# Deep learning

# 4.4. Convolutions

François Fleuret

https://fleuret.org/dlc/

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 $256 \times 256$ 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 transformation meaningful at a certain location can / should be used everywhere.**

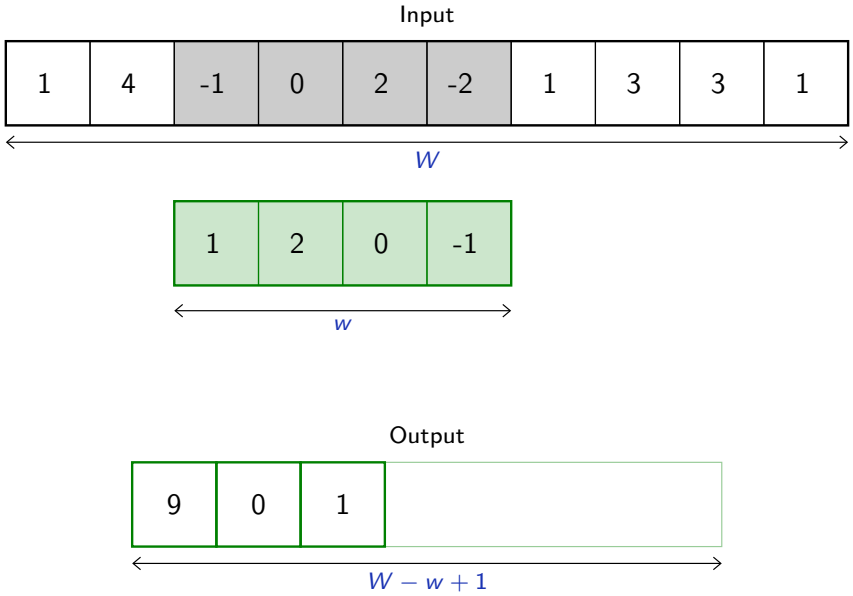
A convolution layer embodies this idea. **It applies the same linear transformation locally, everywhere**



---

**Notes**

Convolution preserve the structure of the signal: if the input signal is a 2d tensor, then the output of a convolution layer is will be 2d tensor and there will be a clear relation between the locations of the values in the input/output tensors.






**Notes**

This is an illustration in 1D of how convolutions works.
Convolving an input signal with a weight vector also called "kernel" (filled in green here) consists of repeating a dot product between the weight vector and a vector of same length (in gray) extracted from the input vector, for every position of the weight vector while it is swiped across the entire input signal.
As we can see on this simple example, the structure of the signal is preserved: convolving a 1D signal with a 1D kernel produces a 1D signal.
The resulting tensor is shorter than the input to account for the size of the weight vector. We will see later that the input signal can be "padded" with zeros so that the output signal is of same size as the input.

Formally, in 1d, given

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

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

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

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

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

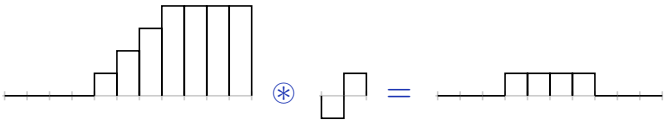
for instance

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

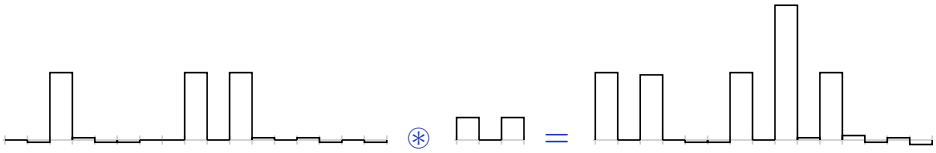
This differs from the usual convolution since the kernel and the signal are both visited in increasing index order.

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.





**Notes**

The first example shows that a convolution can computes the discrete first order derivative of the input signal. The second that it can "detect" a structure in a crude sense: Here the kernel has the shape of two peaks and the response is maximal when the dot product is done with a similar structure in the input.
By having a way of computing derivatives and matching pattern, we can envision the following applications:

- detecting corners, edges in images,
- detecting amplitude modification in sounds.

