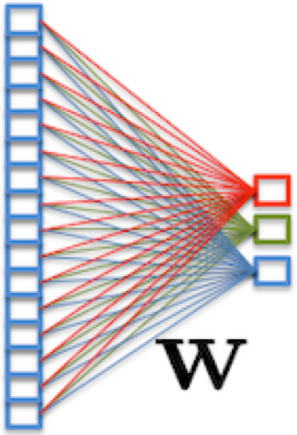
*5IF - Deep Learning and Differentiable Programming*

# 2.7 Shift equi/invariance and convolutions




INSA LYON Christian Wolf

# Fully connected layers might be harmful

400x400 pixels,
3 color channels

Flattened into
480 000 values

500 units

*500 * 480 000 =*
*240 million parameters (for a single layer!)*

# Fully connected layers might be harmful

Parameters learned for one part of the image do not generalize to other parts of the image.

# Shift invariance

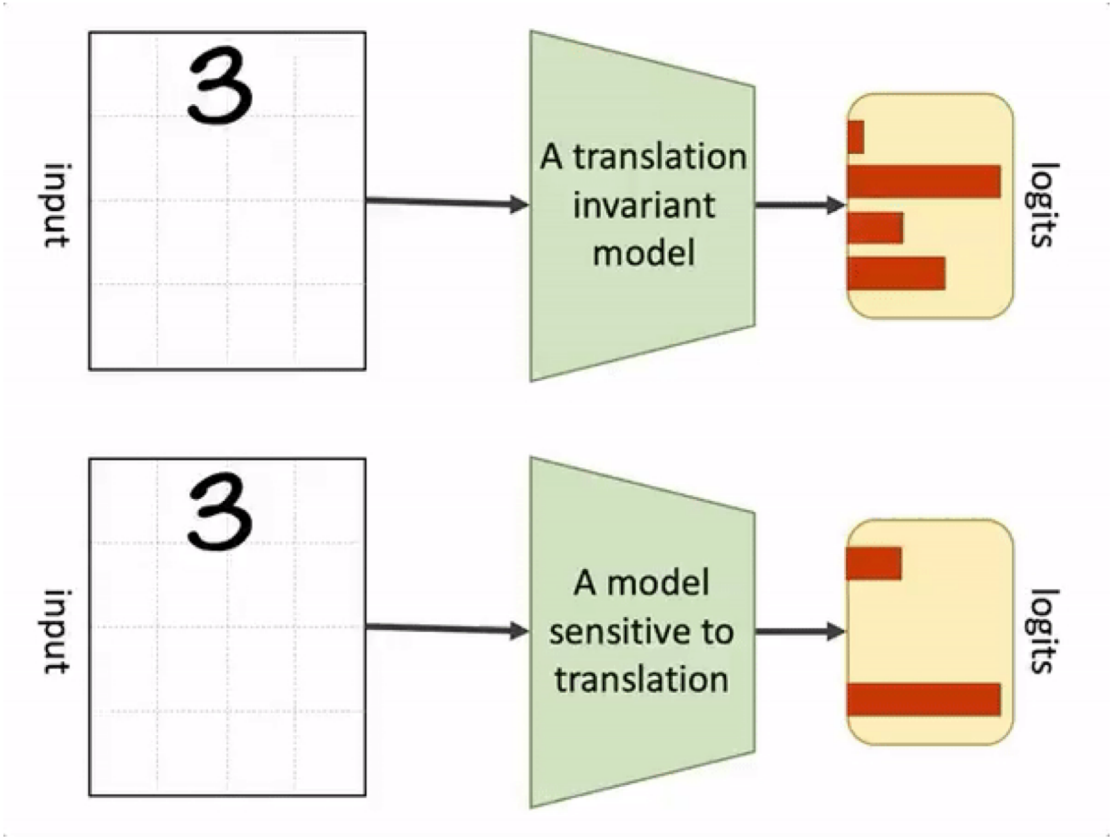
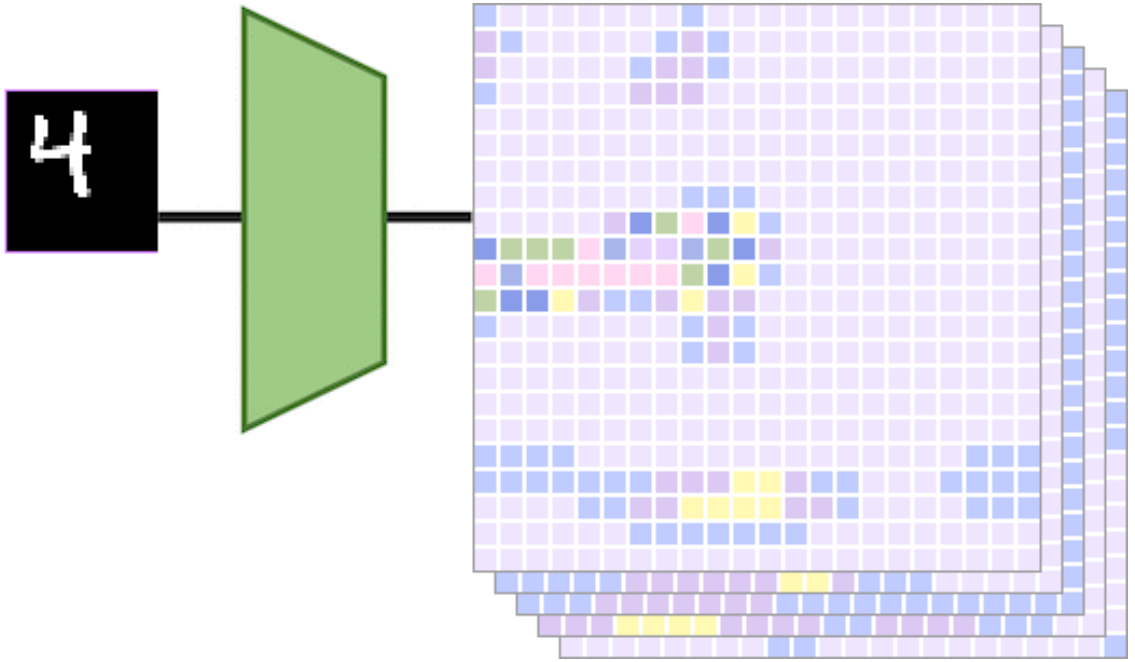


