T05460301-01

Deep Learning (Fall 2023)

Ikbeom Jang

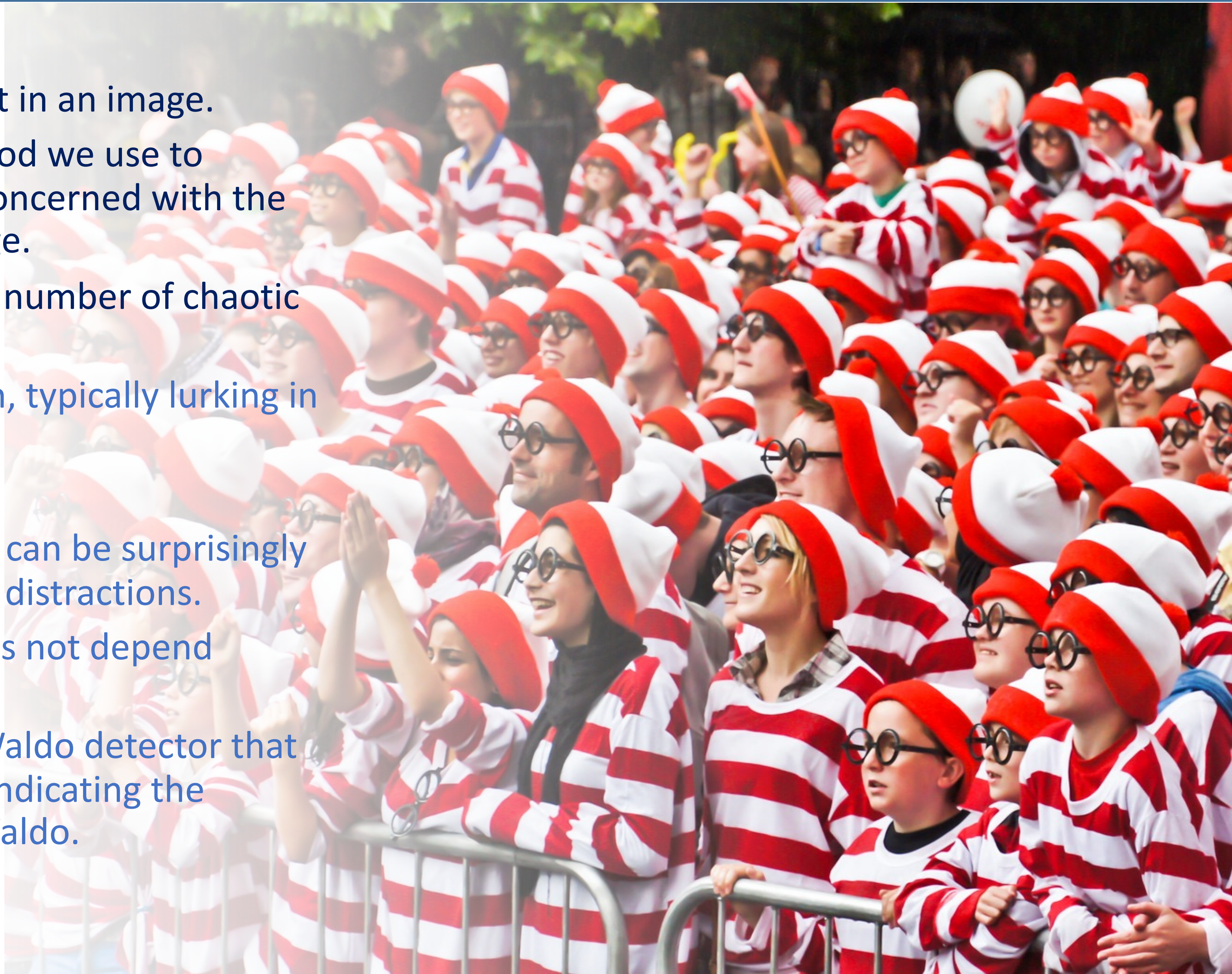
ijang@hufs.ac.kr
CES HUFS

Convolutional Neural Networks

- From Fully Connected Layers to Convolutions
- Convolutions for Images

From Fully Connected Layers to Convolutions

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.
- The game “Where’s Waldo” consists of a number of chaotic scenes bursting with activities.
  - Waldo shows up somewhere in each, typically lurking in some unlikely location.
  - The reader’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.

