Convolution and Pooling

Convolution: why?

If they were handled as normal "unstructured" vectors, large-dimension signals such as sound samples or images would require models of intractable size.

For instance a linear layer taking a 256×256 RGB image as input, and producing an image of same size would require

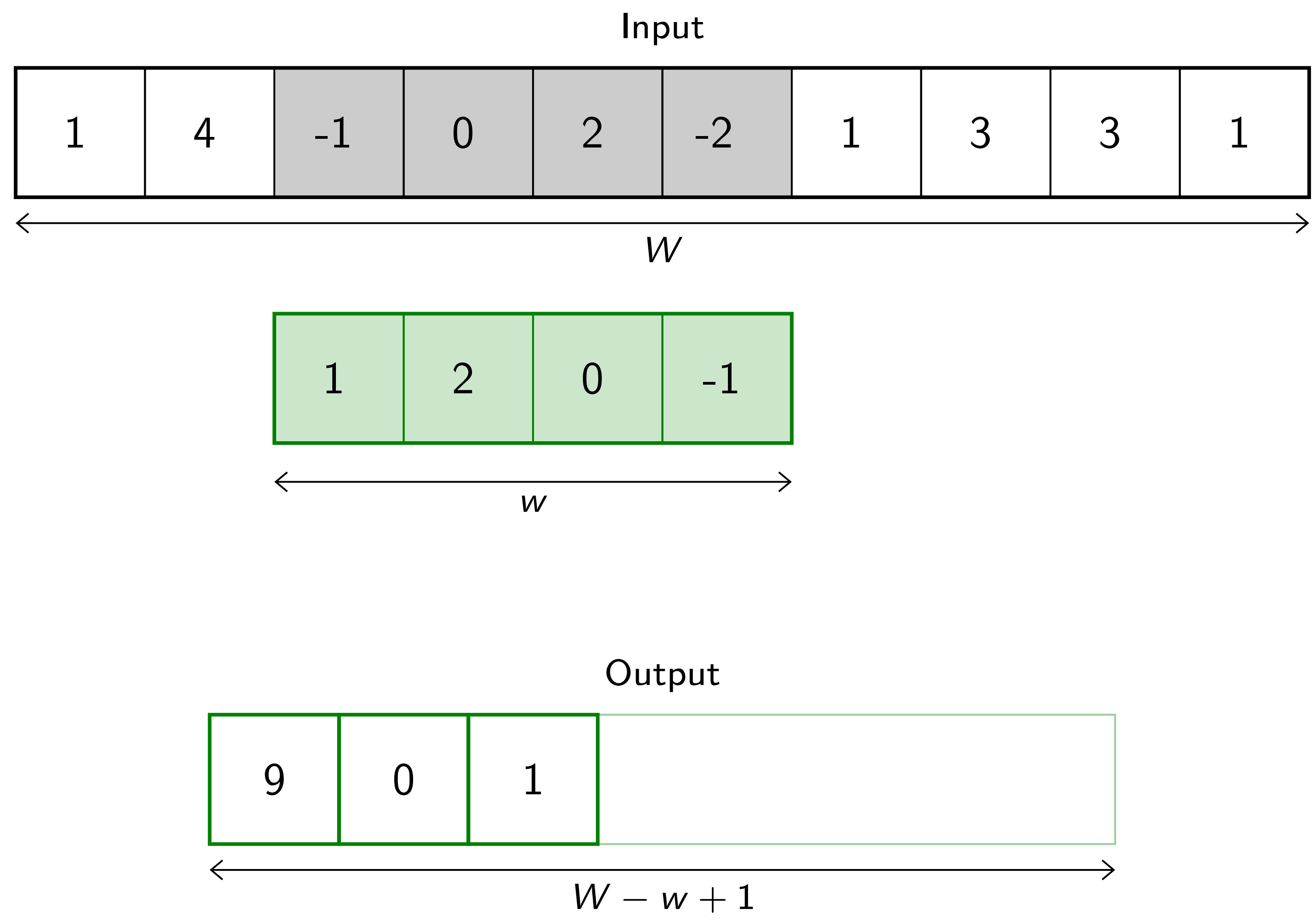
$$(256 \times 256 \times 3)^2 \simeq 3.87e{+}10$$

parameters, with the corresponding memory footprint ($\simeq$150Gb !), and excess of capacity.