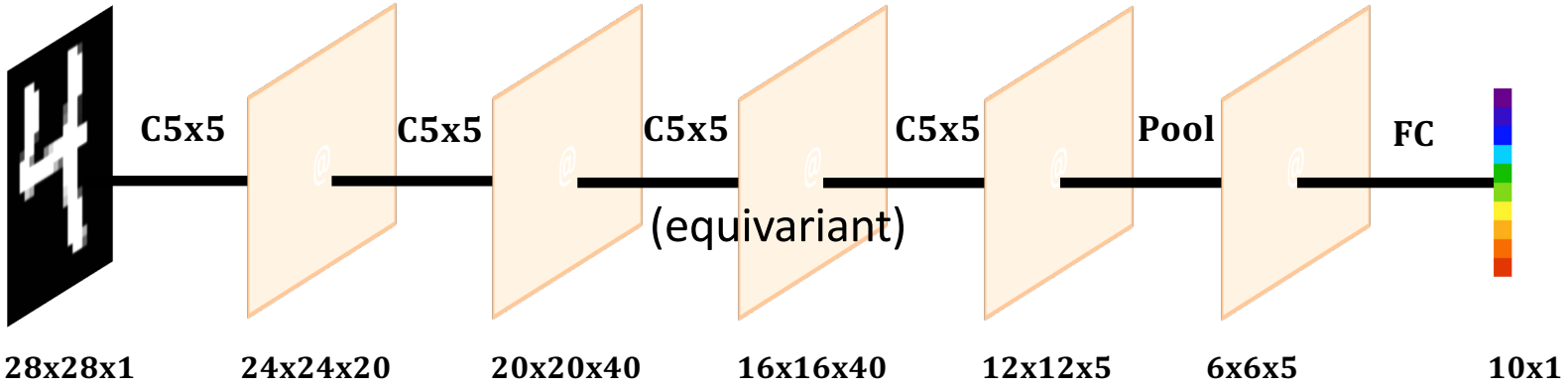

Figure by Samira Abnar (U. of Amsterdam)
https://samiraabnar.github.io/

# Translation equivariance

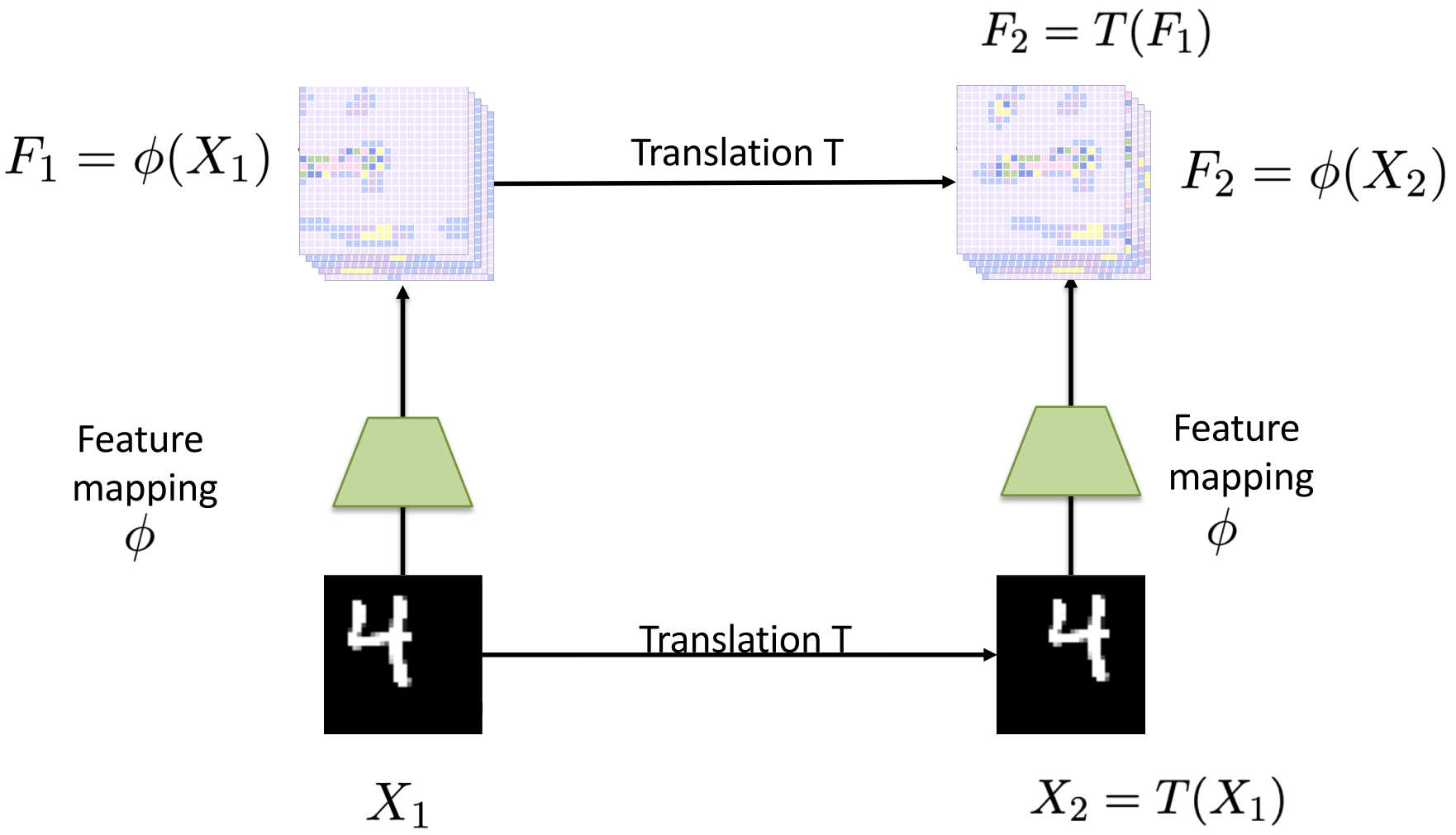
An equivariant mapping preserves the algebric structure of the transformation.



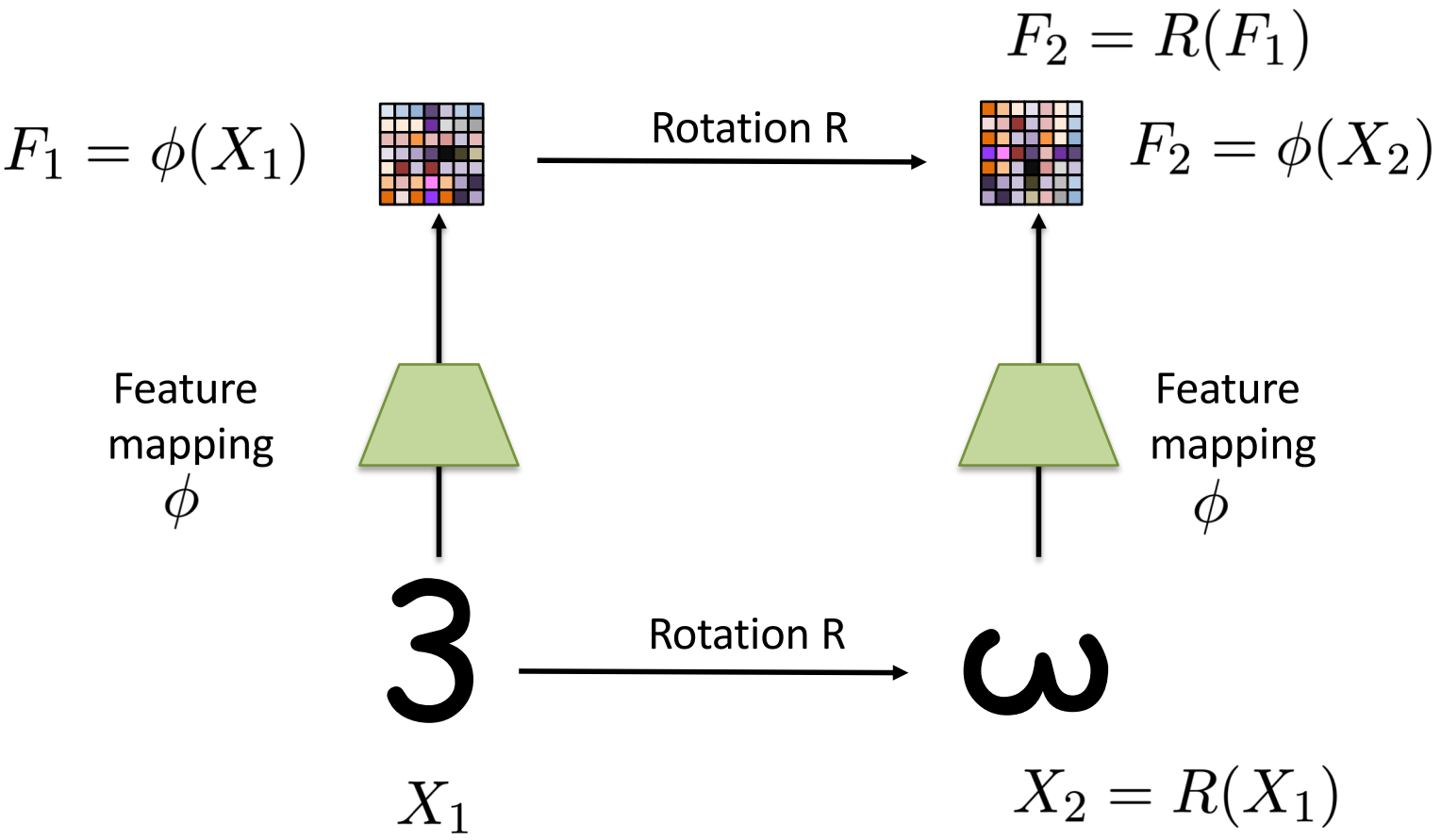
# Translation Equivariance



# Translation Equivariance



# Rotation Equivariance



# Shift-equivariant linear operations

Classical MLPs are sequences of linear layers followed by point-wise non-linearities. What kind of linear and shift-equivariant operators can there exist?

Let's consider an operator $\varnothing$ with following properties:

**Linearity**:

$$\phi\left(\alpha f + \beta f'\right) = \alpha\phi(f) + \beta\phi\left(f'\right)$$

**Shift-equivariance**:

$$\phi\left({}^{m,n}S(f)\right) = {}^{m,n}\,S(\phi(f))$$

where ${}^{m,n}S(f)$ shifts a signal by ${}^{m,n}$.

**Impulse response** $h$:

$$h = \phi({}^{0,0}p)$$

where ${}^{0,0}p$ is a Dirac impulse centered at $0, 0$.

# Shift-equivariant linear operations

We decompose the signal $f$ into a series of Diracs ${}^{m,n}p$:

$$[\phi(f)](x,y) = \left[\phi\left(\sum_{m=-M/2}^{M/2}\sum_{n=-N/2}^{N/2} f(m,n)\,{}^{m,n}p\right)\right](x,y)$$

*Dirac at position m,n*

We use **linearity**:

$$= \left[\sum_{m=-M/2}^{M/2}\sum_{n=-N/2}^{N/2} f(m,n)\,\phi\left({}^{m,n}p\right)\right](x,y)$$

We use **shift-equivariance**:

$$= \left[\sum_{m=-M/2}^{M/2}\sum_{n=-N/2}^{N/2} f(m,n)\,\phi\left({}^{m,n}S\left({}^{0,0}p\right)\right)\right](x,y)$$

We use **linearity** again:

$$= \left[\sum_{m=-M/2}^{M/2}\sum_{n=-N/2}^{N/2} f(m,n)\,{}^{m,n}S\left(\phi\left({}^{0,0}p\right)\right)\right](x,y)$$

# Shift-equivariant linear operations

Change in notation: we replace $\phi(^{0,0}p)$ by $h$ (through definition):

$$[\phi(f)](x,y) = \left[ \sum_{m=-M/2}^{M/2} \sum_{n=-N/2}^{N/2} f(m,n)^{m,n} S(h) \right] (x,y)$$

Change in notation: make the shift-operator explicit:

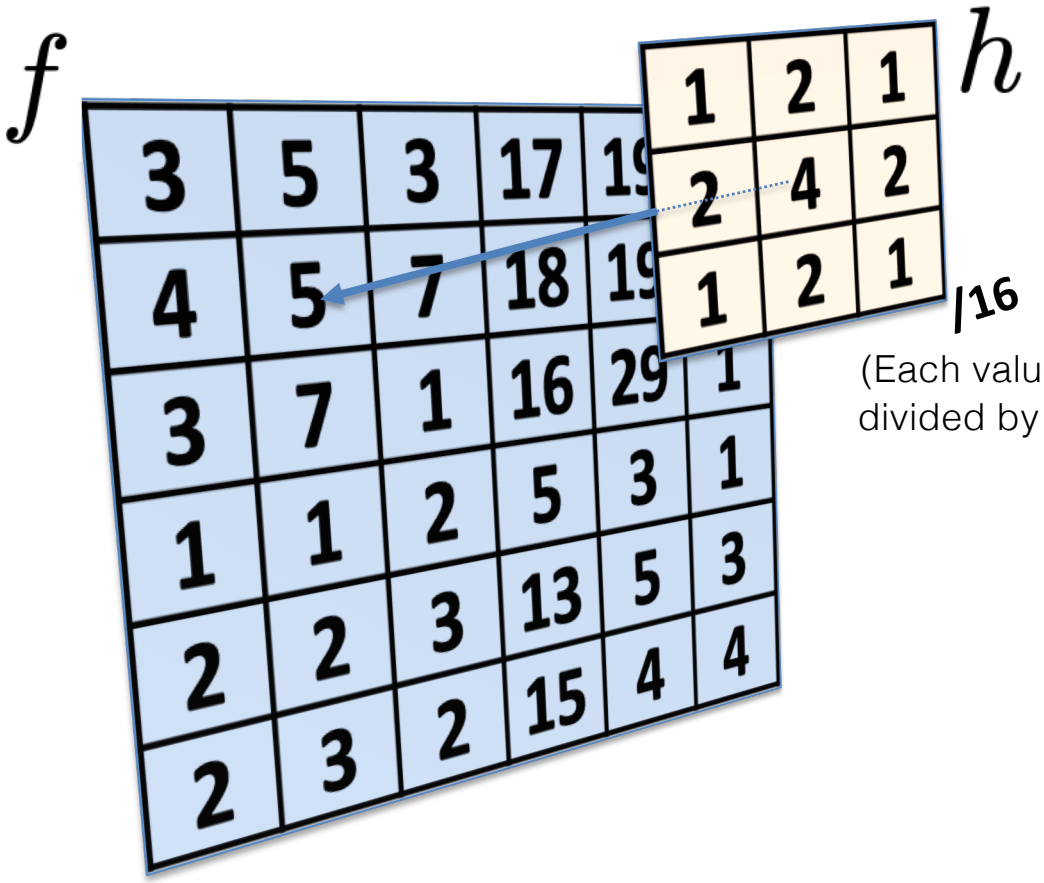
$$= \sum_{m=-M/2}^{M/2} \sum_{n=-N/2}^{N/2} f(m,n) h(x-m, y-n)$$

