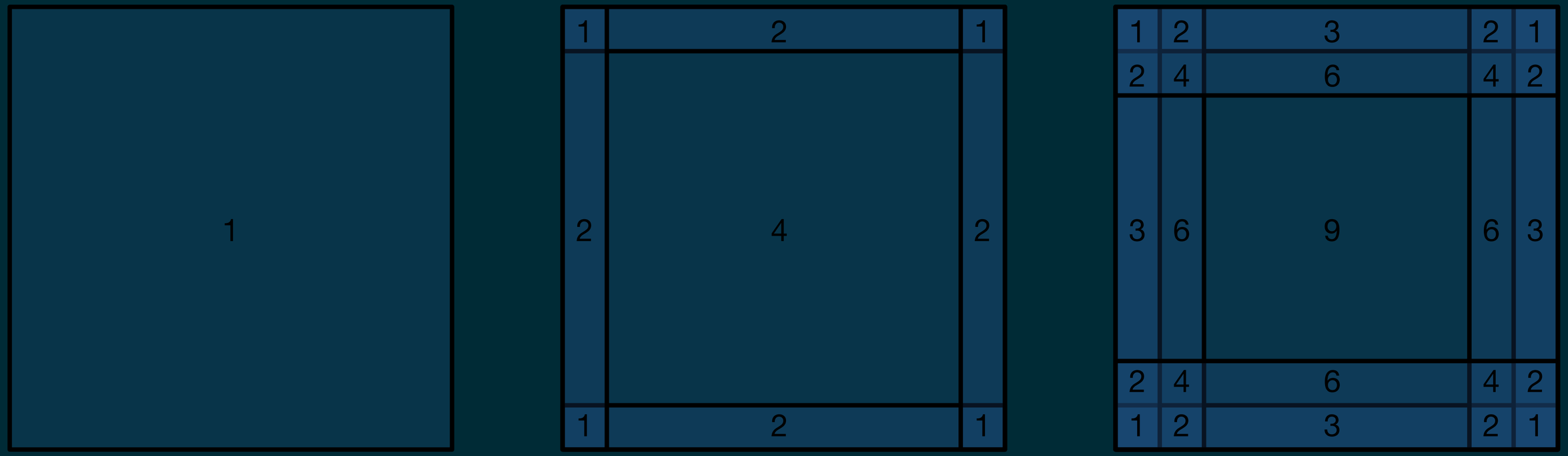
- Recall the example of a convolution in :numref:`fig_correlation`. The input had both a height and width of 3 and the convolution kernel had both a height and width of 2, yielding an output representation with dimension 2×2.
- Assuming that the input shape is $n_h \times n_w$ and the convolution kernel shape is $k_h \times k_w$, the output shape will be $(n_h - k_h + 1) \times (n_w - k_w + 1)$:
- we can only shift the convolution kernel so far until it runs out of pixels to apply the convolution to.
- In the following we will explore a number of techniques, including padding and strided convolutions, that offer more control over 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 240×240 pixel image, 10 layers of 5×5 convolutions reduce the image to 200×200 pixels, slicing off 30% of the image and with it obliterating any interesting information on the boundaries of the original image.

- *Padding* most popular tool for handling this issue.
- may want to reduce the dimensionality drastically, e.g., if we find the original input resolution to be unwieldy. *Strided convolutions* are a popular technique that can help in these instances.

```
1 import torch
2 from torch import nn
```