From Fully Connected Layers to Convolutions

- Many object detection and segmentation algorithms are based on this approach.
- CNNs systematize this idea of spatial invariance, exploiting it to learn useful representations with fewer parameters.

- We can now make these intuitions more concrete by enumerating a few **desired things 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.

From Fully Connected Layers to Convolutions

- Constraining MLP
  - Consider an MLP with 2D images **X** as inputs and their immediate hidden representations **H**.
  - W: weight tensor (4th order; to have each of the hidden units receive input from each of the input pixels)

$$\begin{aligned}[\mathbf{H}]_{i,j} &= [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathsf{W}]_{i,j,k,l} [\mathbf{X}]_{k,l} \\ &= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathsf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}.\end{aligned} \tag{7.1.1}$$

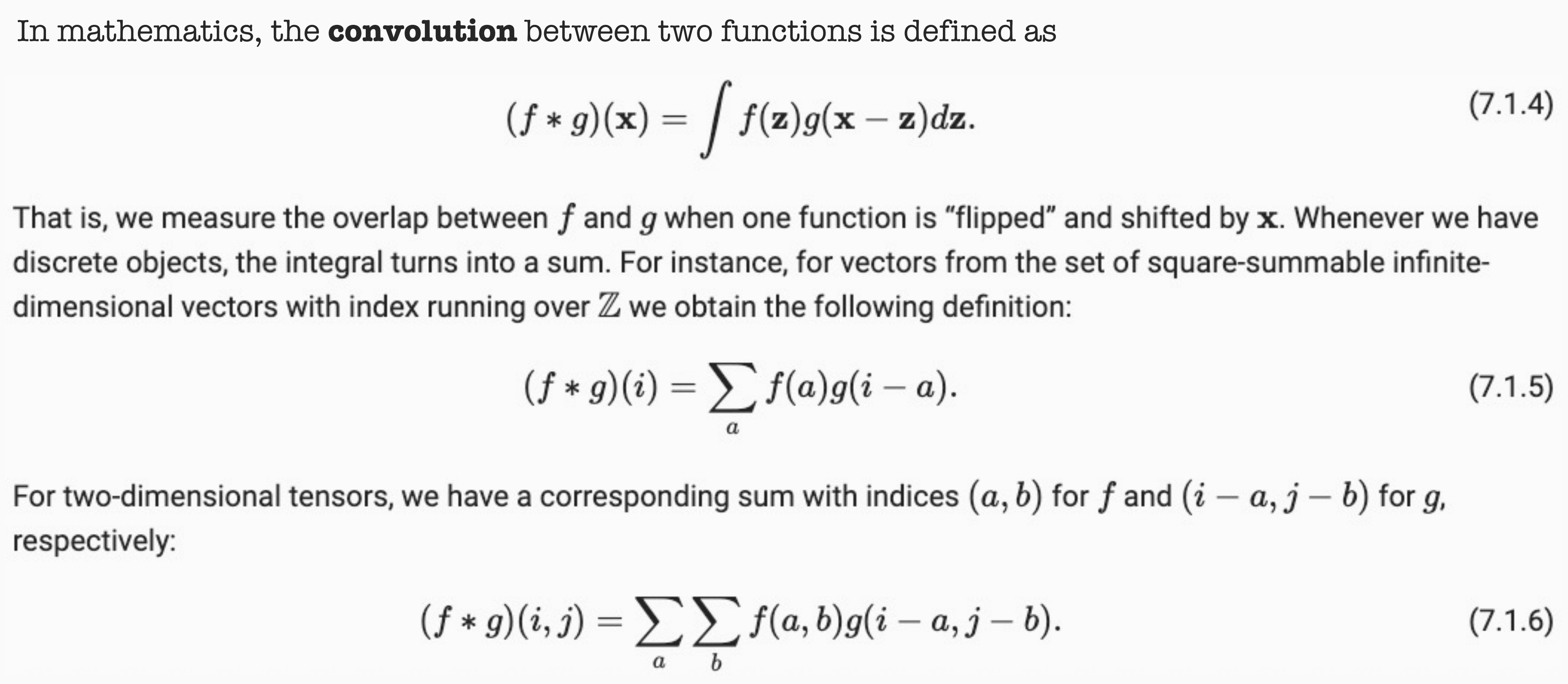
- Translation Invariance
  - A shift in the input **X** should simply lead to a shift in the hidden representation **H**.
  - This is only possible if V and U do not depend on (*i, j*) and U is a constant.