Moreover, this requirement is inconsistent with the intuition that such large signals have some "invariance in translation". **A representation meaningful at a certain location can / should be used everywhere.**

A convolution layer embodies this idea. It applies the same linear transformation locally, everywhere, and preserves the signal structure.

Convolution



Convolution

Formally, in 1d, given

$$x = (x_1, \dots, x_W)$$

and a "convolution kernel" (or "filter") of width w

$$u = (u_1, \dots, u_w)$$

the convolution $x \circledast u$ is a vector of size $W - w + 1$, with

$$\begin{aligned}(x \circledast u)_i &= \sum_{j=1}^{w} x_{i-1+j}\, u_j \\ &= (x_i, \dots, x_{i+w-1}) \cdot u\end{aligned}$$

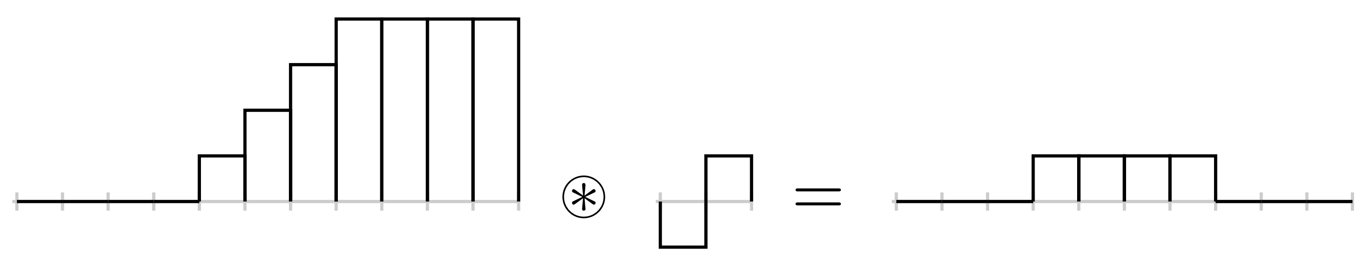
for instance

$$(1, 2, 3, 4) \circledast (3, 2) = (3 + 4, 6 + 6, 9 + 8) = (7, 12, 17).$$

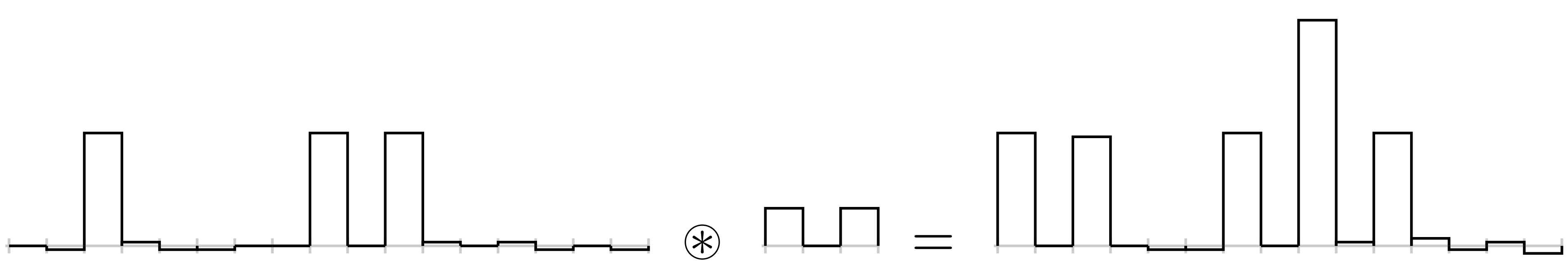
Convolution

Convolution can implement in particular differential operators, *e.g.*

$$(0,0,0,0,1,2,3,4,4,4,4) \circledast (-1,1) = (0,0,0,1,1,1,1,0,0,0).$$



or crude "template matcher", *e.g.*



Convolution in 2D

It generalizes naturally to a multi-dimensional input, although specification can become complicated.