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

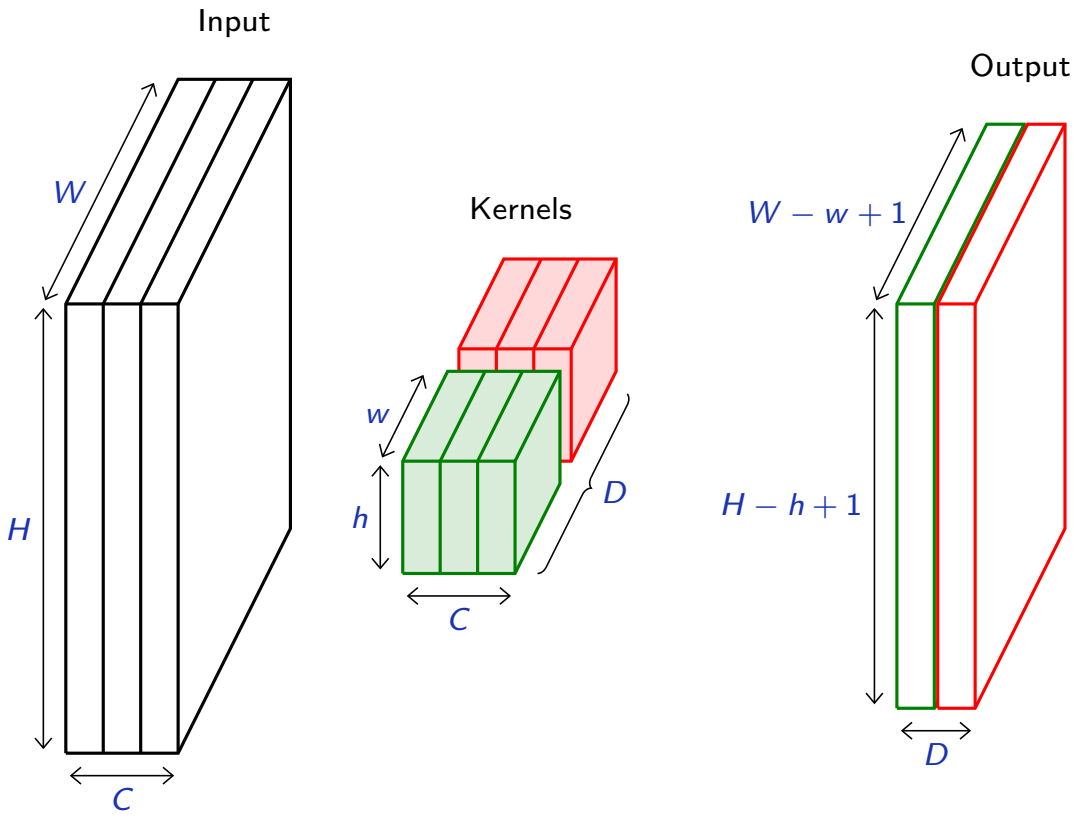
We say "2d signal" even though it has $C$ channels, since it is a feature vector indexed by a 2d location without structure on the feature indexes.

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



**Notes**

An RGB image is a multi-channel 2D signal. It can be viewed as 3 2D signals, one for each color red, green, and blue, and is stored by convention in PyTorch in a tensor of shape `torch.Size([3, H, W])`.