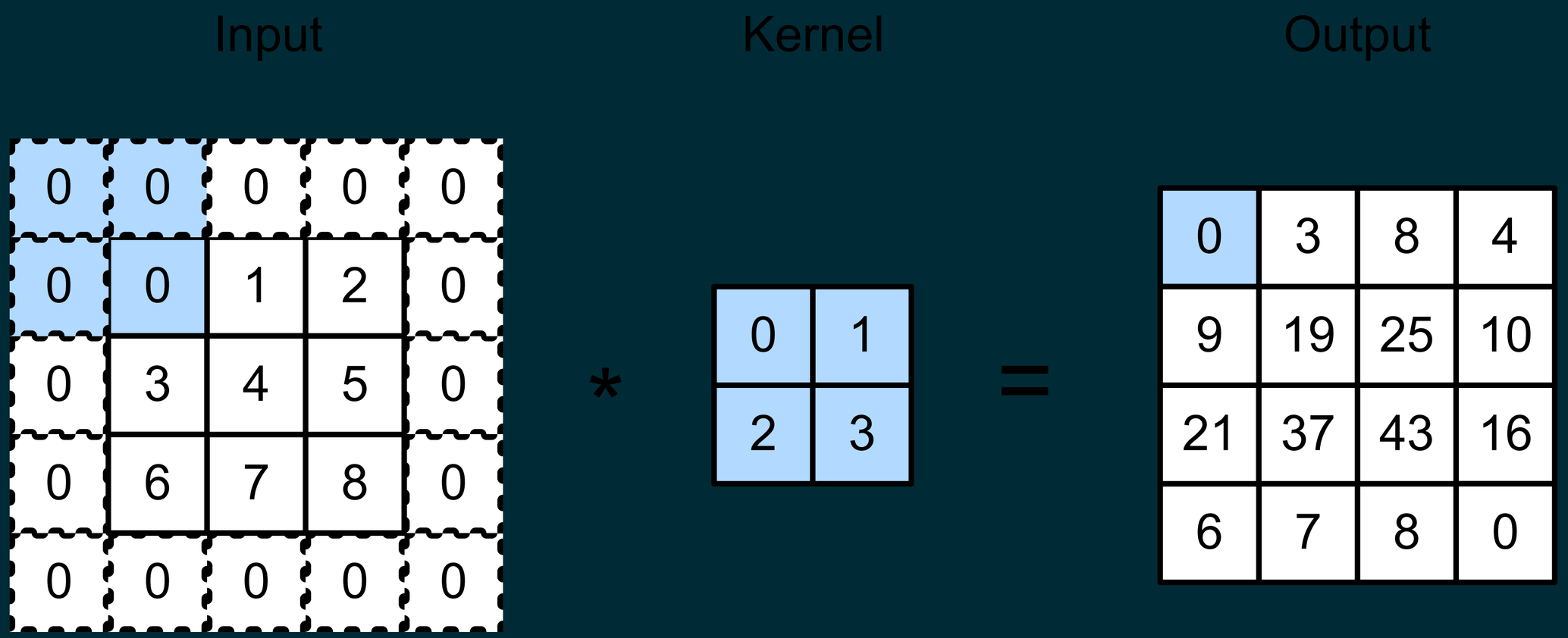
PADDING

- As described above, one tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image. Consider :numref:`img_conv_reuse` that depicts the pixel utilization as a function of the convolution kernel size and the position within the image. The pixels in the corners are hardly used at all.



Pixel utilization for convolutions of size 1×1, 2×2, and 3×3 respectively.

- 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 of filler 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 :numref:`img_conv_pad`, we pad a 3×3 input, increasing its size to 5×5.
- The corresponding output then increases to a 4×4 matrix. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$.



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 pad both sides of the width in the same way.
- 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 dimensionality while padding with the same number of rows on top and bottom, and the same number of columns on left and right.
- practice of using odd kernels and padding to precisely preserve dimensionality offers a clerical benefit.
- any two-dimensional tensor `X`, when the kernel's size is odd and the number of adding 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 `Y[i, j]` is calculated by cross-correlation of the input and convolution kernel with the window centered on `X[i, j]`.

- In the following example, we create a two-dimensional convolutional layer with a height and width of 3 and (**apply 1 pixel of padding on all sides.**)
- input with a height and width of 8, we find that the height and width of the output is also 8.

```
## We define a helper function to calculate convolutions. It initializes the
## convolutional layer weights and performs corresponding dimensionality
## elevations and reductions on the input and output
def comp_conv2d(conv2d, X):
    ## (1, 1) indicates that batch size and the number of channels are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    ## Strip the first two dimensions: examples and channels
    return Y.reshape(Y.shape[2:])

## 1 row and column is padded on either side, so a total of 2 rows or columns
## are added
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

- when the height and width of the convolution kernel are different, we can make the output and input have the same height and width by [**setting different padding numbers for height and width.**]

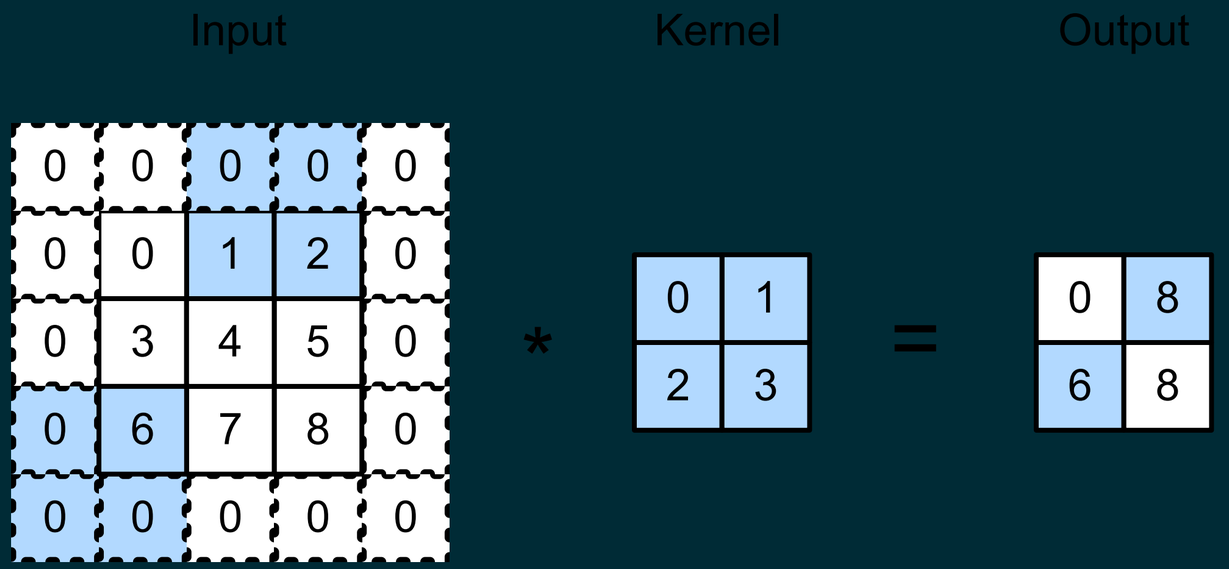
```
## We use a convolution kernel with height 5 and width 3. The padding on either
## side of the height and width are 2 and 1, respectively
conv2d = nn.LazyConv2d(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

STRIDE

- computing the cross-correlation, we start with the convolution window at the upper-left corner of the input tensor, then slide it over all locations both down and to the right.
- previous: defaulted to sliding one element at a time.
- 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.
- particularly useful if the convolution kernel is large since it captures a large area of the underlying image.
- refer to the number of rows and columns traversed per slide as *stride*.
- So far, we have used strides of 1, both for height and width. Sometimes, we may want to use a larger stride. :numref:`img_conv_stride` 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 generated, 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 generated.
- 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).



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 can 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 $(n_h/s_h) \times (n_w/s_w)$. Below, we [**set the strides on both the height and width to 2**], thus halving the input height and width.

```
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

```
torch.Size([4, 4])
```

- Let's look at (**a slightly more complicated example**).