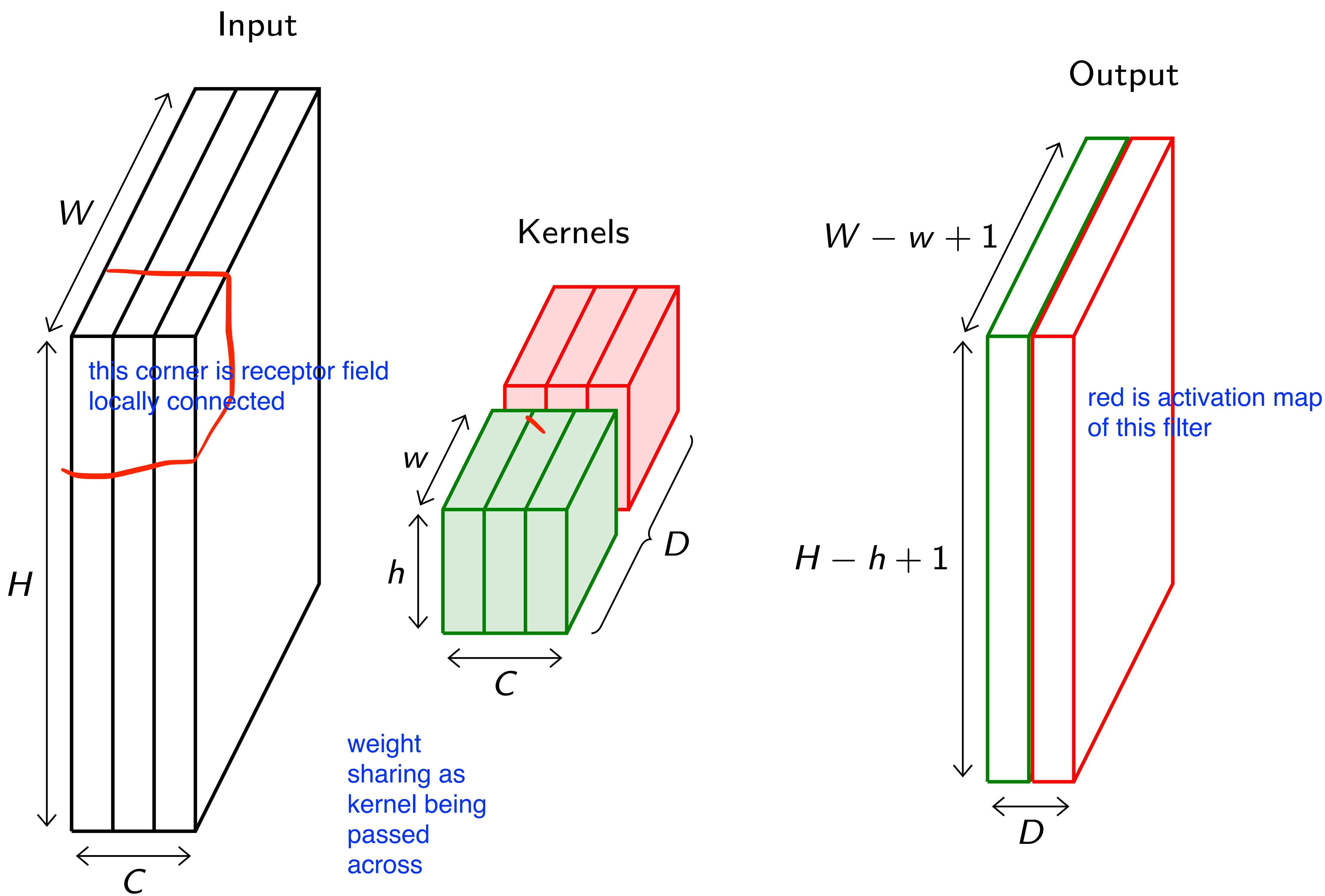
Its most usual form for "convolutional networks" processes a 3d tensor as input (*i.e.* a multi-channel 2d signal) to output a 2d tensor. The kernel is not swiped across channels, just across rows and columns.