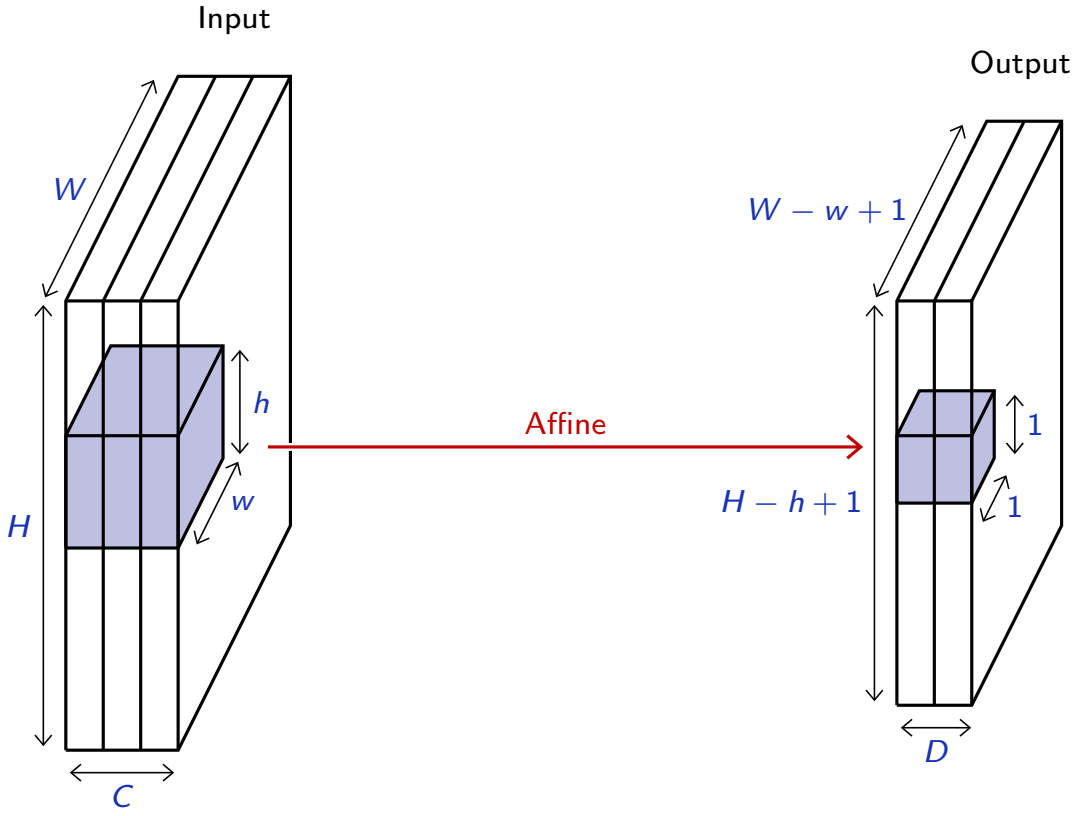



**Notes**

This is an illustration of how convolving a 2d tensor works. Note that in practice, as explained in 4.3. "PyTorch modules and batch processing", this is always done by batch.
In this example:

- the input is a tensor of size $C \times H \times W$, here with $C = 3$,
- the convolution layer consists of $D = 2$ filters of size $C \times h \times w$,
- the output of the layer is a tensor of size $D \times (H - h + 1) \times (W - w + 1)$.

Note that the kernel is not swiped across channels: the kernel has $C$ channels as the input signal: it is not the same $1 \times h \times w$ kernel applied on each channel.






**Notes**

A simpler way of envisioning the convolution is that the output column $D \times 1 \times 1$ is an affine function of the input block of size $C \times h \times w$ in the original input tensor. **This affine mapping is the same everywhere in the tensor.**

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.

And a convolution is **equivariant** to a translation of the input signal, since its output is translated similarly.

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**, or a **dense layer**, since every input influences every output.

The autograd-compliant function

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

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

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

and the result is of dimension

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

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

Similar functions implement 1d and 3d convolutions.



---

**Notes**

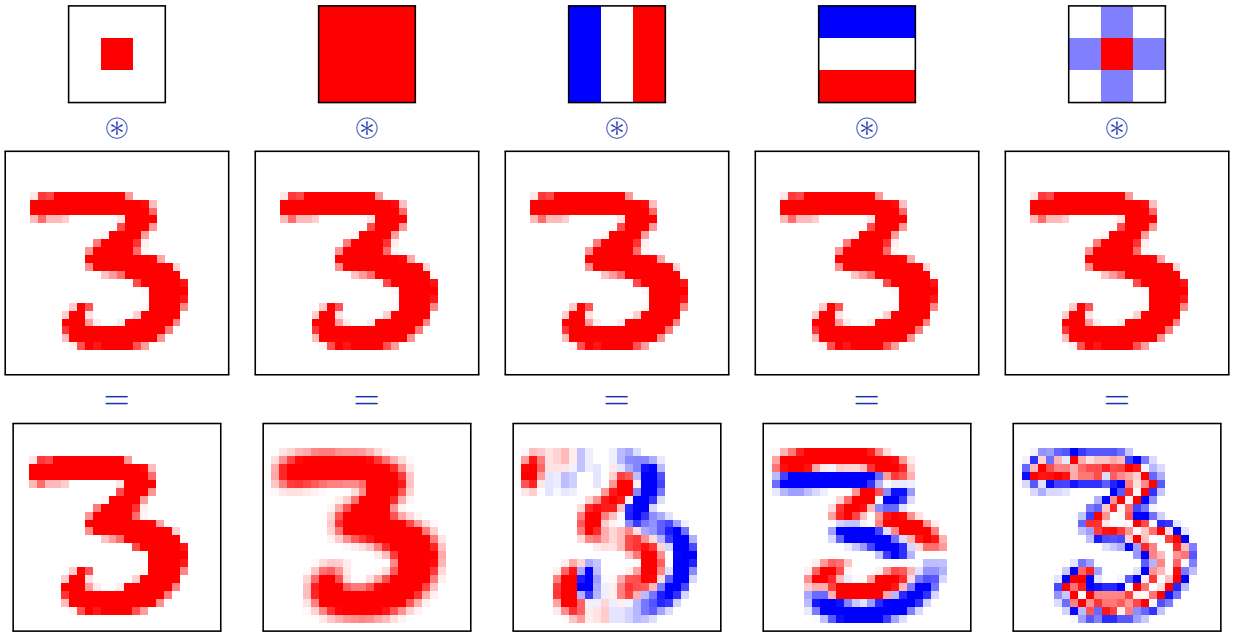
- $N$ is the number of samples to process by the layer,
- $C \times H \times W$ is the size of the input tensor,
- $D$ is the number of filters in the layer,
- $h \times w$ is the size of the filters,