```
conv2d = nn.LazyConv2d(1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([2, 2])
```

SUMMARY AND DISCUSSION

- Padding can increase the height and width of the output. This is often used to give the output the same height and width as the input to avoid undesirable shrinkage of the output.
- Moreover, it ensures that all pixels are used equally frequently.
- Typically we pick symmetric padding on both sides of the input height and width. In this case we refer to (p_h, p_w) padding. Most commonly we set $p_h = p_w$, in which case we simply state that we choose padding p.
- A similar convention applies to strides. When horizontal stride s_h and vertical stride s_w match, we simply talk about stride s.
- The stride can reduce the resolution of the output, for example reducing the height and width of the output to only $1/n$ of the height and width of the input for $n > 1$. By default, the padding is 0 and the stride is 1.
- So far all padding that we discussed simply extended images with zeros.

- This has significant computational benefit since it is trivial to accomplish.
- Moreover, operators can be engineered to take advantage of this padding implicitly without the need to allocate additional memory.
- At the same time, it allows CNNs to encode implicit position information within an image, simply by learning where the “whitespace” is.
- There are many alternatives to zero-padding. :citet:`Alsallakh.Kokhlikyan.Miglani.ea.2020` provided an extensive overview of alternatives (albeit without a clear case to use nonzero paddings unless artifacts occur).

EXERCISES

1. Given the last code example in this section with kernel size $(3, 5)$, padding $(0, 1)$, and stride $(3, 4)$, calculate the output shape to check if it is consistent with the experimental result.
2. For audio signals, what does a stride of 2 correspond to?
3. Implement mirror padding, i.e., padding where the border values are simply mirrored to extend tensors.
4. What are the computational benefits of a stride larger than 1?
5. What might be statistical benefits of a stride larger than 1?
6. How would you implement a stride of $\frac{1}{2}$? What does it correspond to? When would this be useful?

MULTIPLE INPUT AND MULTIPLE OUTPUT CHANNELS

- while we 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 :numref:`subsec_why-conv-channels`, until now, we simplified all of our numerical examples by working with just a single input and a single output channel.
- This 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. The notion of channels is as old as CNNs themselves. For instance LeNet5 :cite:`LeCun.Jackel.Bottou.ea.1995` uses them. In this section, we will take a deeper look at convolution kernels with multiple input and multiple output channels.

```
from d2l import torch as d2l
import torch
```

MULTIPLE INPUT CHANNELS

- When the input data contains multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data.
- For input data with c_i channels, the convolution kernel needs c_i channels, with shape $c_i \times k_h \times k_w$.
- The output is computed by performing cross-correlation on each channel and summing the results.

```python
def corr2d_multi_in(X, K):
    ## Iterate through the 0th dimension (channel) of K first, then add them up
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

We can validate this with an example:

```python
X = d2l.tensor([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
                [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = d2l.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

corr2d_multi_in(X, K)
```

```
tensor([[ 56.,  72.],
        [104., 120.]])
```

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 :numref:`subsec_why-conv-channels`, it turns out to be essential to have multiple channels at each layer.
- Denote by c_i and c_o 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 $c_i \times k_h \times k_w$ for *every* output channel.
- We concatenate them on the output channel dimension, so that the shape of the convolution kernel is $c_o \times c_i \times k_h \times k_w$.

```
def corr2d_multi_in_out(X, K):
    ## Iterate through the 0th dimension of K, and each time, perform
    ## cross-correlation operations with input X. All of the results are
    ## stacked together
    return d2l.stack([corr2d_multi_in(X, k) for k in K], 0)
```

```
K = d2l.stack((K, K + 1, K + 2), 0)
K.shape
```

```
torch.Size([3, 2, 2, 2])
```

Below, we perform cross-correlation operations on the input tensor X with the kernel tensor K. Now the output contains 3 channels.

```
corr2d_multi_in_out(X, K)
```

```
tensor([[[ 56.,  72.],
         [104., 120.]],

        [[ 76., 100.],
```

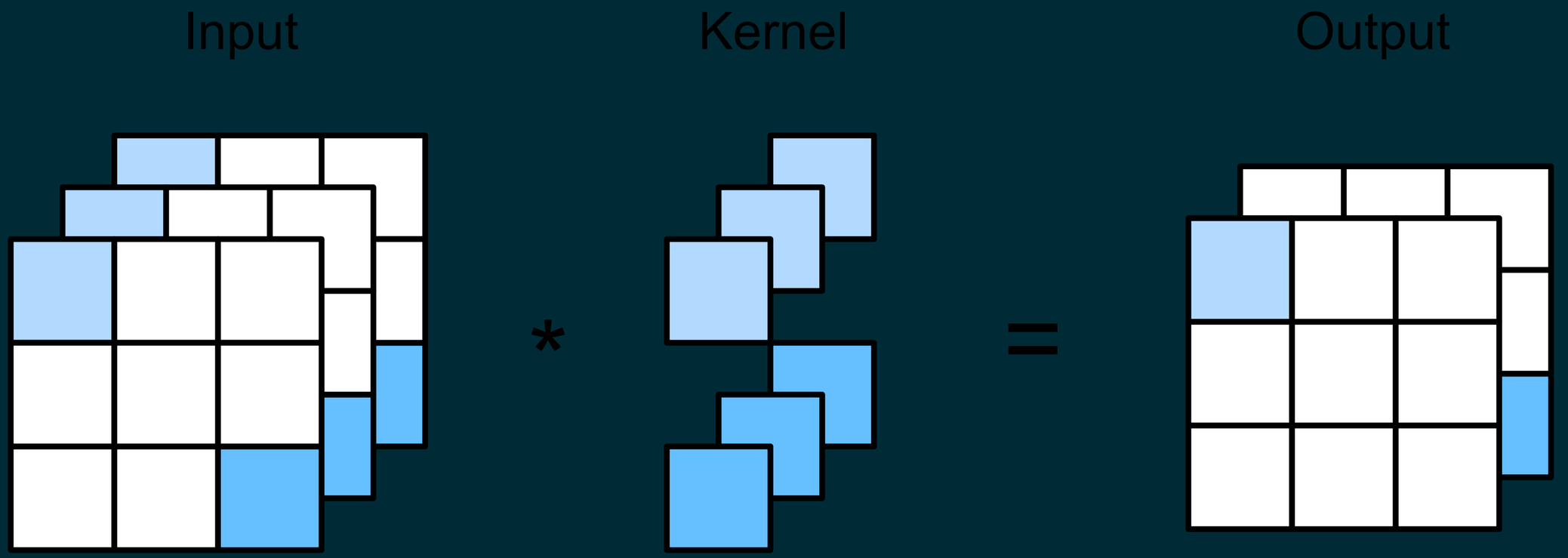
```
 [148., 172.]],

[[ 96., 128.],
 [192., 224.]]])
```