In this case, if the input tensor is of size $C \times H \times W$, and the kernel is $C \times h \times w$, the output is $(H - h + 1) \times (W - w + 1)$.

We say "2d signal" even though it has C channels, since it is a feature vector indexed by a 2d location without structure on the feature indexes.

In a standard convolution layer, D such convolutions are combined to generate a $D \times (H - h + 1) \times (W - w + 1)$ output.

Convolution in 2D



Convolution in 2D

Note that a convolution **preserves the signal support structure**.

A 1d signal is converted into a 1d signal, a 2d signal into a 2d, and neighboring parts of the input signal influence neighboring parts of the output signal.

A 3d convolution can be used if the channel index has some metric meaning, such as time for a series of grayscale video frames. Otherwise swiping across channels makes no sense.

We usually refer to one of the channels generated by a convolution layer as an **activation map.**

The sub-area of an input map that influences a component of the output as the **receptive field** of the latter.

In the context of convolutional networks, a standard linear layer is called a **fully connected layer** since every input influences every output.

Convolution in 2D

```
torch.nn.functional.conv2d(input, weight, bias=None,
                           stride=1, padding=0, dilation=1, groups=1)
```

Implements a 2d convolution, where `weight` contains the kernels, and is $D \times C \times h \times w$, `bias` is of dimension D, `input` is of dimension

$$N \times C \times H \times W$$

N number of instances, C number of channels.

and the result is of dimension

$$N \times D \times (H - h + 1) \times (W - w + 1).$$

```
>>> weight = torch.empty(5, 4, 2, 3).normal_()
>>> bias = torch.empty(5).normal_()
>>> input = torch.empty(117, 4, 10, 3).normal_()
>>> output = torch.nn.functional.conv2d(input, weight, bias)
>>> output.size()
torch.Size([117, 5, 9, 1])
```

Similar functions implement 1d and 3d convolutions.

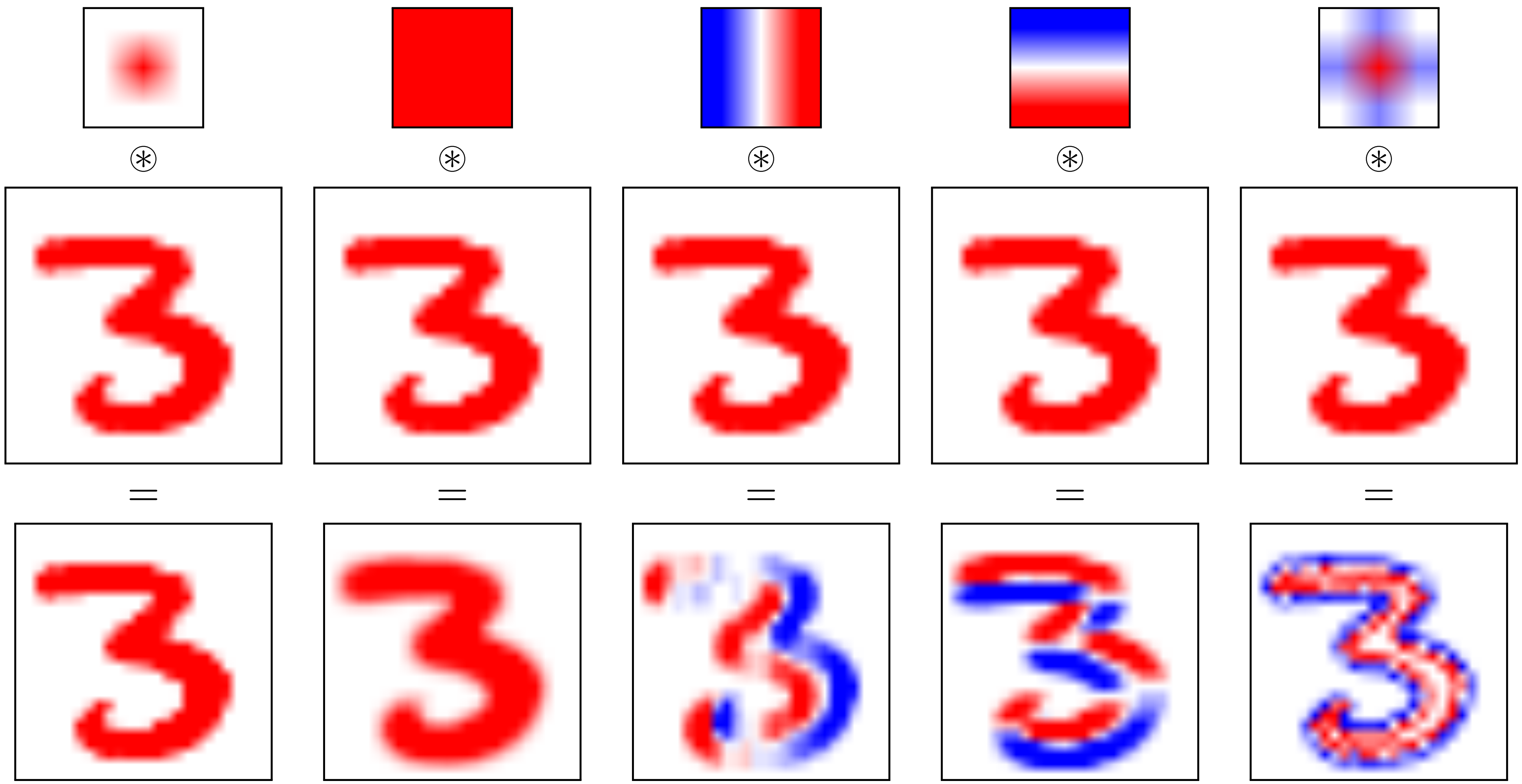
Convolution in 2D

```
x = mnist_train.data[12].float().view(1, 1, 28, 28)

weight = torch.empty(5, 1, 3, 3)
```