Change of variables:

$$= \sum_{m'=-M/2}^{M/2} \sum_{n'=-N/2}^{N/2} f\left(x-m', y-n'\right) h\left(m', n'\right)$$

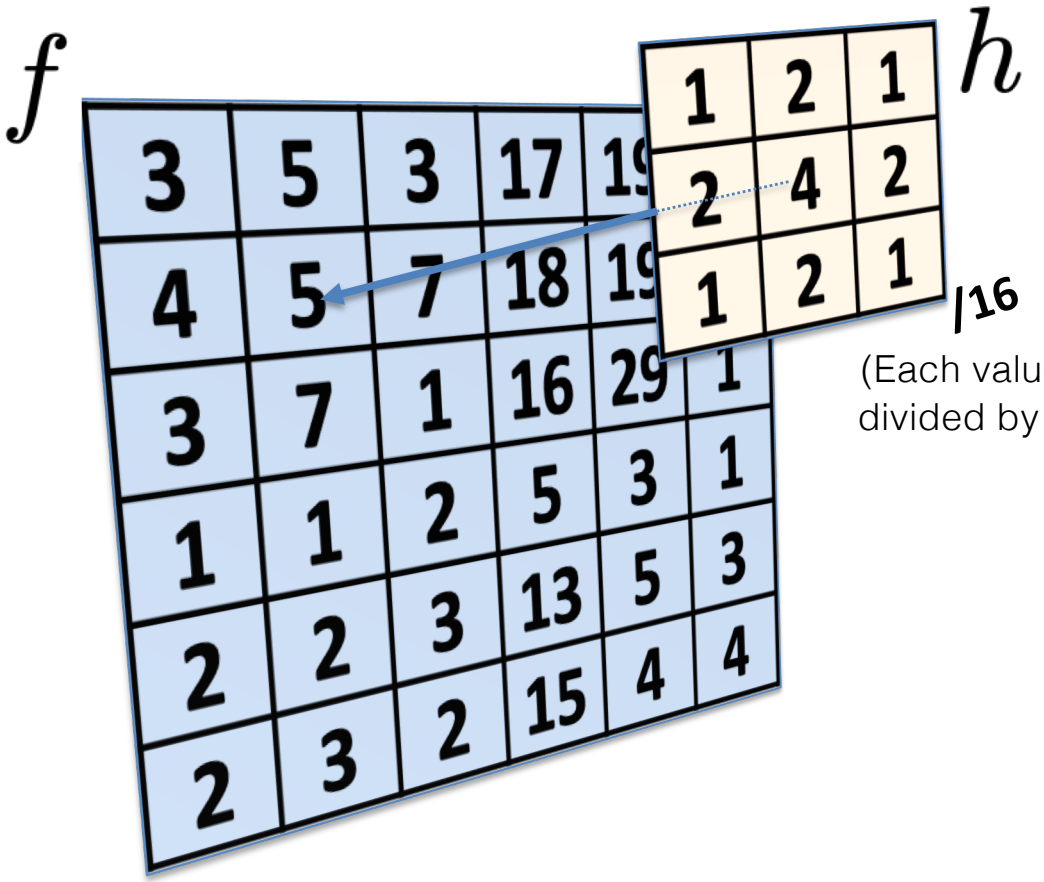
$(m' = x - m, n' = y - n) \Rightarrow$ we get a convolution!

# Convolutions

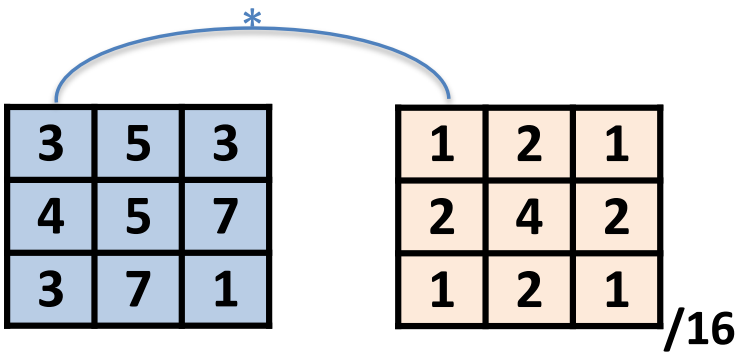



(Each value is divided by 16)

# Convolutions

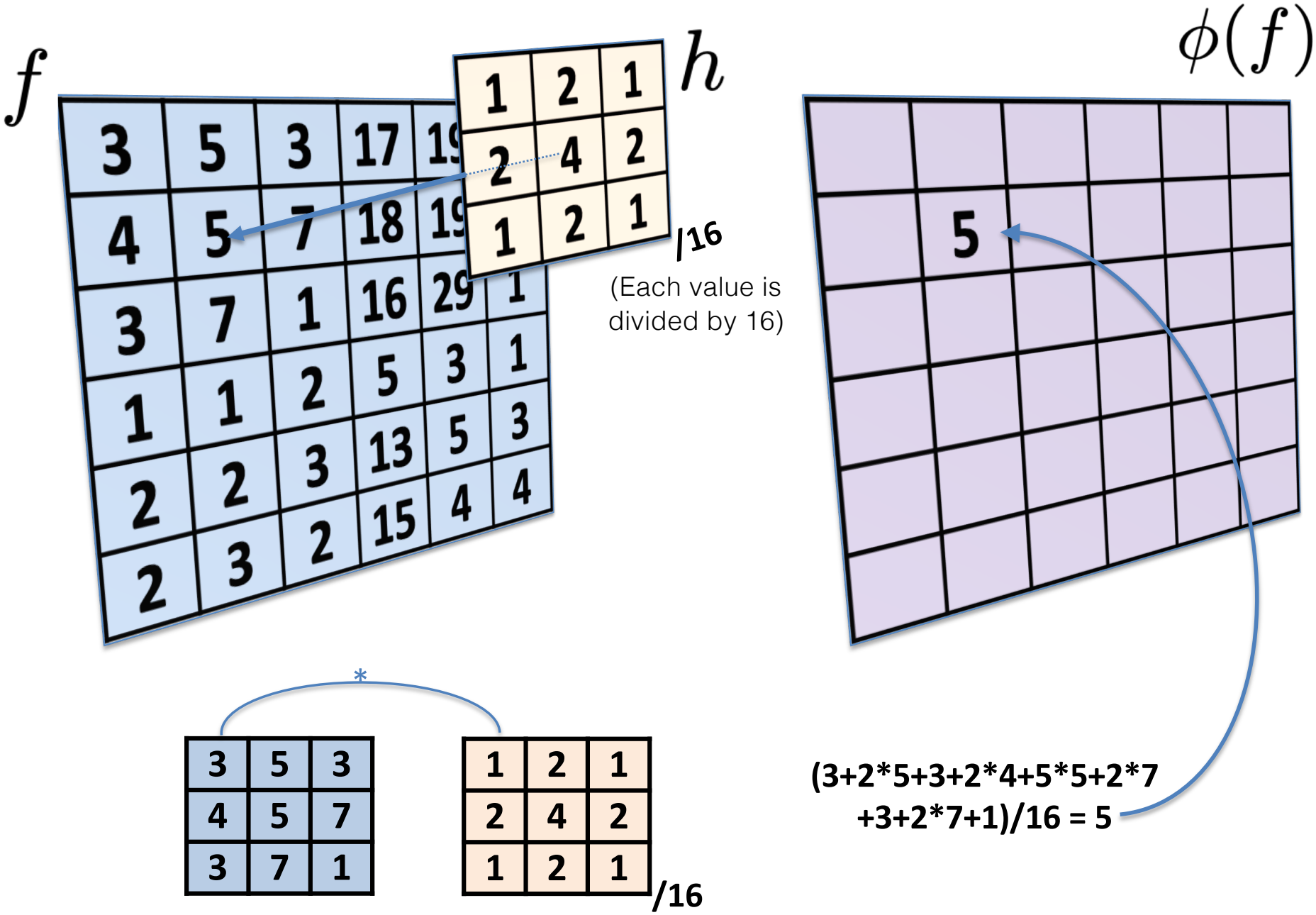



(Each value is divided by 16)