1×1 CONVOLUTIONAL LAYER

- At first, a [1×1 **convolution**], i.e., $k_h = k_w = 1$, does not seem to make much sense.
- After all, a convolution correlates adjacent pixels. A 1×1 convolution obviously does not. Nonetheless, they are popular operations that are sometimes included in the designs of complex deep networks :cite:`Lin.Chen.Yan.2013,Szegedy.Ioffe.Vanhoucke.ea.2017` Let's see in some detail what it actually does.
- Because the minimum window is used, the 1×1 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 1×1 convolution occurs on the channel dimension.
- :numref:`fig_conv_1x1` shows the cross-correlation computation using the 1×1 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.
- You could think of the 1×1 convolutional layer as constituting a fully connected layer applied at every single pixel location to transform the c_i corresponding input values into c_o output values.

- Because this is still a convolutional layer, the weights are tied across pixel location. Thus the 1×1 convolutional layer requires $c_o \times c_i$ weights (plus the bias). Also note that convolutional layers are typically followed by nonlinearities. This ensures that 1×1 convolutions cannot simply be folded into other convolutions.



The cross-correlation computation uses the 1×1 convolution kernel with 3 input channels and 2 output channels. The input and output have the same height and width.

```python
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = d2l.reshape(X, (c_i, h * w))
    K = d2l.reshape(K, (c_o, c_i))
    ## Matrix multiplication in the fully connected layer
    Y = d2l.matmul(K, X)
    return d2l.reshape(Y, (c_o, h, w))
```

- When performing 1×1 convolutions, the above function is equivalent to the previously implemented cross-correlation function `corr2d_multi_in_out`. Let's check this with some sample data.

```python
X = d2l.normal(0, 1, (3, 3, 3))
K = d2l.normal(0, 1, (2, 3, 1, 1))
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(d2l.reduce_sum(d2l.abs(Y1 - Y2))) < 1e-6
```

DISCUSSION

- Channels allow us to combine the best of both worlds: MLPs that allow for significant nonlinearities and convolutions that allow for *localized* analysis of features. In particular, channels allow the CNN to reason with multiple features, such as edge and shape detectors at the same time.
- also offer a practical trade-off between the drastic parameter reduction arising from translation invariance and locality, and the need for expressive and diverse models in computer vision.
- flexibility comes at a price. Given an image of size $(h \times w)$, the cost for computing a $k \times k$ convolution is $\mathcal{O}(h \cdot w \cdot k^2)$. For c_i and c_o input and output channels respectively this increases to $\mathcal{O}(h \cdot w \cdot k^2 \cdot c_i \cdot c_o)$.
- For a 256×256 pixel image with a 5×5 kernel and 128 input and output channels respectively
- amounts to over 53 billion operations (we count multiplications and additions separately).
- Later: effective strategies to cut down on the cost, e.g., by requiring the channel-wise operations to be block-diagonal, leading to architectures such as ResNeXt :cite:`Xie.Girshick.Dollar.ea.2017`.

EXERCISES

1. Assume that we have two convolution kernels of size k_1 and k_2, respectively (with no nonlinearity in-between).
   1. Prove that the result of the operation can be expressed by a single convolution.
   2. What is the dimensionality of the equivalent single convolution?
   3. Is the converse true, i.e., can you always decompose a convolution into two smaller ones?
2. Assume an input of shape $c_i \times h \times w$ and a convolution kernel of shape $c_o \times c_i \times k_h \times k_w$, padding of (p_h, p_w), and stride of (s_h, s_w).
   1. What is the computational cost (multiplications and additions) for the forward propagation?
   2. What is the memory footprint?
   3. What is the memory footprint for the backward computation?
   4. What is the computational cost for the backpropagation?
3. By what factor does the number of calculations increase if we double the number of input channels c_i and the number of output channels c_o? What happens if we double the padding?

1. Are the variables Y1 and Y2 in the last example of this section exactly the same? Why?
2. Express convolutions as a matrix multiplication, even when the convolution window is not 1×1?
3. Your task is to implement fast convolutions with a $k \times k$ kernel. One of the algorithm candidates is to scan horizontally across the source, reading a k-wide strip and computing the 1-wide output strip one value at a time. The alternative is to read a $k + \Delta$ wide strip and compute a Δ-wide output strip. Why is the latter preferable? Is there a limit to how large you should choose Δ?
4. Assume that we have a $c \times c$ matrix.
    1. How much faster is it to multiply with a block-diagonal matrix if the matrix is broken up into b blocks?
    2. What is the downside of having b blocks? How could you fix it, at least partly?