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{7.1.2}$$

- Locality
  - No need to look very far away from location (*i, j*)
  - outside some range $|a|>\Delta$ or $|b|>\Delta$, we should set $[\mathbf{V}]_{a,b}=0$

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{7.1.3}$$

From Fully Connected Layers to Convolutions



In mathematics, the **convolution** between two functions is defined as

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z}) g(\mathbf{x} - \mathbf{z}) d\mathbf{z}. \tag{7.1.4}$$

That is, we measure the overlap between f and g when one function is "flipped" and shifted by $\mathbf{x}$. 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:

$$(f * g)(i) = \sum_a f(a) g(i - a). \tag{7.1.5}$$

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

$$(f * g)(i, j) = \sum_a \sum_b f(a, b) g(i - a, j - b). \tag{7.1.6}$$

From Fully Connected Layers to Convolutions

- Returning to our Waldo detector, let’s see what this looks like.
- The convolutional layer picks windows of a given size and weighs intensities according to the filter.
- We might aim to learn a model so that wherever the “waldoness” is highest, we should find a peak in the hidden layer representations.

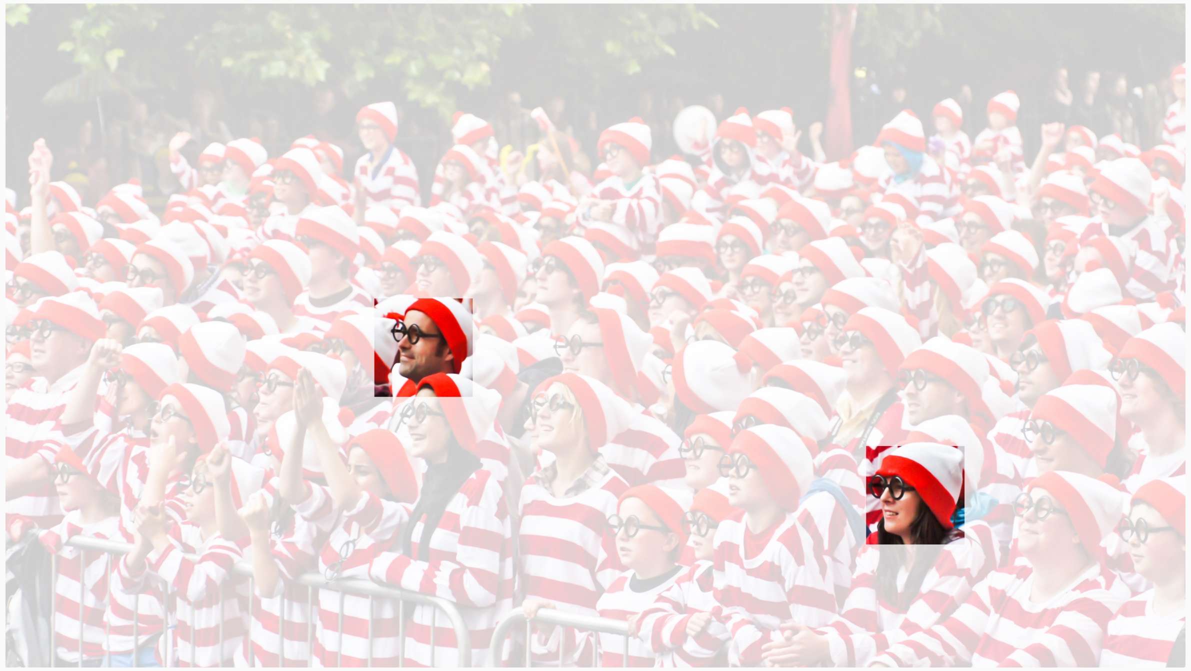


Fig. 7.1.2 Detect Waldo (image courtesy of William Murphy).