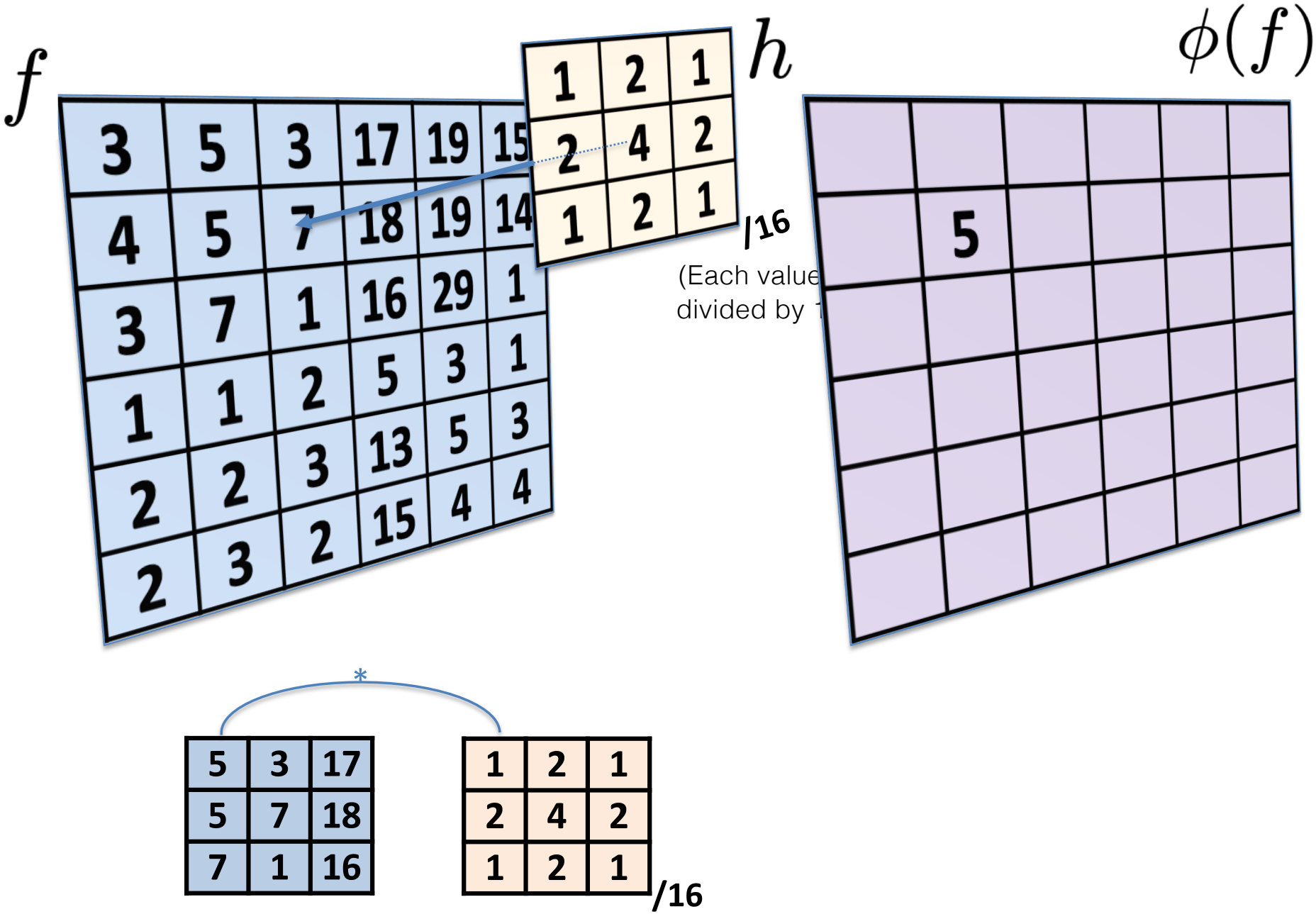



**(3+2*5+3+2*4+5*5+2*7 +3+2*7+1)/16 = 5**

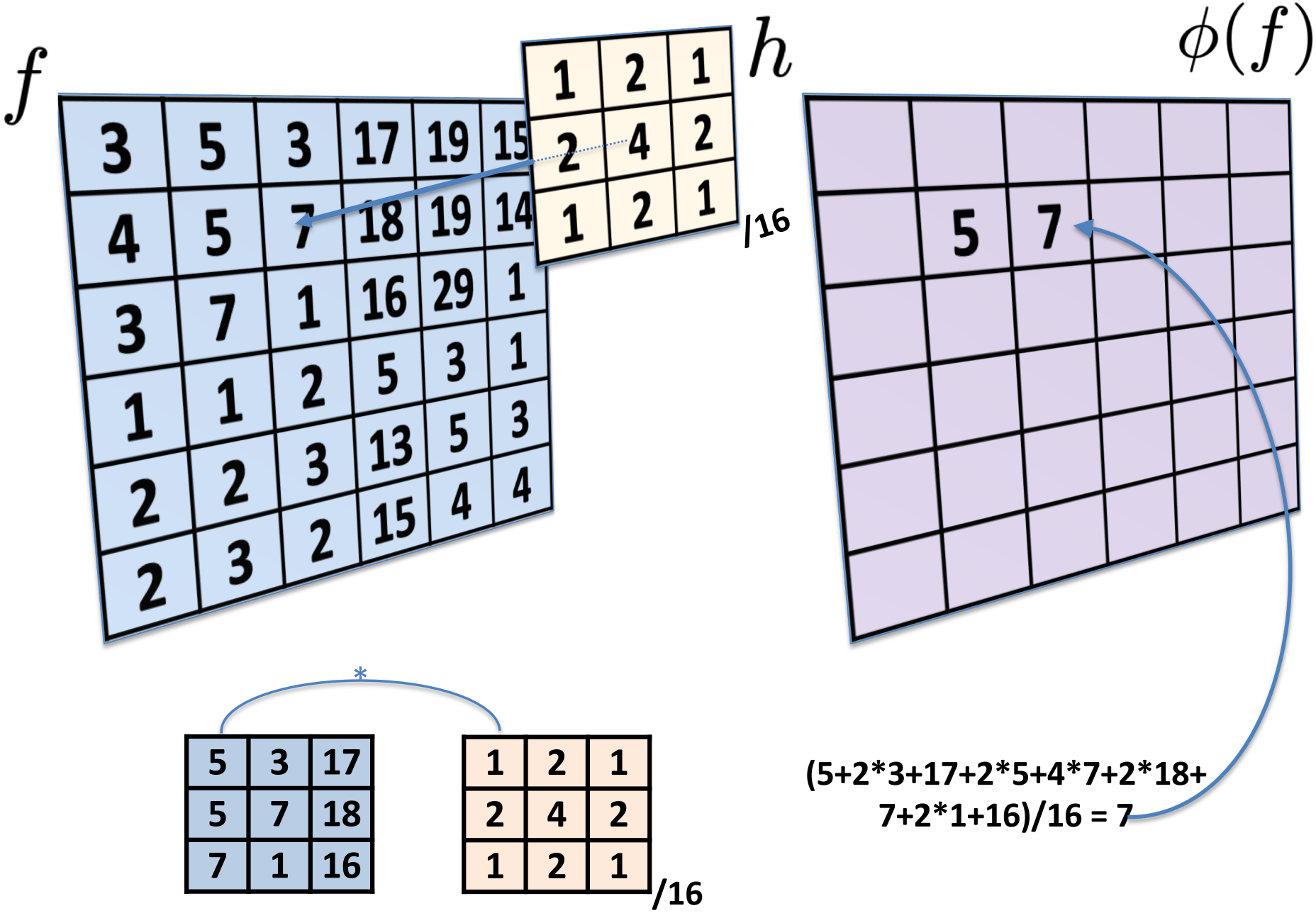
# Convolutions



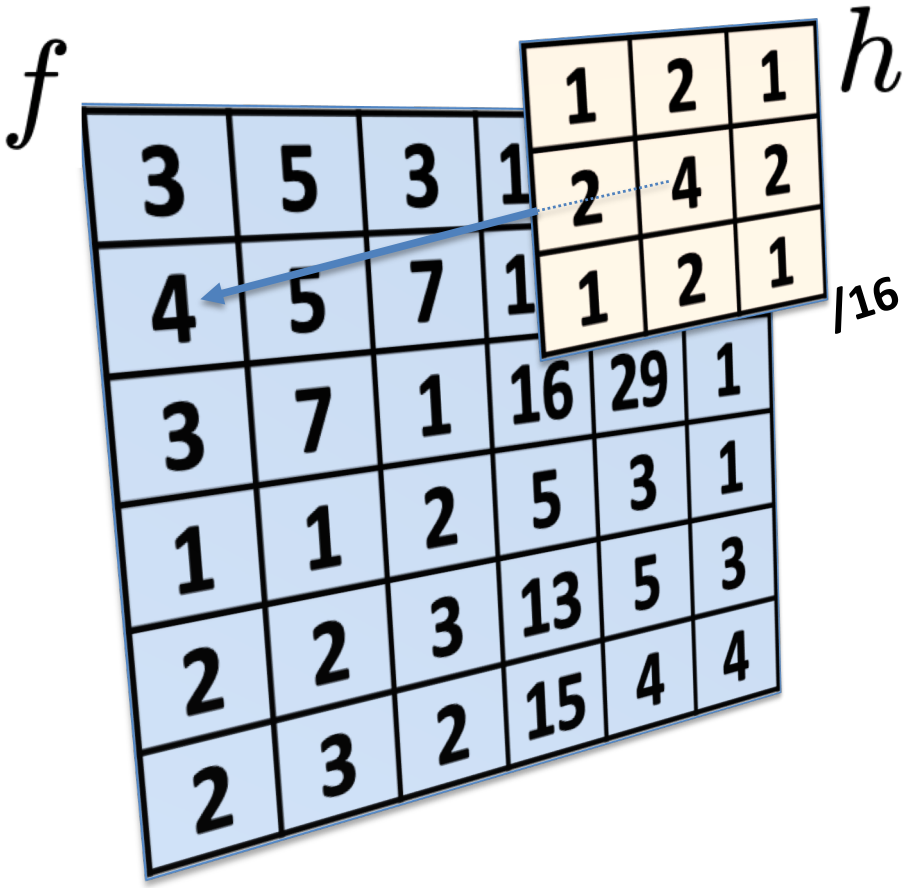
# Convolutions



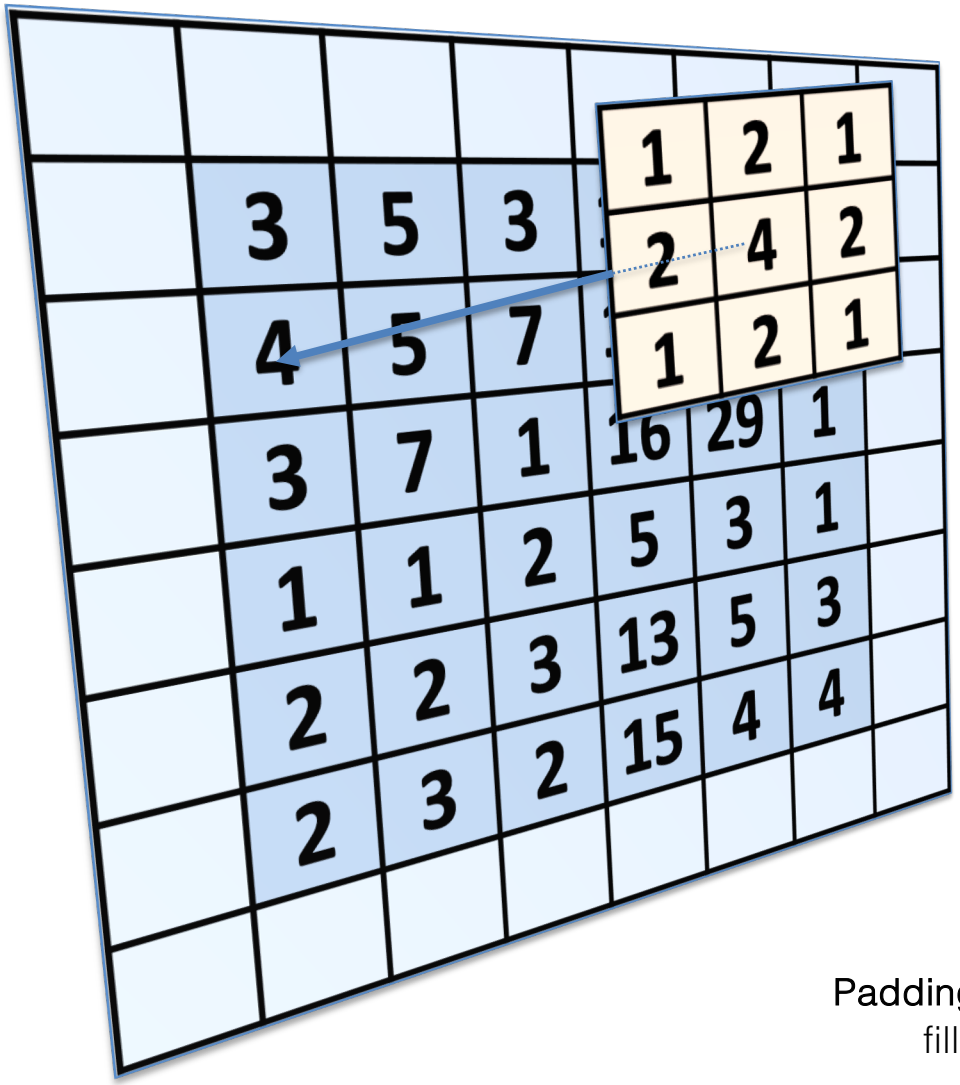
# Convolutions



# What happens on the borders?



# Padding




**Padding:** create a virtual border filled with zero values

# « Handcrafted « convolutions

Convolutions are standard operators in signal processing and image processing. In applications other than machine learning, their parameters (filter values) are not necessarily learned, but handcrafted.

Specifically chosen filters are frequently used for:

- Noise reduction
- Blurring / effect creation
- Deriving a signal
- Signal reconstruction (inverse filtering, Wiener filter)