POOLING

- in many cases our ultimate task asks some global question about the image, e.g., *does it contain a cat?* Consequently, the units of our final layer should be sensitive to the entire input.
- By gradually aggregating information, yielding coarser and coarser maps, we accomplish this goal of ultimately learning a global representation, while keeping all of the advantages of convolutional layers at the intermediate layers of processing.
- The deeper we go in the network, the larger the receptive field (relative to the input) to which each hidden node is sensitive. Reducing spatial resolution accelerates this process, since the convolution kernels cover a larger effective area.
- Moreover, when detecting lower-level features, such as edges (as discussed in :numref:`sec_conv_layer`), we often want our representations to be somewhat invariant to translation.
- For instance, if we take the image `X` with a sharp delineation between black and white and shift the whole image by one pixel to the right, i.e., `Z[i, j] = X[i, j + 1]`, then the output for the new image `Z` might be vastly different.

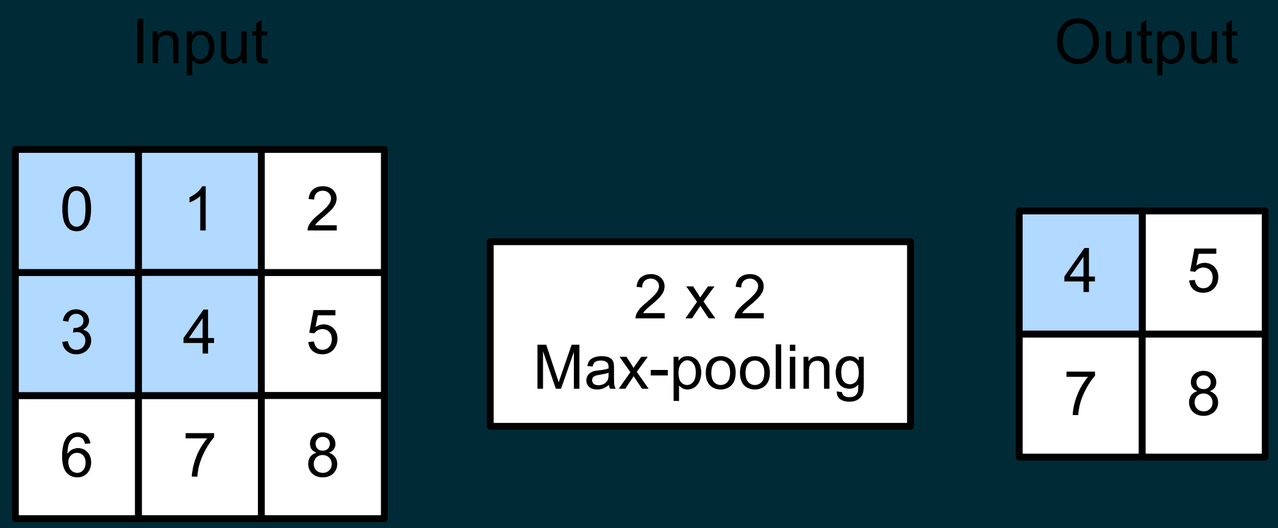
- The edge will have shifted by one pixel. In reality, objects hardly ever occur exactly at the same place. In fact, even with a tripod and a stationary object, vibration of the camera due to the movement of the shutter might shift everything by a pixel or so (high-end cameras are loaded with special features to address this problem).
- This section introduces *pooling layers*, which serve the dual purposes of mitigating the sensitivity of convolutional layers to location and of spatially downsampling representations.

```
from d2l import torch as d2l
import torch
from torch import nn
```

MAXIMUM POOLING AND AVERAGE POOLING

- Like convolutional layers, *pooling* operators consist of a fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed by the fixed-shape window (sometimes known as the *pooling window*).
- However, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters (there is no *kernel*).
- Instead, pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window.
- These operations are called *maximum pooling* (*max-pooling* for short) and *average pooling*, respectively.
- *Average pooling* is essentially as old as CNNs. The idea is akin to downsampling an image.
- Rather than just taking the value of every second (or third) pixel for the lower resolution image, we can average over adjacent pixels to obtain an image with better signal to noise ratio since we are combining the information from multiple adjacent pixels.

- *Max-pooling* was introduced in :citet:`Riesenhuber.Poggio.1999` in the context of cognitive neuroscience to describe how information aggregation might be aggregated hierarchically for the purpose of object recognition, and an earlier version in speech recognition :cite:`Yamaguchi.Sakamoto.Akabane.ea.1990`.
- In almost all cases, max-pooling, as it is also referred to, is preferable.
- In both cases, as with the cross-correlation operator, we can think of the pooling window as starting from the upper-left of the input tensor and sliding across the input tensor from left to right and top to bottom.
- At each location that the pooling window hits, it computes the maximum or average value of the input subtensor in the window, depending on whether max or average pooling is employed.



Max-pooling with a pooling window shape of 2×2. The shaded portions are the first output element as well as the input tensor elements used for the output computation: $\max(0, 1, 3, 4) = 4$.

- output tensor in :numref:`fig_pooling` has a height of 2 and a width of 2. The four elements are derived from the maximum value in each pooling window:

$$\begin{aligned}\max(0, 1, 3, 4) &= 4,\\ \max(1, 2, 4, 5) &= 5,\\ \max(3, 4, 6, 7) &= 7,\\ \max(4, 5, 7, 8) &= 8.\end{aligned}$$

- More generally, we can define a $p \times q$ pooling layer by aggregating over a region of said size. Returning to the problem of edge detection, we use the output of the convolutional layer as input for 2×2 max-pooling.
- Denote by `X` the input of the convolutional layer input and `Y` the pooling layer output. Regardless of whether or not the values of `X[i, j]`, `X[i, j + 1]`, `X[i+1, j]` and `X[i+1, j + 1]` are different, the pooling layer always outputs `Y[i, j] = 1`.
- That is to say, using the 2×2 max-pooling layer, we can still detect if the pattern recognized by the convolutional layer moves no more than one element in height or width.
- In the code below, we (**implement the forward propagation of the pooling layer**) in the `pool2d` function. This function is similar to the `corr2d` function in :numref:`sec_conv_layer`.