From Fully Connected Layers to Convolutions

- So far, we ignored that images consist of three channels: red, green, and blue.
- In sum, images are not two-dimensional objects but rather third-order tensors, characterized by a height, width, and channel.
    - e.g., with shape 1024×1024×3 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 input $\mathbf{X}$ as $[\mathbf{X}]_{i,j,k}$.
    - We index the convolutional filter $[\mathbf{V}]_{a,b,c}$.

From Fully Connected Layers to Convolutions

- Moreover, just as our input consists of a third-order tensor, it is 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 for the subsequent layer.

- Intuitively, you 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**: $[\mathbf{V}]_{a,b,c,d}$.

From Fully Connected Layers to Convolutions

- Putting everything together we have:

$$[\mathsf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} [\mathsf{V}]_{a,b,c,d} [\mathsf{X}]_{i+a,j+b,c}, \tag{7.1.7}$$

- *d*: output channels in the hidden representation **H**

- The subsequent convolutional layer will go on to take a third-order tensor, **H**, as input.
- Because of its generality, we take (7.1.7) as the **definition of a convolutional layer for multiple channels**, where **V** is a kernel or filter of the layer.

Convolutions for Images

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

The Cross-Correlation Operation

- 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 Section 7.1, an input tensor and a kernel tensor are combined to produce an output tensor through a cross-correlation operation.

- Let's ignore channels for now and see how this works with two-dimensional data and hidden representations.
- In Fig. 7.2.1, the input is a two-dimensional tensor with a height of 3 and width of 3.
- We mark the shape of the tensor as 3×3 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 2×2).

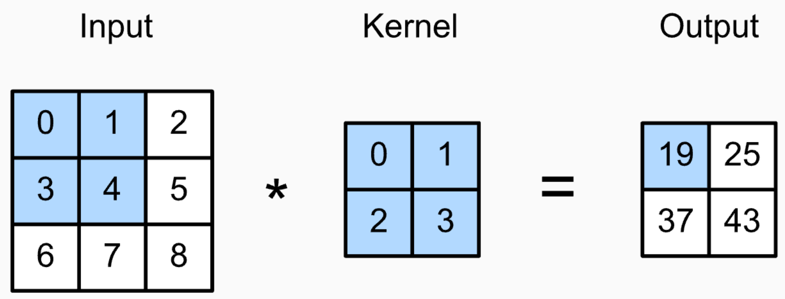


Fig. 7.2.1 Two-dimensional cross-correlation operation. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation:

$$0\times0+1\times1+3\times2+4\times3=19$$

The 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.
- 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:

$$\begin{aligned}
0\times 0+1\times 1+3\times 2+4\times 3&=19,\\
1\times 0+2\times 1+4\times 2+5\times 3&=25,\\
3\times 0+4\times 1+6\times 2+7\times 3&=37,\\
4\times 0+5\times 1+7\times 2+8\times 3&=43.
\end{aligned} \tag{7.2.1}$$

The Cross-Correlation Operation

- Note that along each axis, the output size is slightly smaller than the input size.
- Because the kernel has width and height greater than 1, 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

$$(n_\mathrm{h} - k_\mathrm{h} + 1) \times (n_\mathrm{w} - k_\mathrm{w} + 1). \tag{7.2.2}$$

- 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.

- Next, we implement this process in the *corr2d* function, which accepts an input tensor X and a kernel tensor K and returns an output tensor Y.

The Cross-Correlation Operation

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

```
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

Output:

```
tensor([[19., 25.],
        [37., 43.]])
```

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.

- We are now ready to implement a two-dimensional convolutional layer based on the *corr2d* function defined above.
- In the __init__ constructor method, we declare weight and bias as the two model parameters.
- The forward propagation method calls the *corr2d* function and adds the bias.

Convolutional Layers

```
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 an $h \times w$ convolution kernel, the height and width of the convolution kernel are h and w, respectively.
- We refer to a convolutional layer with an $h \times w$ convolution kernel simply as an ”$h \times w$ convolutional layer”.