# The average filter

$$g = \frac{1}{4}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad g = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$F(x,y)=f \otimes g \Leftrightarrow$ $$F(x,y) = \frac{1}{9}\sum_{u=-1}^{u=1}\sum_{v=-1}^{v=1} f(x+v, y+u)$$

# Increasing the filter size

$$g = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Bigger kernel = more blur

Computational complexity!

# Gaussian filter

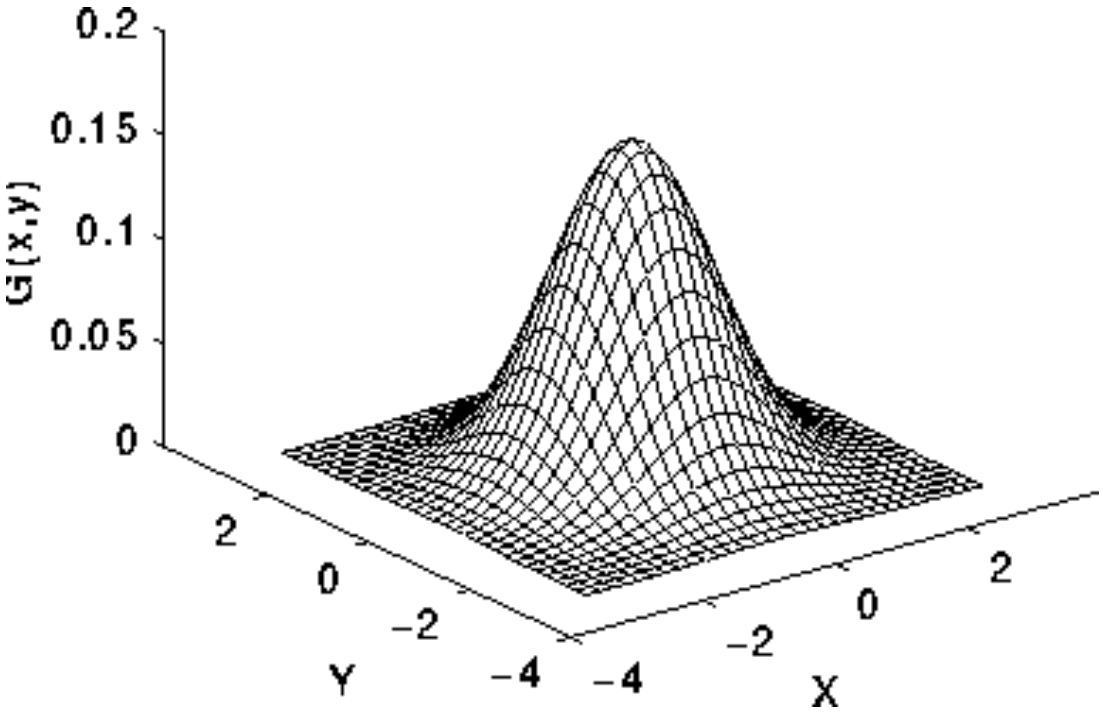
Discrete approximation of a 2D Gaussian filter:

**Weighted** mean, the center pixels are more important than the border pixels

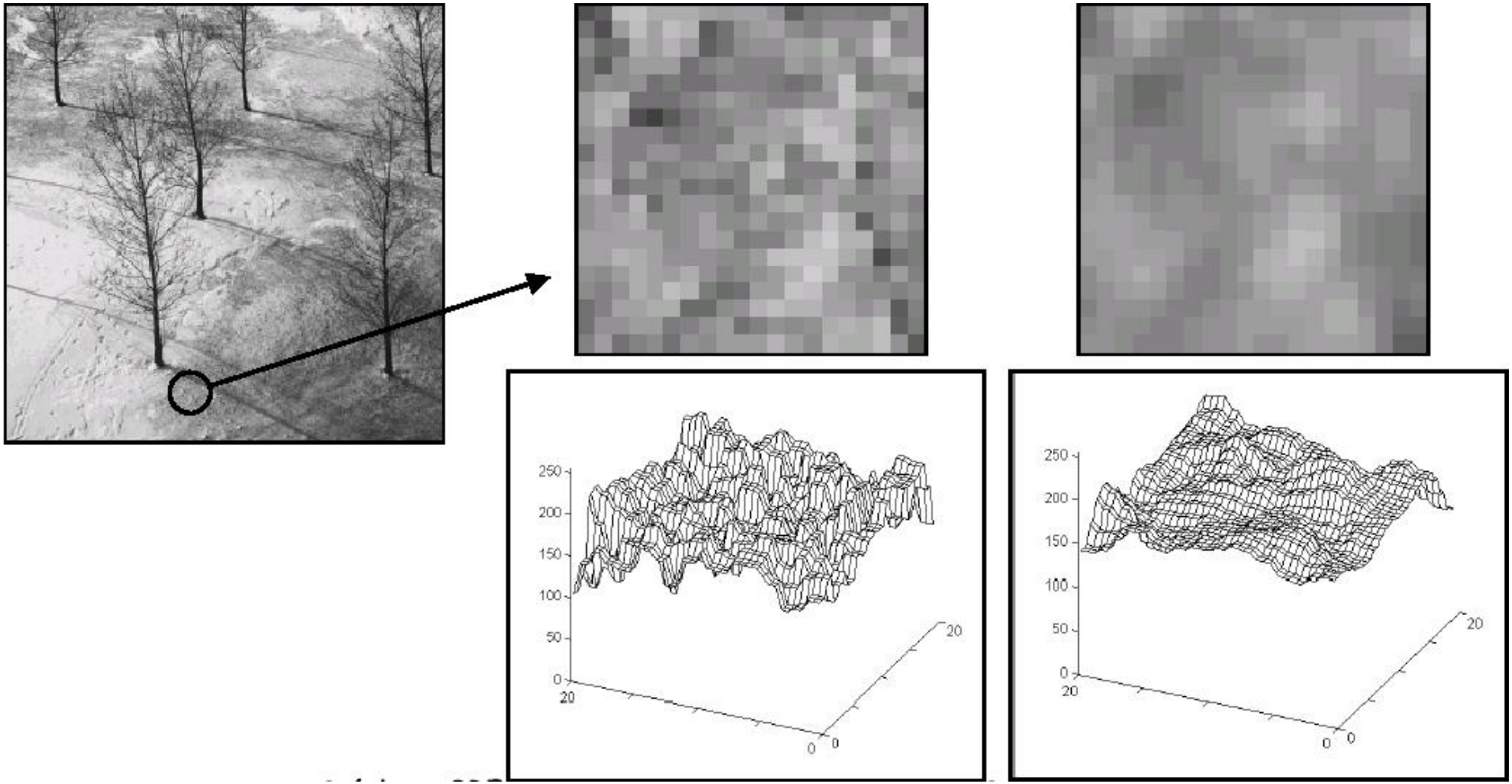
$$G(x,y) = \frac{1}{2\pi\sigma_x\sigma_y} e^{-\frac{1}{2}\left(\frac{x^2}{\sigma_x} + \frac{y^2}{\sigma_y}\right)}$$