- However, no kernel is needed, computing the output as either the maximum or the average of each region in the input.

```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = d2l.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i: i + p_h, j: j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
    return Y
```

- can construct the input tensor `X` in :numref:`fig_pooling` to [**validate the output of the two-dimensional max-pooling layer**].

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
tensor([[4., 5.],
        [7., 8.]])
```

- Also, we experiment with (**the average pooling layer**).

```
pool2d(X, (2, 2), 'avg')
```

```
tensor([[2., 3.],
        [5., 6.]])
```

PADDING AND STRIDE

- As with convolutional layers, pooling layers change the output shape. And as before, we can adjust the operation to achieve a desired output shape by padding the input and adjusting the stride.
- We can demonstrate the use of padding and strides in pooling layers via the built-in two-dimensional max-pooling layer from the deep learning framework.
- We first construct an input tensor X whose shape has four dimensions, where the number of examples (batch size) and number of channels are both 1.

```
1 X = d2l.reshape(d2l.arange(16, dtype=d2l.float32), (1, 1, 4, 4))
2 X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

- pooling aggregates information from an area, (**deep learning frameworks default to matching pooling window sizes and stride.**) For instance, if we use a pooling window of shape `(3, 3)` we get a stride shape of `(3, 3)` by default.

```
pool2d = nn.MaxPool2d(3)
## Pooling has no model parameters, hence it needs no initialization
pool2d(X)
```

```
tensor([[[[10.]]]])
```

- as expected, [**the stride and padding can be manually specified**] to override framework defaults if needed.

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

Of course, we can specify an arbitrary rectangular pooling window with arbitrary height and width respectively, as the example below shows.

```
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

MULTIPLE CHANNELS

- When processing multi-channel input data, [**the pooling layer pools each input channel separately**], rather than summing the inputs up over channels as in a convolutional layer.
- This means that the number of output channels for the pooling layer is the same as the number of input channels. Below, we will concatenate tensors `X` and `X + 1` on the channel dimension to construct an input with 2 channels.

```
1 X = d2l.concat((X, X + 1), 1)
2 X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]],

         [[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
```

As we can see, the number of output channels is still 2 after pooling.

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]],

         [[ 6.,  8.],
          [14., 16.]]]])
```

SUMMARY

- Pooling is an exceedingly simple operation. It does exactly what its name indicates, aggregate results over a window of values.
- All convolution semantics, such as strides and padding apply in the same way as they did previously.
- Note that pooling is indifferent to channels, i.e., it leaves the number of channels unchanged and it applies to each channel separately. Lastly, of the two popular pooling choices, max-pooling is preferable to average pooling, as it confers some degree of invariance to output. A popular choice is to pick a pooling window size of 2×2 to quarter the spatial resolution of output.
- Note that there are many more ways of reducing resolution beyond pooling. For instance, in stochastic pooling :cite:`Zeiler.Fergus.2013` and fractional max-pooling :cite:`Graham.2014` aggregation is combined with randomization.
- This can slightly improve the accuracy in some cases. Lastly, as we will see later with the attention mechanism, there are more refined ways of aggregating over outputs, e.g., by using the alignment between a query and representation vectors.

EXERCISES

1. Implement average pooling through a convolution.
2. Prove that max-pooling cannot be implemented through a convolution alone.
3. Max-pooling can be accomplished using ReLU operations, i.e., $\mathrm{ReLU}(x) = \max(0, x)$.
    1. Express $\max(a, b)$ by using only ReLU operations.
    2. Use this to implement max-pooling by means of convolutions and ReLU layers.
    3. How many channels and layers do you need for a 2×2 convolution? How many for a 3×3 convolution.

2. What is the computational cost of the pooling layer? Assume that the input to the pooling layer is of size $c \times h \times w$, the pooling window has a shape of $p_h \times p_w$ with a padding of (p_h, p_w) and a stride of (s_h, s_w).
3. Why do you expect max-pooling and average pooling to work differently?
4. Do we need a separate minimum pooling layer? Can you replace it with another operation?
5. We could use the softmax operation for pooling. Why might it not be so popular?

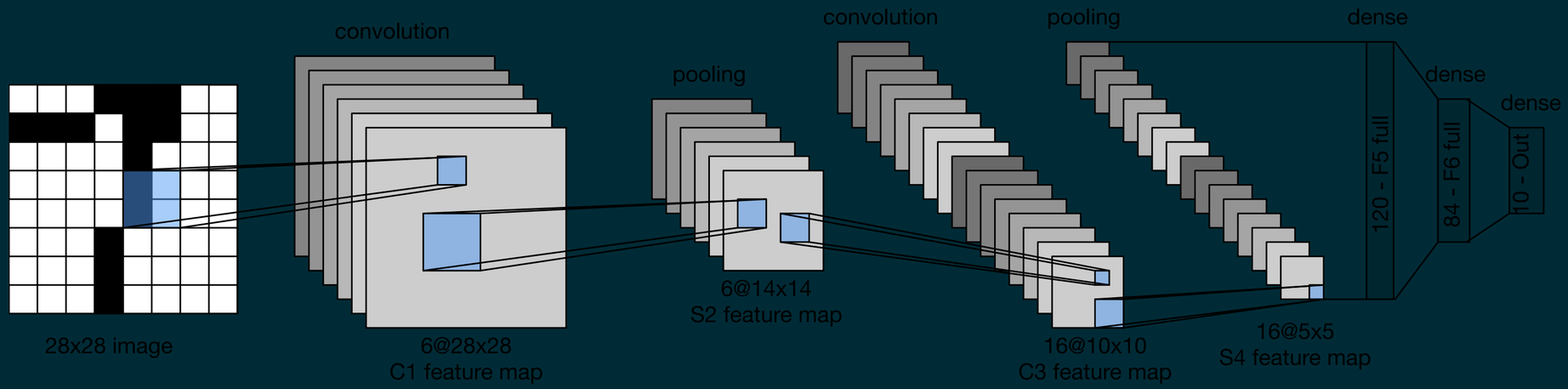
CONVOLUTIONAL NEURAL NETWORKS (LENET)