5 filters, 1 channel

```
weight[0, 0] = torch.tensor([ [  0.,   0.,   0. ],
                              [  0.,   1.,   0. ],
                              [  0.,   0.,   0. ] ])

weight[1, 0] = torch.tensor([ [  1.,   1.,   1. ],
                              [  1.,   1.,   1. ],
                              [  1.,   1.,   1. ] ])

weight[2, 0] = torch.tensor([ [ -1.,   0.,   1. ],
                              [ -1.,   0.,   1. ],
                              [ -1.,   0.,   1. ] ])

weight[3, 0] = torch.tensor([ [ -1.,  -1.,  -1. ],
                              [  0.,   0.,   0. ],
                              [  1.,   1.,   1. ] ])

weight[4, 0] = torch.tensor([ [  0.,  -1.,   0. ],
                              [ -1.,   4.,  -1. ],
                              [  0.,  -1.,   0. ] ])

y = torch.nn.functional.conv2d(x, weight)
```

Convolution in 2D



the same

blurring effect

Convolution in 2D

```
class torch.nn.Conv2d(in_channels, out_channels,
                      kernel_size, stride=1, padding=0, dilation=1,
                      groups=1, bias=True)
```

Wraps the convolution into a `Module`, with the kernels and biases as `Parameter` properly randomized at creation.

The kernel size is either a pair (h, w) or a single value k interpreted as (k, k).