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

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

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 = F.conv2d(x, weight)
```



**Notes**

We define by hand five $3 \times 3$ kernels to illustrates the 2d convolutions on a MNIST image.
The first kernel simply copies the central value, the second sums over the $3 \times 3$ area, the third computes the difference between the sum of the right pixels and the sum of the left ones, the fourth does the same with top/bottom, and the fifth computes the difference between four times the central pixel and the sum of its neighbors.





**Notes**

The results images are normalized and the signed values are converted to color as follows:

- positive values are in red,
- negative values are in blue,
- white is zero.

The output tensor are two coefficients smaller than the input since the filters are $3 \times 3$.
We observe that the first filter keep the image unchanged, the second blurs it, the third and fourth compute respectively vertical and horizontal edge responses, and the fifth "sharpens" the image and keeps only high frequencies.

```
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 $(k, k)$.

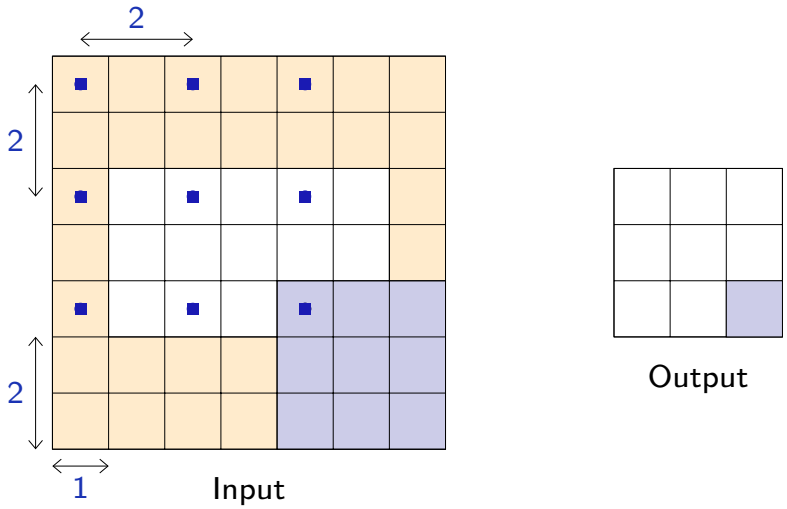
```
>>> f = nn.Conv2d(in_channels = 4, out_channels = 5, kernel_size = (2, 3))
>>> for n, p in f.named_parameters(): print(n, p.size())
...
weight torch.Size([5, 4, 2, 3])
bias torch.Size([5])
>>> x = torch.randn(117, 4, 10, 3)
>>> y = f(x)
>>> y.size()
torch.Size([117, 5, 9, 1])
```

# Padding, stride, and dilation

Convolutions have three additional 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,
- the **dilation** modulates the expansion of the filter without adding weights.

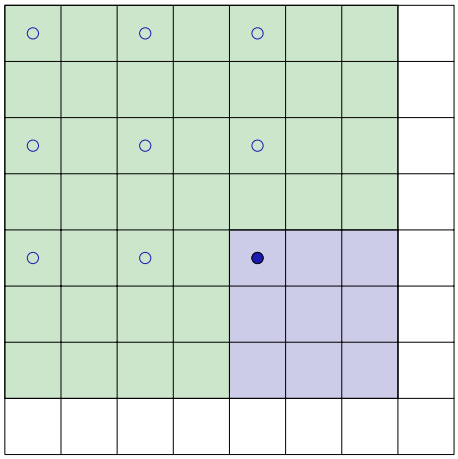
Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$, the output is $1 \times 3 \times 3$.






**Notes**

The padding states how many zeroes are added on the top, bottom, left, and right part of the signal. Here the padding is $(2, 1)$ hence two rows are added at the top/bottom and one column on the left/right. They are depicted in brown.
The stride states how coarsely the filter should be swiped across the signal. Here it is $(2, 2)$, and the locations where the filter is applied, indicated with blue dots are only even coordinates.
Finally the kernel itself is $C \times 3 \times 3$, and we do not depict the depth $C$, and the $3 \times 3$ shape of the filter is shown in blue.