- We now have all the ingredients required to assemble a fully-functional CNN. In our earlier encounter with image data, we applied a linear model with softmax regression and an MLP to pictures of clothing in the Fashion-MNIST dataset. To make such data amenable we first flattened each image from a 28×28 matrix into a fixed-length 784-dimensional vector. Now that we have a handle on convolutional layers, we can retain the spatial structure in our images.
- In this section, we will introduce *LeNet*, among the first published CNNs to capture wide attention for its performance on computer vision tasks. The model was introduced by (and named for) Yann LeCun, then a researcher at AT&T Bell Labs, for the purpose of recognizing handwritten digits in images :cite:`LeCun.Bottou.Bengio.ea.1998`.
- LeNet achieved outstanding results matching the performance of support vector machines, achieving an error rate of less than 1% per digit. LeNet was eventually adapted to recognize digits for processing deposits in ATM machines.

```
from d2l import torch as d2l
import torch
from torch import nn
```

LENET

- LeNet (LeNet-5) consists of two parts:
- i. a convolutional encoder consisting of two convolutional layers; and
- ii. a dense block consisting of three fully connected layers
- The architecture is summarized in :numref:`img_lenet`.



Data flow in LeNet. The input is a handwritten digit, the output a probability over 10 possible outcomes.

- The basic units in each convolutional block are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation. Note that while ReLUs and max-pooling work better, these discoveries had not yet been made at the time. Each convolutional layer uses a 5×5 kernel and a sigmoid activation function. These layers map spatially arranged inputs to a number of two-dimensional feature maps, typically increasing the number of channels. The first convolutional layer has 6 output channels, while the second has 16. Each 2×2 pooling operation (stride 2) reduces dimensionality by a factor of 4 via spatial downsampling. The convolutional block emits an output with shape given by (batch size, number of channel, height, width).
- In order to pass output from the convolutional block to the dense block, we must flatten each example in the minibatch. In other words, we take this four-dimensional input and transform it into the two-dimensional input expected by fully connected layers:

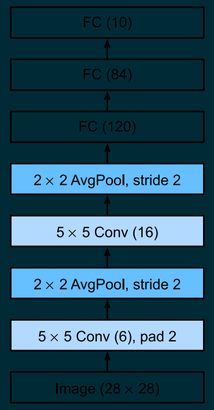
- as a reminder, the two-dimensional representation that we desire uses the first dimension to index examples in the minibatch and the second to give the flat vector representation of each example.
- LeNet's dense block has three fully connected layers, with 120, 84, and 10 outputs, respectively.
- Because we are still performing classification, the 10-dimensional output layer corresponds to the number of possible output classes.
- While getting to the point where you truly understand what is going on inside LeNet may have taken a bit of work, hopefully the following code snippet will convince you that implementing such models with modern deep learning frameworks is remarkably simple. We need only to instantiate a `Sequential` block and chain together the appropriate layers, using Xavier initialization as introduced in :numref:`subsec_xavier`.

```python
def init_cnn(module):  #@save
    """Initialize weights for CNNs."""
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)
```

```python
class LeNet(d2l.Classifier):  #@save
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.Sigmoid(),
            nn.LazyLinear(84), nn.Sigmoid(),
            nn.LazyLinear(num_classes))
```

- take some liberty in the reproduction of LeNet insofar as we replace the Gaussian activation layer by a softmax layer. This greatly simplifies the implementation, not the least due to the fact that the Gaussian decoder is rarely used nowadays. Other than that, this network matches the original LeNet-5 architecture.

- Let's see what happens inside the network. By passing a single-channel (black and white) 28×28 image through the network and printing the output shape at each layer, we can [**inspect the model**] to make sure that its operations line up with what we expect from :numref:`img_lenet_vert`.



:label:`img_lenet_vert`

```
@d2l.add_to_class(d2l.Classifier)  #@save
def layer_summary(self, X_shape):
    X = d2l.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape)

model = LeNet()
model.layer_summary((1, 1, 28, 28))
```

```
Conv2d output shape:     torch.Size([1, 6, 28, 28])
Sigmoid output shape:    torch.Size([1, 6, 28, 28])
AvgPool2d output shape:  torch.Size([1, 6, 14, 14])
Conv2d output shape:     torch.Size([1, 16, 10, 10])
Sigmoid output shape:    torch.Size([1, 16, 10, 10])
AvgPool2d output shape:  torch.Size([1, 16, 5, 5])
Flatten output shape:    torch.Size([1, 400])
Linear output shape:     torch.Size([1, 120])
Sigmoid output shape:    torch.Size([1, 120])
Linear output shape:     torch.Size([1, 84])
```

- height and width of the representation at each layer throughout the convolutional block is reduced (compared with the previous layer).
- The first convolutional layer uses 2 pixels of padding to compensate for the reduction in height and width that would otherwise result from using a 5×5 kernel.
- As an aside, the image size of 28×28 pixels in the original MNIST OCR dataset is a result of *trimming* 2 pixel rows (and columns) from the original scans that measured 32×32 pixels.
- done primarily to save space (a 30% reduction) at a time when Megabytes mattered.