$$g = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

$$g_5 = \frac{1}{246}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$



# Gaussian low pass filter: details

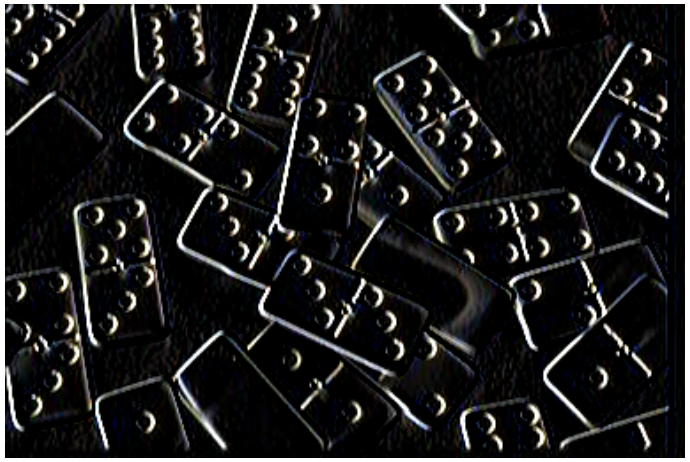


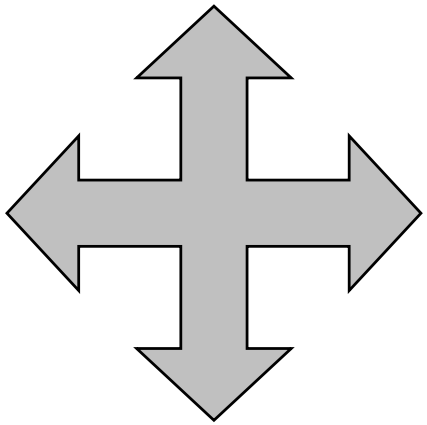
# Derive a signal: the Roberts filter

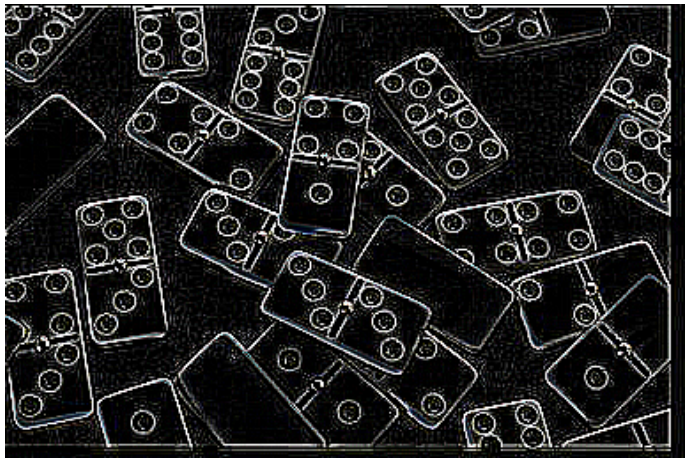
$$\frac{\partial f}{\partial x} \cong f \otimes \begin{bmatrix} 1 & -1 \end{bmatrix} \qquad \frac{\partial f}{\partial y} \cong f \otimes \begin{bmatrix} 1 \\ -1 \end{bmatrix} \qquad \frac{\partial f}{\partial \vec{v}} \cong f \otimes \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$\frac{\partial f}{\partial x} \cong f \otimes \begin{bmatrix} 1 & -1 \end{bmatrix}$$

$$\frac{\partial f}{\partial y} \cong f \otimes \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$







$$\|\nabla f\| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

# Derive a signal: the Sobel filter

$$\frac{\partial f}{\partial x} \cong f \otimes \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad \frac{\partial f}{\partial y} \cong f \otimes \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$\frac{\partial f}{\partial v1} \cong f \otimes \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix} \qquad \frac{\partial f}{\partial v2} \cong f \otimes \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}$$

# Sharpen

Sharpened image = (A-1) original image + high pass filtered image

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Laplacian filter (high pass filter)

A=1: standard Laplacian filter
A>1: a part of the original image is added to the high pass

A=2

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Noise!!!

# Iterative application

Iteratively convolving an image N times by a filter of size M corresponds to the application of a single convolution of a filter of size M* = 2 × ( (M-1) / 2 ) × N ) +1

**2** x a filter of size **3** = 1 x filter of size 5

**3** x a filter of size **3** = 1 x filter of size **7**

$$(f \otimes h) \otimes g = f \otimes (h \otimes g)$$

$$2 \times \quad g_3 = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad \Leftrightarrow \quad g_5 = \frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

# Convolutions in PyTorch

```
import torch
from torch.nn import functional as F

# Create a binary random tensor 5x5 tensor
# don't forget batch & channel dimensions
f = torch.rand((1,1,5,5))
f[f>0.5]=1
f[f<0.5]=0
print (f)
```

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

```
# Create an average filter (all ones)
h = torch.ones((1,1,3,3))
print (h)
```

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

# Convolutions in PyTorch

```
print (f)
```

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

```
# Convolve
F.conv2d (f, h)
```

```
tensor([[[[4., 3., 2.],
          [1., 2., 2.],
          [2., 3., 3.]]]])
```

The output is smaller: we lost 2 border rows and 2 border columns because we did not explicitely state that we want padding.

# Convolutions in PyTorch

```
print (f)
```

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

```
# Convolve with padding
F.conv2d (f, h, padding=1)
```

```
tensor([[[[2., 3., 2., 1., 0.],
          [2., 4., 3., 2., 0.],
          [0., 1., 2., 2., 1.],
          [0., 2., 3., 3., 1.],
          [0., 1., 2., 2., 1.]]]])
```

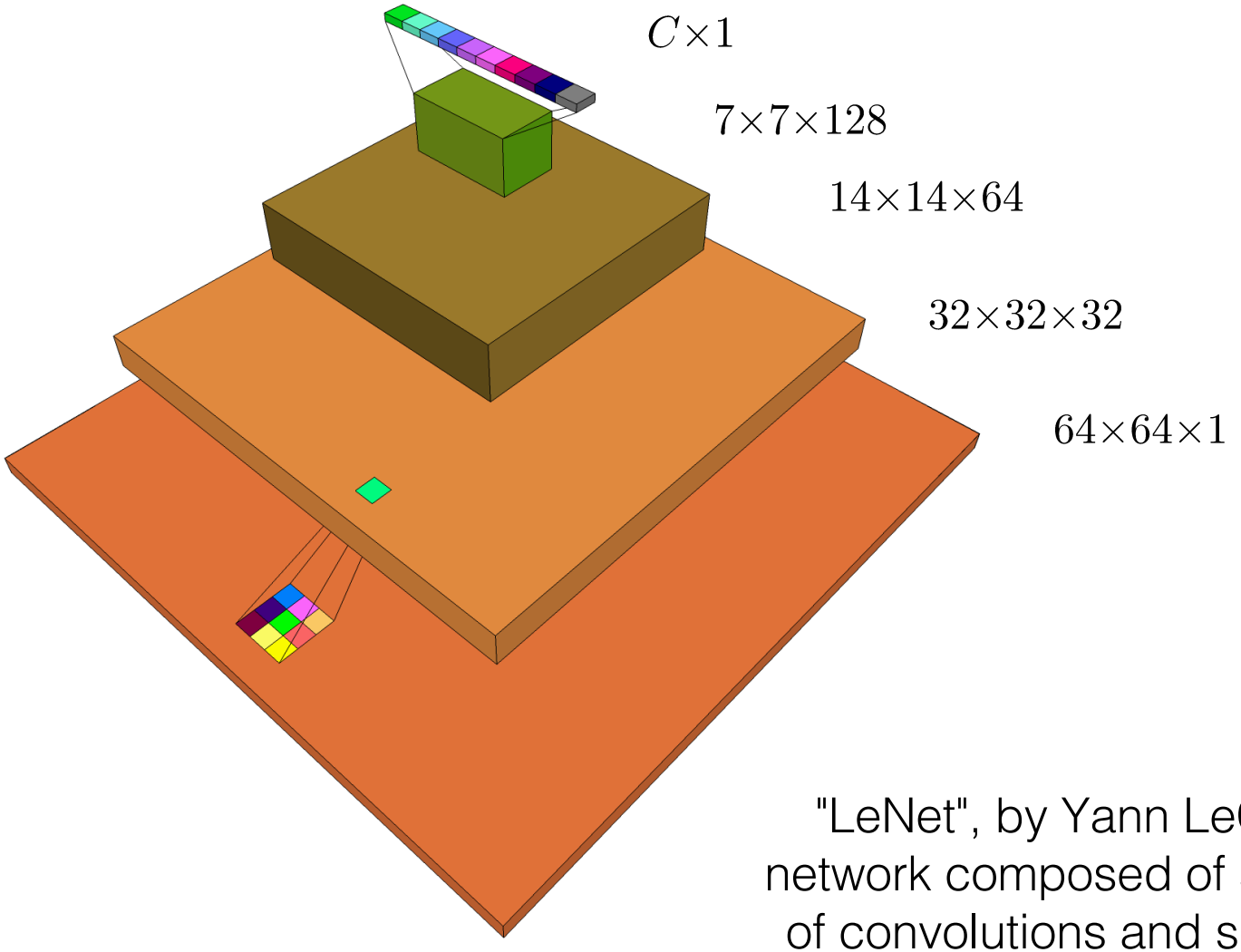
# Learned convolutions

In Deep Learning, we use convolutions in the form of convolutional layers in neural networks.