Object Edge Detection in Images

- Simple application of a convolutional layer
  - Detecting the edge of an object in an image by finding the location of the pixel change.

- First, we construct an “image” of 6×8 pixels.
- The middle four columns are black (0) and the rest are white (1).

```
X = torch.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.]])
```

Object Edge Detection in Images

- Next, we construct a **kernel K** with a **height of 1 and a width of 2**.
- When we perform the cross-correlation operation with the input,
  - if the horizontally adjacent elements are the same, the output is 0.
  - Otherwise, the output is nonzero.
- Note that this kernel is a special case of a finite difference operator.
- At location (*i*, *j*) it computes $x_{i,j} - x_{i+1,j}$,

  i.e., it computes the difference between the values of horizontally adjacent pixels.
- This is a discrete approximation of the first derivative in the horizontal direction.
- After all, for a function $f(i,j)$ its derivative $\partial_i f(i,j) = \lim_{\epsilon \to 0} \frac{f(i,j) - f(i+\epsilon,j)}{\epsilon}$.
- Let's see how this works in practice.

Object Edge Detection in Images

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

- We are ready to perform the cross-correlation operation with arguments X (our input) and K (our kernel).
- As you can see, 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.]])
```

Object Edge Detection in Images

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

```
Z = corr2d(X.t(), K)
Z
```

```
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

- Designing an **edge detector** by finite differences [1, -1] is neat 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.

- 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.
    - We can then calculate the gradient to update the kernel.
    - For simplicity, we will use the built-in class for two-dimensional convolutional layers and ignore the bias.

Learning a Kernel

```python
# 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:
        print(f'epoch {i + 1}, loss {l.sum():.3f}')
```

Learning a Kernel

```
epoch 2, loss 16.481
epoch 4, loss 5.069
epoch 6, loss 1.794
epoch 8, loss 0.688
epoch 10, loss 0.274
```

- Note that the error has dropped to a small value after 10 iterations.
- Now we will take a look at the kernel tensor we learned.

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

```
tensor([[ 1.0398, -0.9328]])
```

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

Feature Map & Receptive Field

- The convolutional layer output in Fig. 7.2.1 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.

- In Fig. 7.2.1, given the 2×2 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's denote the 2×2 output as **Y** and consider a deeper CNN with an additional 2×2 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.

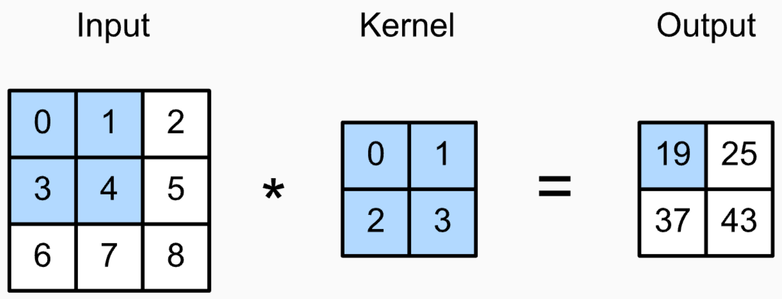


Fig. 7.2.1 Two-dimensional cross-correlation operation. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation.

Feature Map & Receptive Field

- Receptive fields derive their name from neurophysiology.
- A series of experiments on a range of animals using different stimuli (Hubel and Wiesel, 1959~1968) explored the response of what is called the visual cortex on said stimuli.
- By and large they found that lower levels respond to edges and related shapes.
- Later on, Field (1987) illustrated this effect on natural images with, what can only be called, convolutional kernels.
- We reprint a key figure in Fig. 7.2.2 to illustrate the striking similarities.