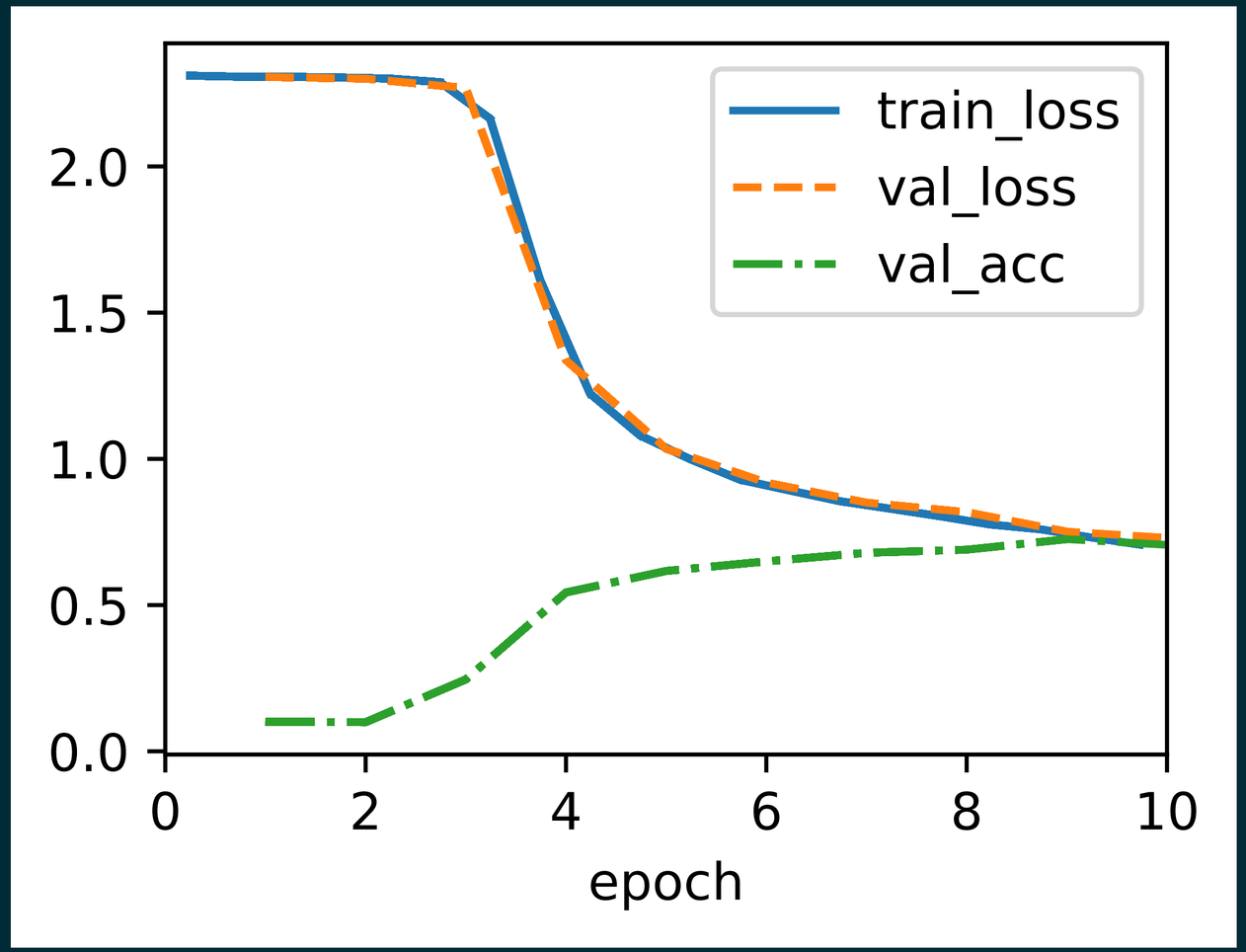
- In contrast, the second convolutional layer forgoes padding, and thus the height and width are both reduced by 4 pixels.
- As we go up the stack of layers, the number of channels increases layer-over-layer from 1 in the input to 6 after the first convolutional layer and 16 after the second convolutional layer.
- However, each pooling layer halves the height and width. Finally, each fully connected layer reduces dimensionality, finally emitting an output whose dimension matches the number of classes.

TRAINING

- Now that we have implemented the model, let's [**run an experiment to see how the LeNet-5 model fares on Fashion-MNIST**].
- While CNNs have fewer parameters, they can still be more expensive to compute than similarly deep MLPs because each parameter participates in many more multiplications.
- If you have access to a GPU, this might be a good time to put it into action to speed up training.
- Note that the `d2l.Trainer` class takes care of all details. By default, it initializes the model parameters on the available devices.

- Just as with MLPs, our loss function is cross-entropy, and we minimize it via minibatch stochastic gradient descent.

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet(lr=0.1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```



SUMMARY

- moved from the MLPs of the 1980s to the CNNs of the 1990s and early 2000s. The architectures proposed, e.g., in the form of LeNet-5 remain meaningful, even to this day. It is worth comparing the error rates on Fashion-MNIST achievable with LeNet-5 both to the very best possible with MLPs (:numref:`sec_mlp-implementation`) and those with significantly more advanced architectures such as ResNet (:numref:`sec_resnet`). LeNet is much more similar to the latter than to the former. One of the primary differences, as we shall see, is that greater amounts of computation afforded significantly more complex architectures.
- second difference is the relative ease with which we were able to implement LeNet. What used to be an engineering challenge worth months of C++ and assembly code, engineering to improve SN, an early Lisp based deep learning tool :cite:`Bottou.Le-Cun.1988`, and finally experimentation with models can now be accomplished in minutes. It is this incredible productivity boost that has democratized deep learning model development tremendously. In the next chapter we will follow down this rabbit to hole to see where it takes us.