A convolution with a stride greater than 1 may not cover the input map entirely, hence may ignore some of the input values.



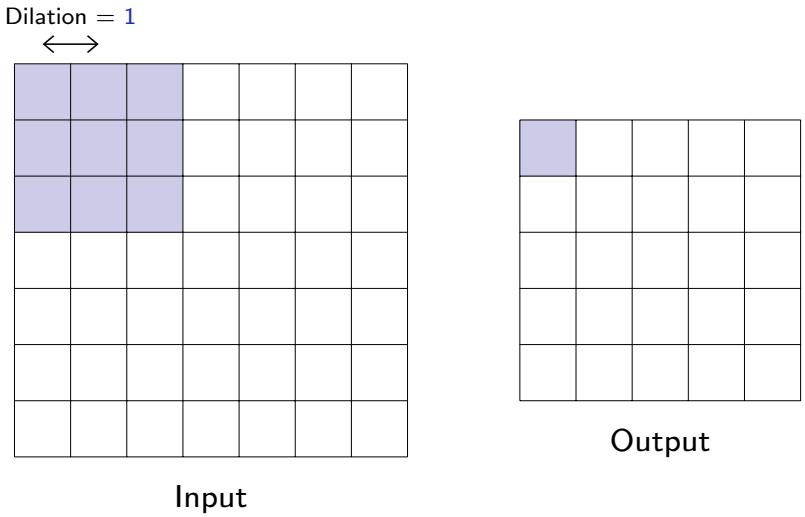
**Notes**

On the right part of the tensor, the kernel cannot be moved more, hence the last row and the last column will not be taken into account. The green area shows which part of the input signal the output actually depends on.

The dilation modulates the expansion of the filter support by adding rows and columns of zeros between coefficients (Yu and Koltun, 2015).

It is 1 for standard convolutions, but can be greater, in which case the resulting operation can be envisioned as a convolution with a regularly sparsified filter.

This notion comes from signal processing, where it is referred to as *algorithme à trous*, hence the term sometime used of "convolution à trous".

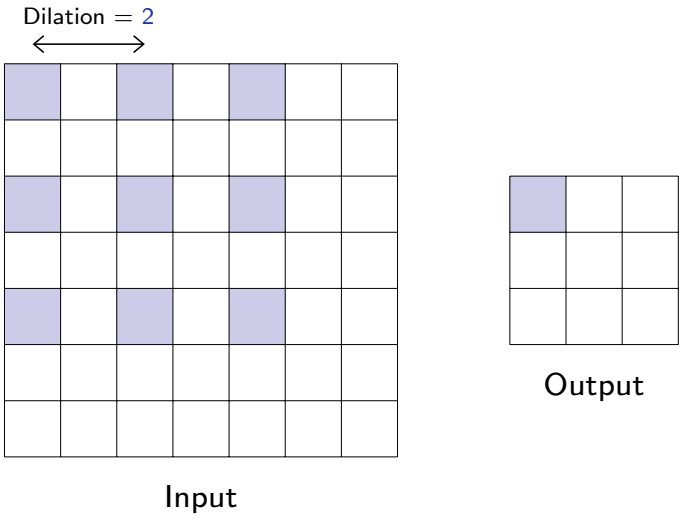





**Notes**

A standard convolution:

- the blue pixels on the left are the locations in the input signal where the filter has coefficients,
- the blue pixel on the right is the output.






**Notes**

A dilated convolution:

- the blue pixels on the left are the locations in the input signal where the filter has coefficients,
- the blue pixel on the right is the output.

In this case, the output is smaller than with a standard convolution because the filter cannot be moved as much.
The dilation is the number of zeros which are inserted between the rows and the column of the filter to "dilate" it.

A 1d convolution with a kernel of size $k$ and dilation $d$ can be interpreted as a convolution with a filter of size $1 + (k - 1)d$ with only $k$ non-zero coefficients.

For example with $k = 3$ and $d = 4$, the difference between the input map size and the output map size is $1 + (3 - 1)4 - 1 = 8$.

```
>>> x = torch.randn(1, 1, 20, 30)
>>> l = nn.Conv2d(1, 1, kernel_size = 3, dilation = 4)
>>> l(x).size()
torch.Size([1, 1, 12, 22])
```

Having a dilation greater than one increases the units' receptive field size without increasing the number of parameters.

**Convolutions with stride or dilation strictly greater than one reduce the activation map size, for instance to make a final classification decision.**

## References

F. Yu and V. Koltun. **Multi-scale context aggregation by dilated convolutions**. CoRR, abs/1511.07122v3, 2015.