Deep learning

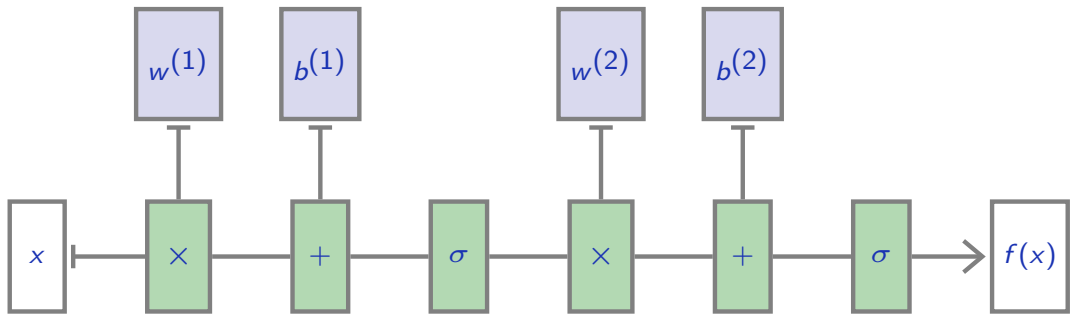
4.1. DAG networks

François Fleuret

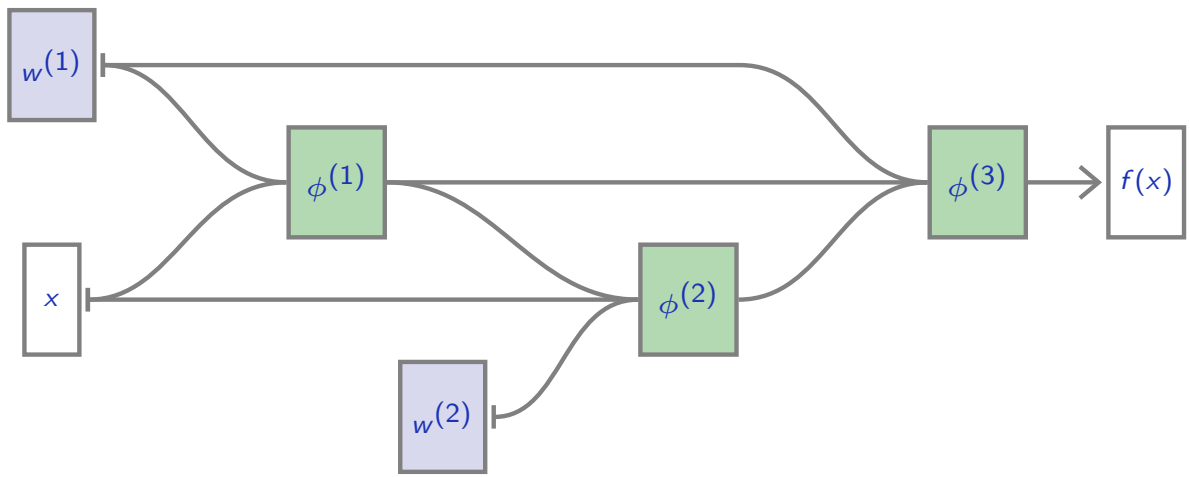
https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

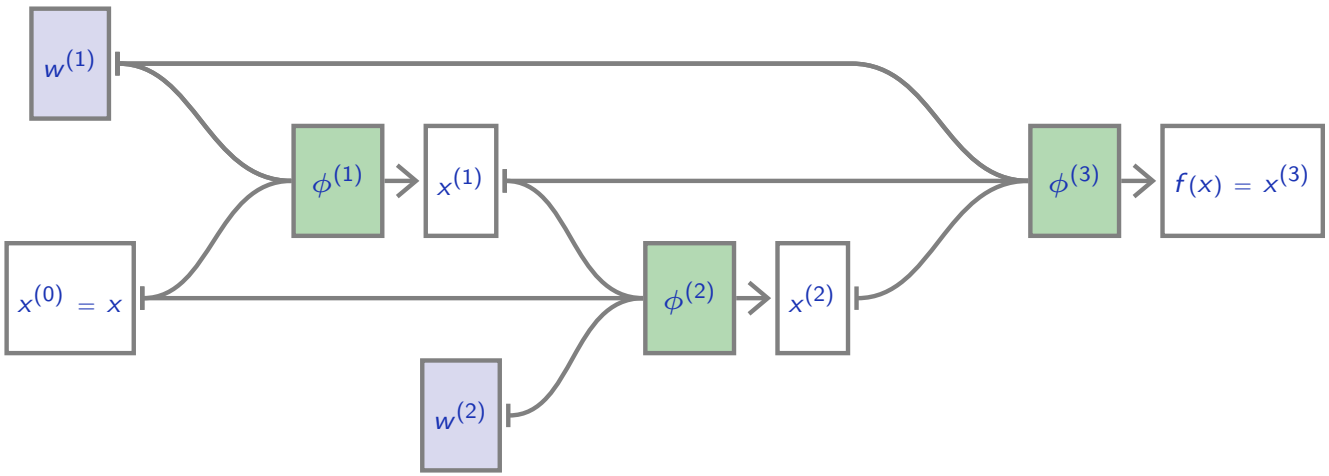
We can generalize an MLP



to an arbitrary "Directed Acyclic Graph" (DAG) of operators



Forward pass



$$
\begin{aligned}
x^{(0)} &= x \\
x^{(1)} &= \phi^{(1)}(x^{(0)}; w^{(1)}) \\
x^{(2)} &= \phi^{(2)}(x^{(0)}, x^{(1)}; w^{(2)}) \\
f(x) = x^{(3)} &= \phi^{(3)}(x^{(1)}, x^{(2)}; w^{(1)})
\end{aligned}
$$