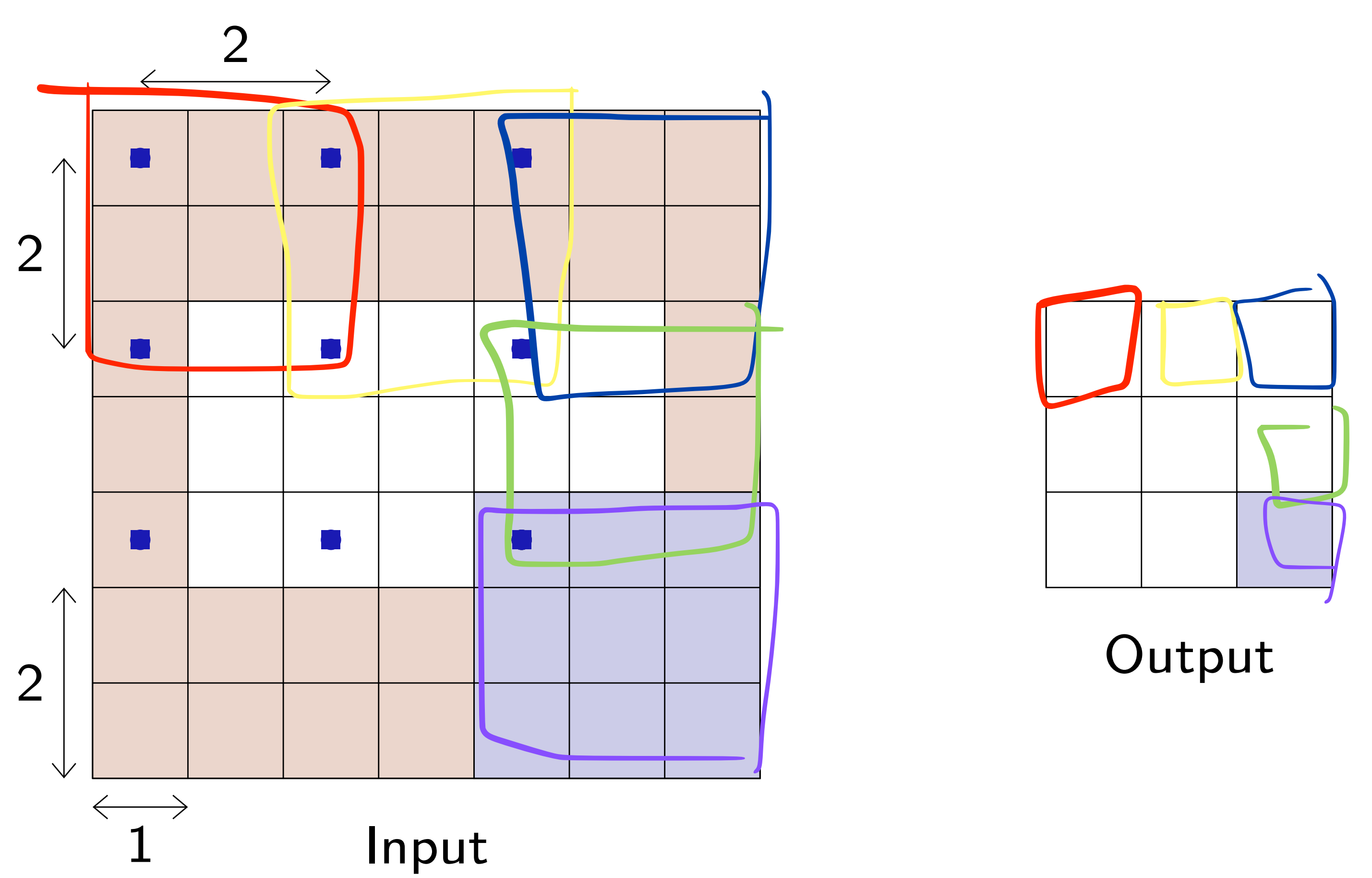
```
>>> f = nn.Conv2d(in_channels = 4, out_channels = 5, kernel_size = (2, 3))
>>> for n, p in f.named_parameters(): print(n, p.size())
...
weight torch.Size([5, 4, 2, 3])
bias torch.Size([5])
>>> x = torch.empty(117, 4, 10, 3).normal_()
>>> y = f(x)
>>> y.size()
torch.Size([117, 5, 9, 1])
```

Convolution: padding, stride

Convolutions have two additional standard parameters:

- The **padding** specifies the size of a zeroed frame added around the input,
- the **stride** specifies a step size when moving the kernel across the signal.