Feature Map & Receptive Field

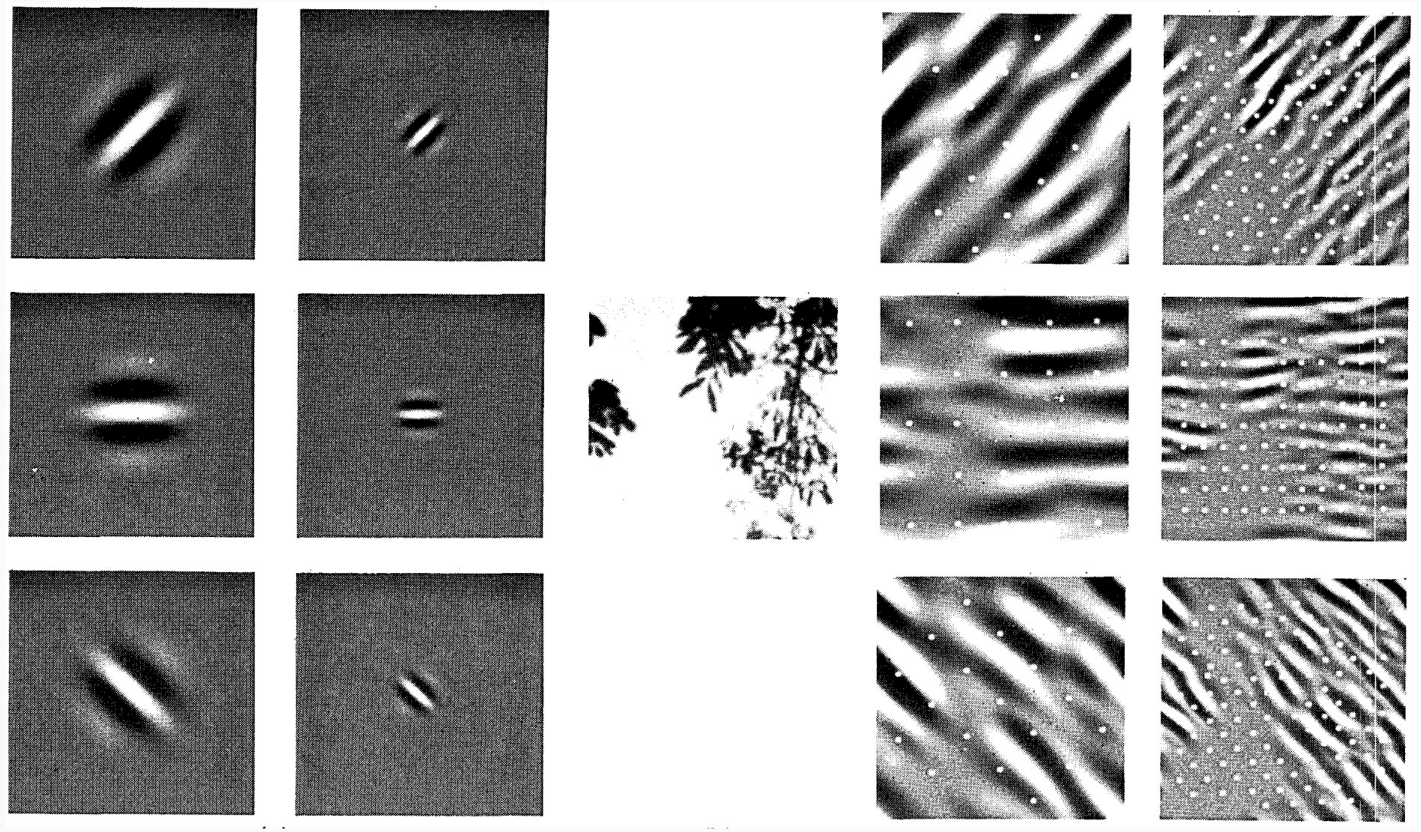


Fig. 7.2.2 Figure and caption taken from Field (1987): An example of coding with six different channels. (Left) Examples of the six types of sensor associated with each channel. (Right) Convolution of the image in (Middle) with the six sensors shown in (Left). The response of the individual sensors is 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.

Summary

- Convolutional layers primarily use cross-correlation operations, efficiently handled with simple for-loops and suitable for multiple input/output channels.
- This computation is straightforward and localized, enabling significant hardware optimization for advancements in computer vision.
- The focus on speed over memory in chip design specifically benefits convolution optimization, though it may not suit all applications.
- Convolutions serve various purposes like edge detection and image processing, learning filters from data instead of manual feature engineering.
- The similarity of convolutional filters to brain's receptive fields suggests alignment with natural neural processing, bolstering deep network design.