They structure neural networks (“inductive bias"):

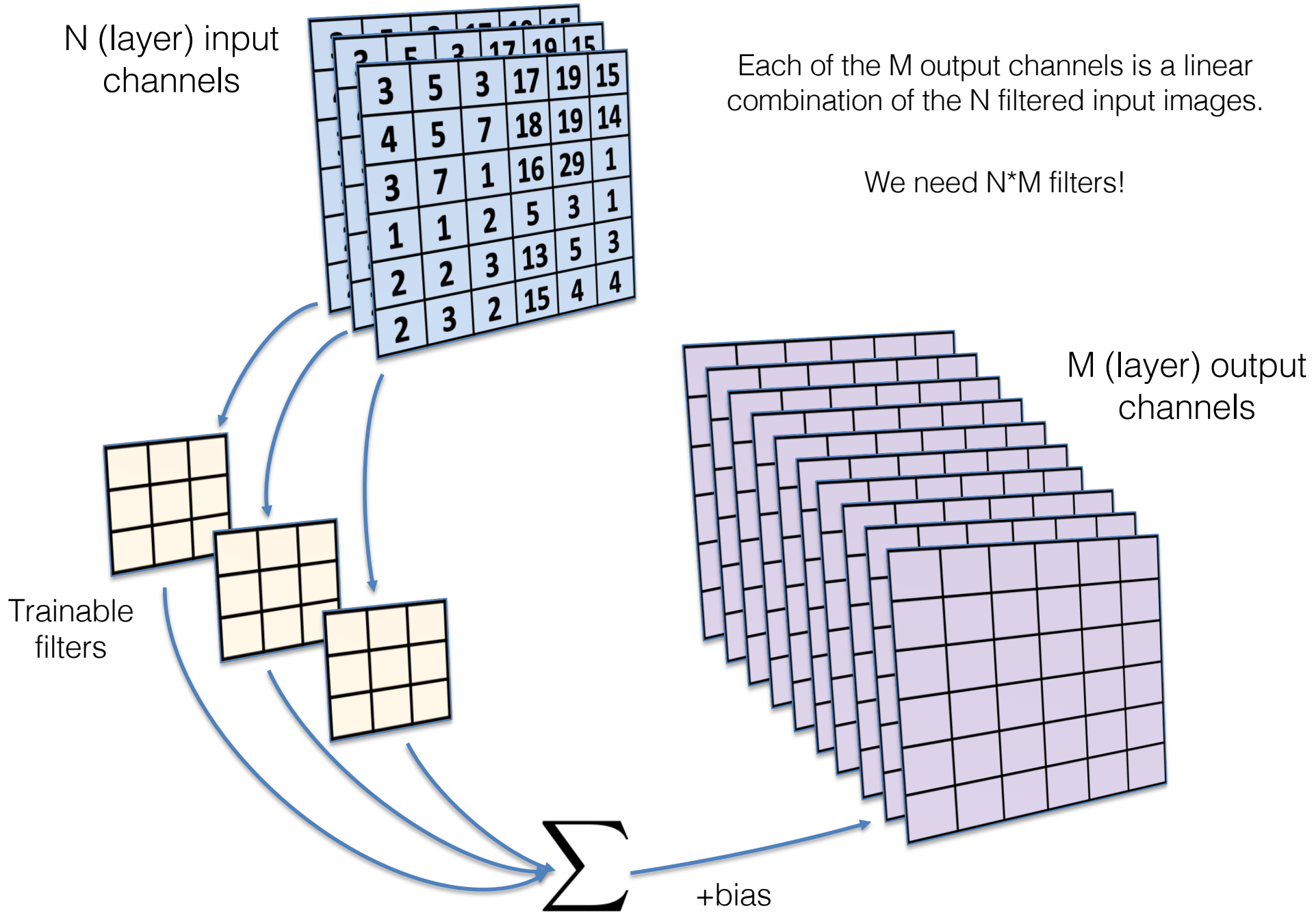
- Input layers can be multidimensional tensors (2D images, 3D videos) and are (usually) **not** flattened into vectors.
- Activations keep the structure: hidden layers can be spatially, temporally, or spatio-temporally structured.

# « LeNet »




"LeNet", by Yann LeCun, is a network composed of sequences of convolutions and spatial size reductions (« pooling »).

[LeCun et al., *1998*]


N (layer) input channels
3 5 3 17 19 15
4 5 7 18 19 14
3 7 1 16 29 1
1 1 2 5 3 1
2 2 3 13 5 3
2 3 2 15 4 4
Each of the M output channels is a linear combination of the N filtered input images.
We need N*M filters!
M (layer) output channels
Trainable filters
Σ
+bias

# Or, in equations ...

In matrix/vector notation:

$$y_i = \sum_j f_j * h_{ij} + b_i$$

| | |
|---|---|
| $*$ | is the convolution operation |
| $y_i$ | is output channel $i$ (a 2D tensor for each sample of the batch) |
| $h_{ij}$ | is filter $j$ for output channel $i$ (a 2D tensor for each sample of the batch) |
| $b$ | is a bias vector (one value for each output channel) |

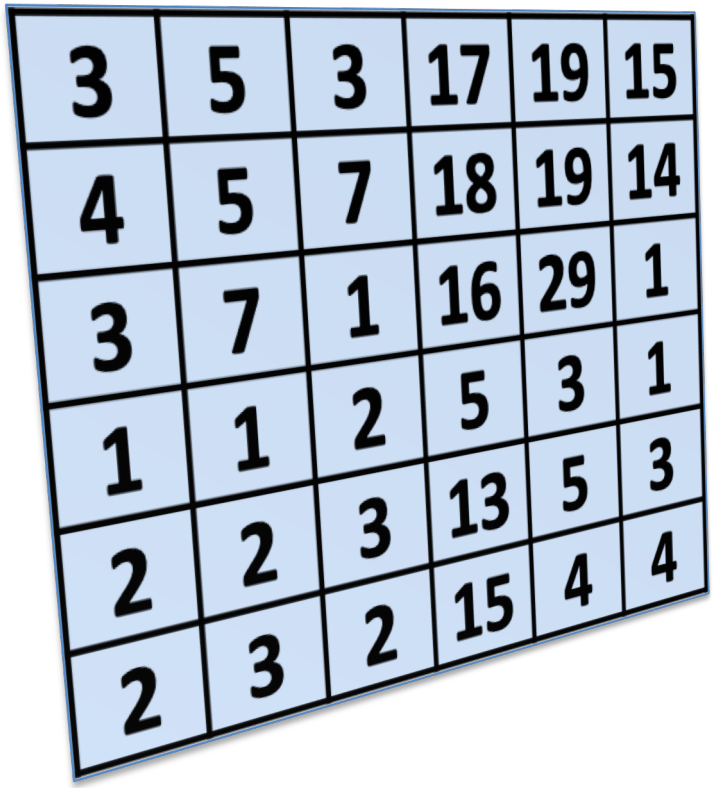
# Summary/recall: tensor dimensions

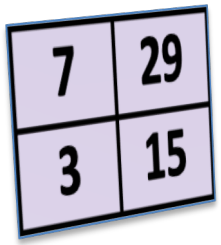
**Recall:** PyTorch functions operate on multi-dimensional tensors and follow conventions on the order of dimensions.

- The first dimension is the batch dimension
  - Use 1 if you don't use batches (= batches of size 1).
  - Losses are reduced (sum or mean) over samples in a batch
  - We don't convolve over batches!
- the second dimension is the channel dimension
  - Use 1 if you don't use channels (= single channels).
  - We don't convolve over channels!
  - Each output channel receives filtered responses of each input channel.
- the following dimensions are application dependant:
  - e.g. rows, columns in images.
  - Convolutions can be done over these dimensions.
  - We can convolve over sub-dimensions (eg 1D convolutions of image rows).

# Max pooling

Pooling reduces the spatial resolution of a tensor, combining multiple values into a single one.




Max pooling: take the maximum of a HxW neighborhood.


Average pooling

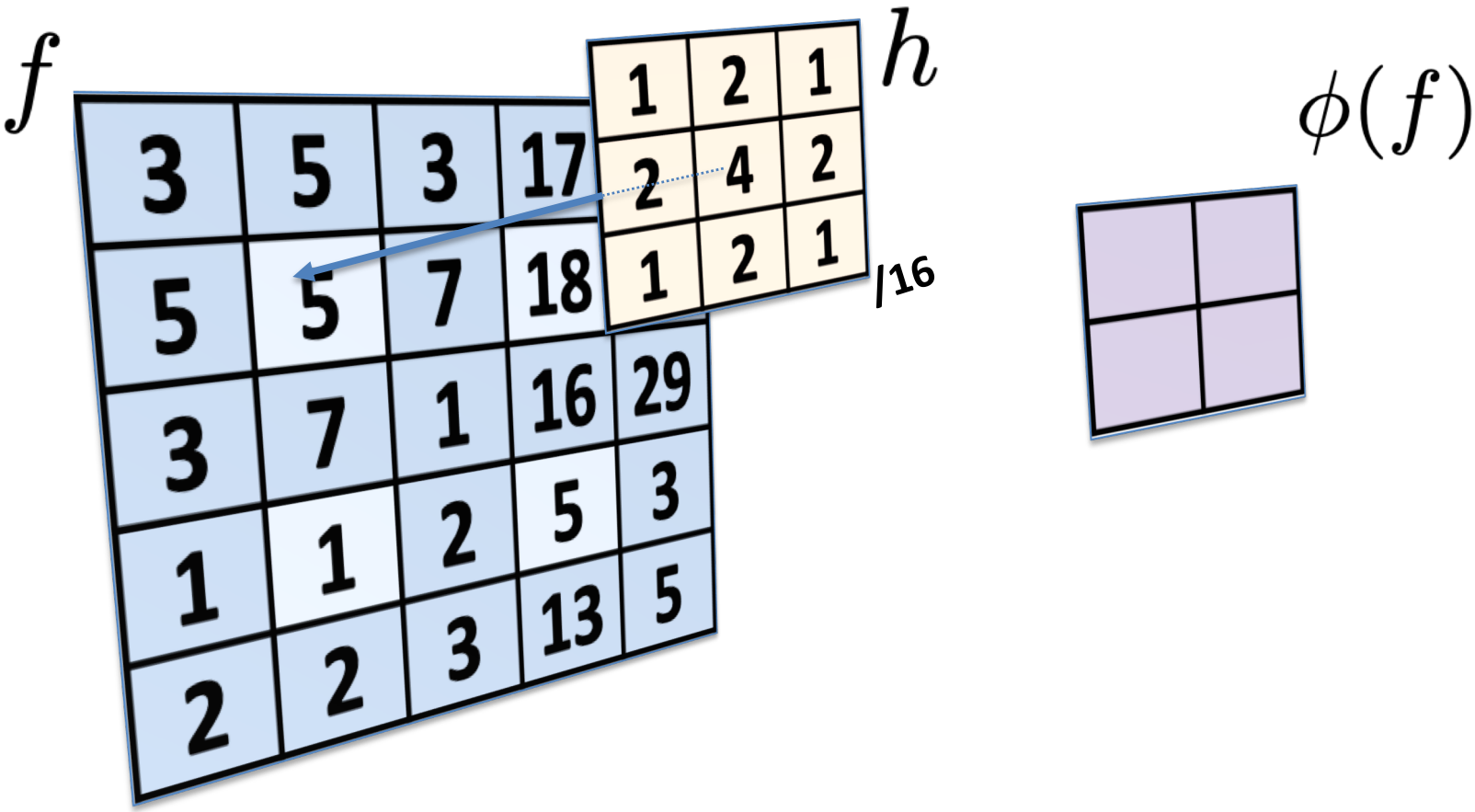
Example: kernel=3x3

# Convolutions + pooling ...

- Convolutions introduce shift-**equi**variance
- Convolutions + pooling create shift-**inv**ariant models

# Convolutions with stride

The stride parameter allows to move the kernel in multiples. Shown below: stride = 2 (and no padding).