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$, the output is $1 \times 3 \times 3$. stride length a hypaparam to set yourself



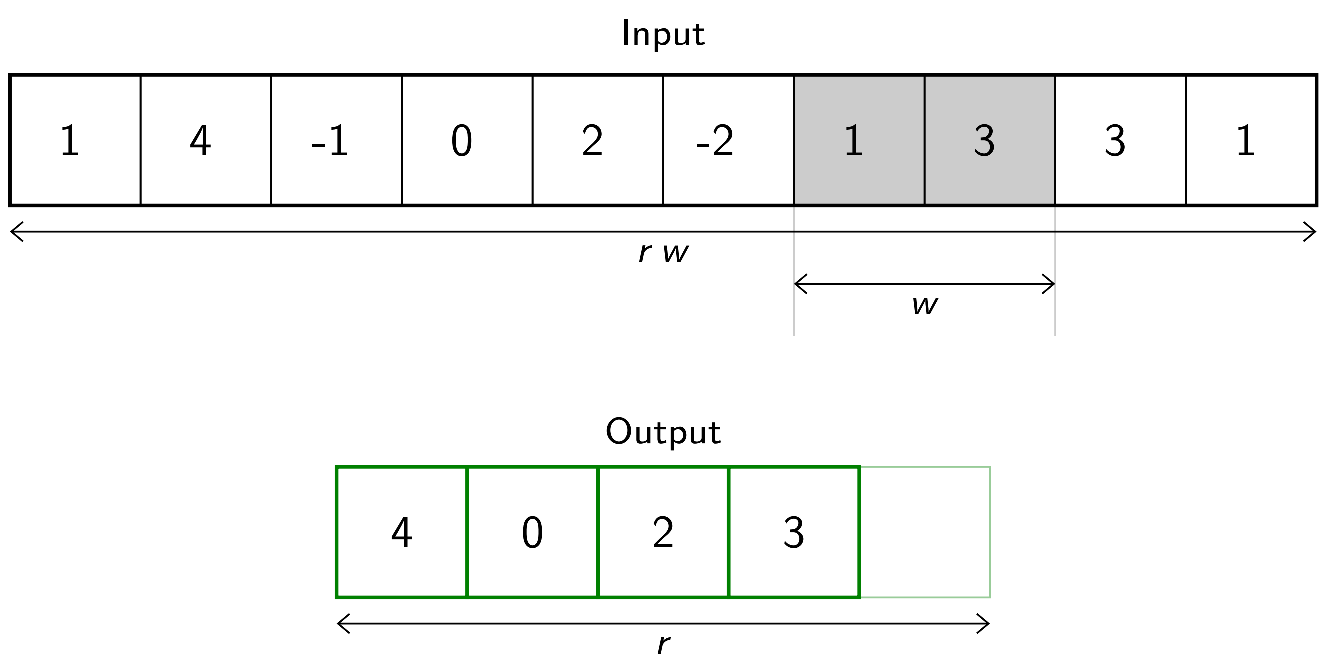
Pooling

The historical approach to compute a low-dimension signal (*e.g.* a few scores) from a high-dimension one (*e.g.* an image) was to use **pooling** operations.

Such an operation aims at grouping several activations into a single "more meaningful" one.

The most standard type of pooling is the **max-pooling**, which computes max values over non-overlapping blocks.

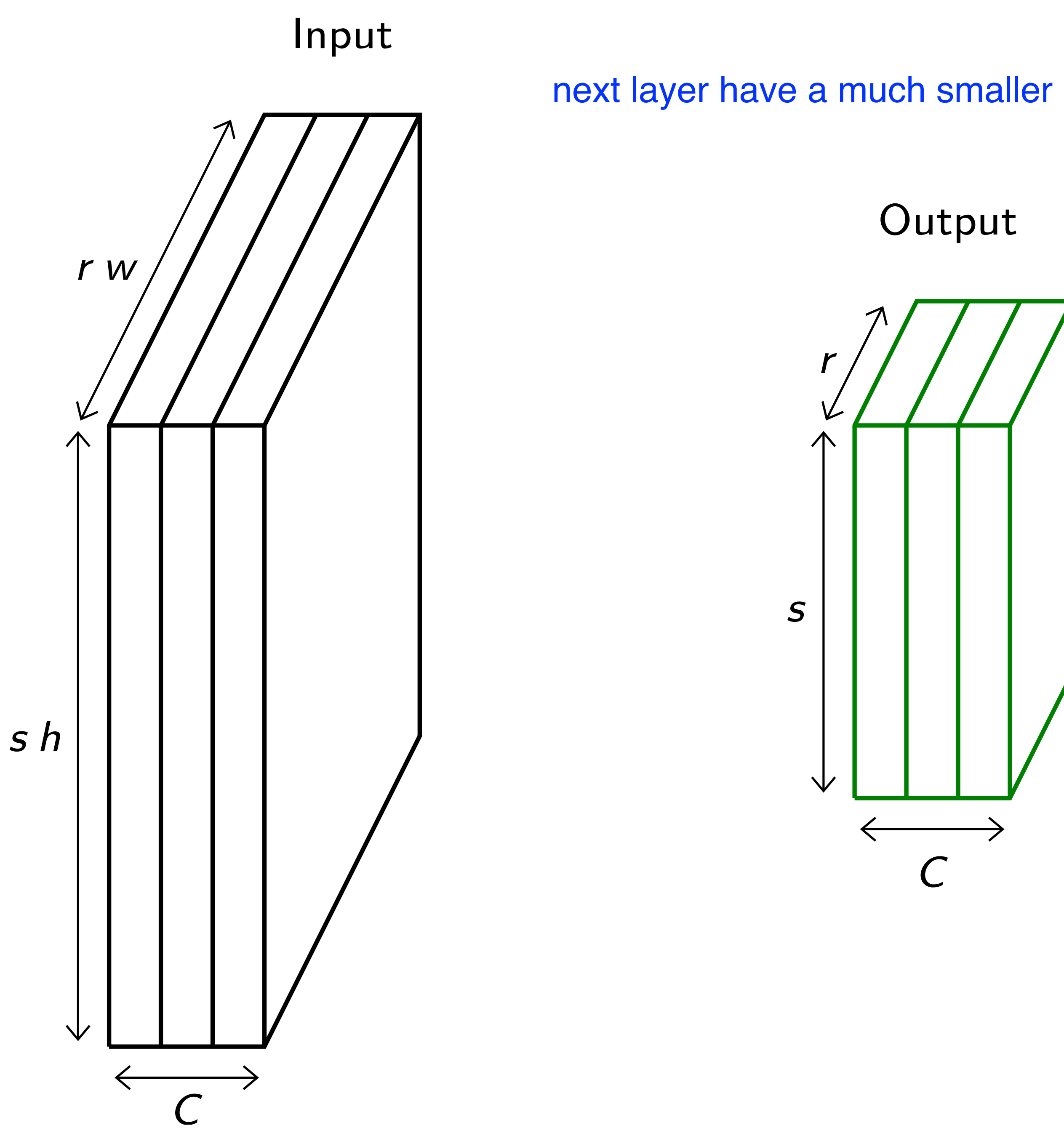
For instance in 1d with a kernel of size 2:



The **average pooling** computes average values per block instead of max values.

Pooling

next layer have a much smaller input, activation map etc.



Pooling

```
torch.nn.functional.max_pool2d(input, kernel_size,
                               stride=None, padding=0, dilation=1,
                               ceil_mode=False, return_indices=False)
```

takes as input a $N \times C \times H \times W$ tensor, and a kernel size (h, w) or k interpreted as (k, k), applies the max-pooling on each channel of each sample separately, and produce if the padding is 0 a $N \times C \times \lfloor H/h \rfloor \times \lfloor W/w \rfloor$ output.

```
>>> x = torch.empty(2, 2, 6).random_(3)
>>> x
tensor([[[ 1.,  2.,  2.,  1.,  2.,  1.],
         [ 2.,  0.,  0.,  0.,  1.,  0.]],

        [[ 2.,  0.,  2.,  1.,  1.,  1.],
         [ 0.,  0.,  0.,  1.,  2.,  1.]]])
>>> F.max_pool2d(x, (1, 2))
tensor([[[ 2.,  2.,  2.],
         [ 2.,  0.,  1.]],

        [[ 2.,  2.,  1.],
         [ 0.,  1.,  2.]]])
```

Similar functions implements 1d and 3d max-pooling, and average pooling.