# LeNet in PyTorch

```
class LeNet(torch.nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        # 1 input channel, 20 output channels,
        # 5x5 filters, stride=1, no padding
        self.conv1 = torch.nn.Conv2d(1, 20, 5, 1, 0)
        self.conv2 = torch.nn.Conv2d(20, 50, 5, 1, 0)
        self.fc1 = torch.nn.Linear(4*4*50, 500)
        self.fc2 = torch.nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        # Max pooling with a filter size of 2x2
        # and a stride of 2
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

# The Conv2D function

Description of the `torch.nn.Conv2d` method:

```
torch.nn.Conv2d(
  in_channels,              # number of input channels
  out_channels,             # number of input channels
  kernel_size,              # scalar=square, 5 means 5x5
                            # or tuple eg (5,3)
  stride=1,                 # stride (scalar or tuple)
  padding=0,                # padding (scalar or tuple)
  dilation=1,
  groups=1,
  bias=True,                # add bias, True or False
  padding_mode='zeros') # zeroes, replicate, reflect
```

# Conv2D: functional vs. NN layer

There are two different functions:

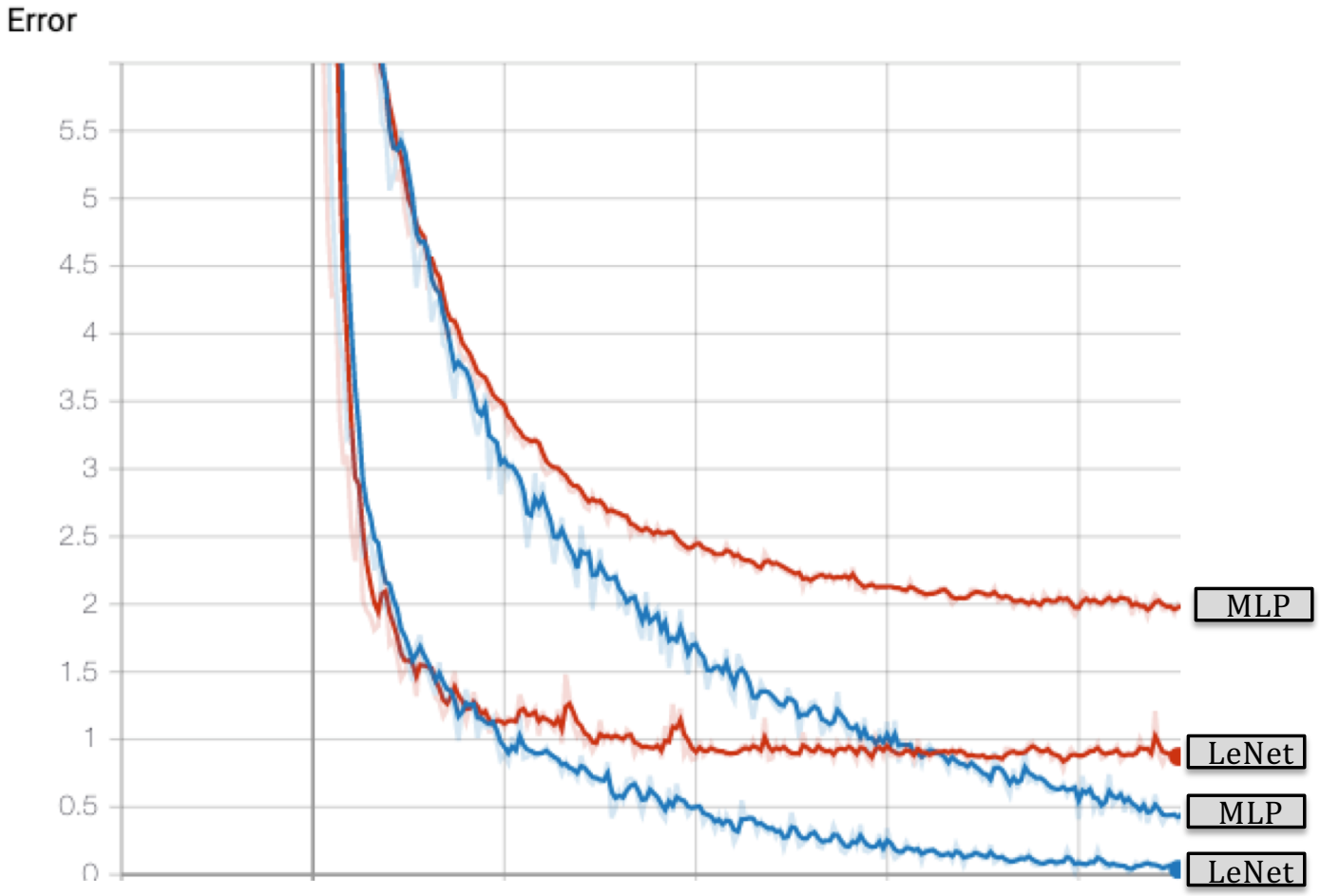
- `torch.nn.functional.conv2d` is a function which takes two tensors and returns the convolved result tensor:

```
from torch.nn import functional as F
(...)
out = F.conv2d (f, h)
```

- `torch.nn.Conv2d` is a method which creates a 2D convolution layer with trainable filters and eventually a bias. The result is the configured convolution function:

```
import torch
(...)
c = torch.nn.Conv2d(1, 20, 5, 1, 0)
out = c(f)
```

# MLP vs LeNet on MNIST




Blue: training, Red: validation

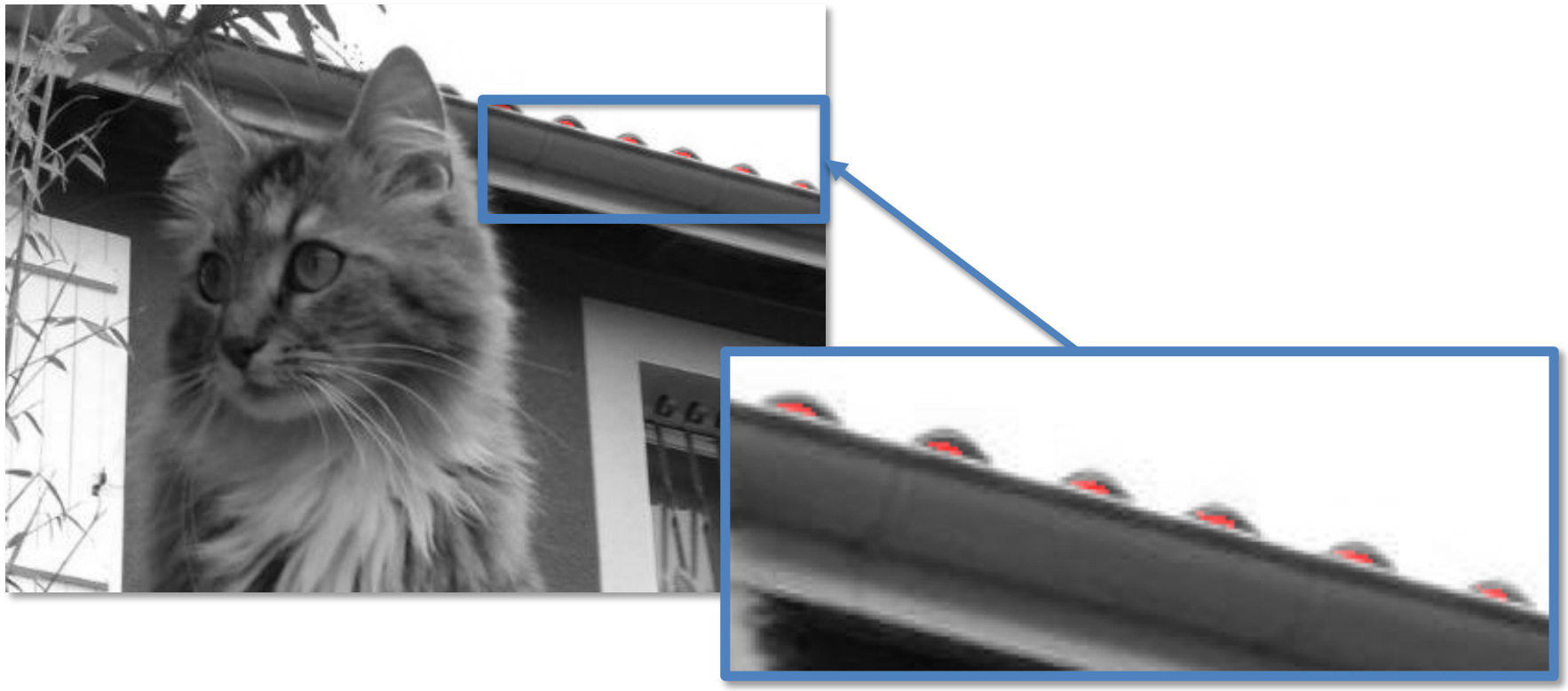
# Template matching

Input image:

Filter kernel

- We normalize input and kernel (subtract mean, divide by standard deviation).
- We convolve the input with the kernel
- We threshold response and superimpose it on the red channel.

# Template matching



Convolutions can match patterns:

- either in input space
- or in intermediate network layers (which still have a spatial organization).