If $(a_1, \ldots, a_Q) = \phi(b_1, \ldots, b_R)$, we use the notation

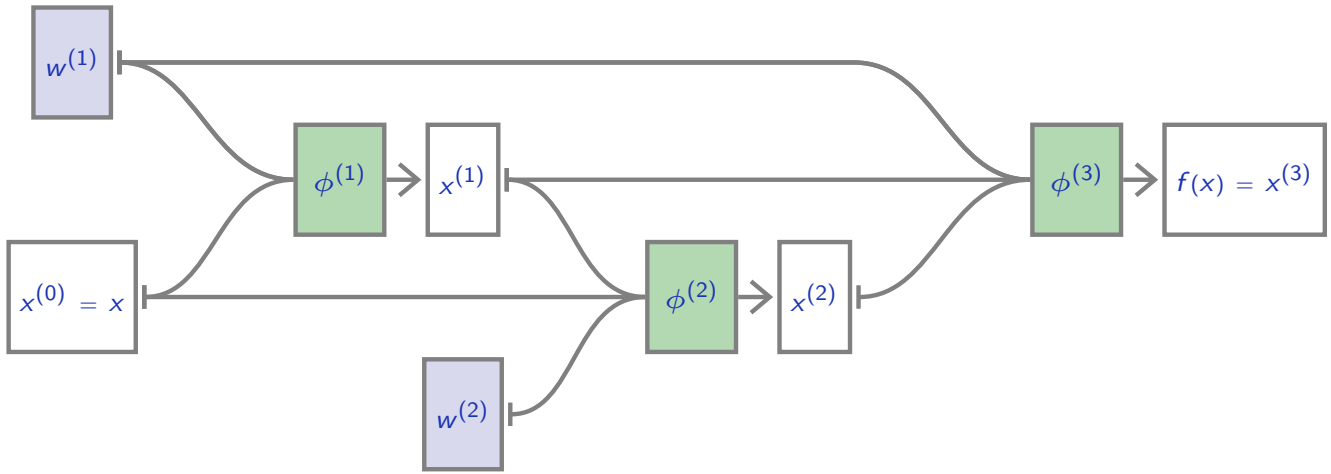
$$\left[\frac{\partial a}{\partial b}\right] = J_\phi^\top = \begin{pmatrix} \frac{\partial a_1}{\partial b_1} & \cdots & \frac{\partial a_Q}{\partial b_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_1}{\partial b_R} & \cdots & \frac{\partial a_Q}{\partial b_R} \end{pmatrix}.$$

It does not specify at which point this is computed, but it will always be for the forward-pass activations.

Also, if $(a_1, \ldots, a_Q) = \phi(b_1, \ldots, b_R, c_1, \ldots, c_S)$, we use

$$\left[\frac{\partial a}{\partial c}\right] = J_{\phi|c}^\top = \begin{pmatrix} \frac{\partial a_1}{\partial c_1} & \cdots & \frac{\partial a_Q}{\partial c_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_1}{\partial c_S} & \cdots & \frac{\partial a_Q}{\partial c_S} \end{pmatrix}.$$

Backward pass, derivatives w.r.t activations

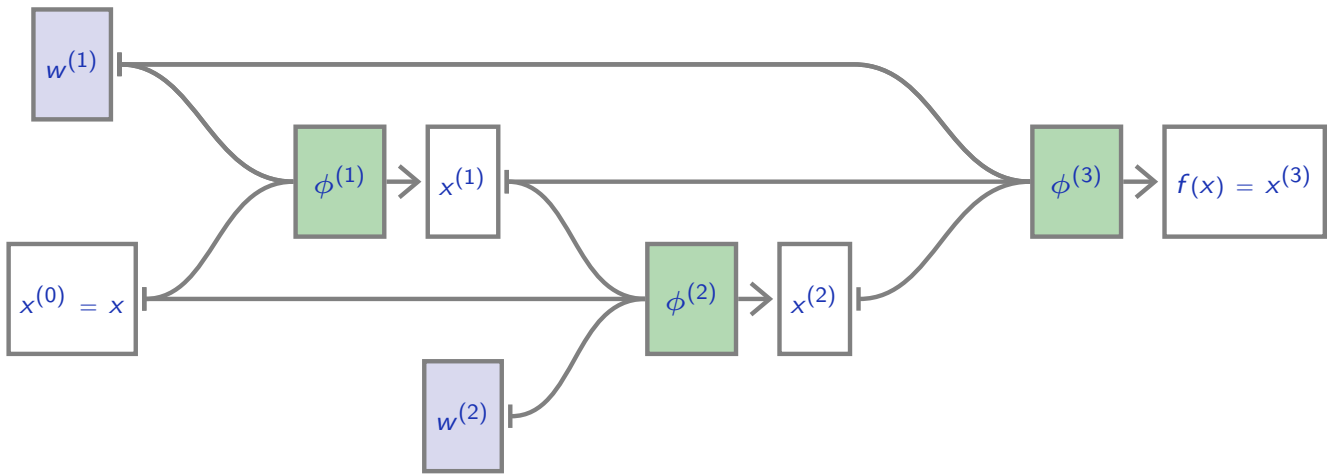


$$\left[\frac{\partial \ell}{\partial x^{(2)}}\right] = \left[\frac{\partial x^{(3)}}{\partial x^{(2)}}\right] \left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(3)}|x^{(2)}}^{\top} \left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

$$\left[\frac{\partial \ell}{\partial x^{(1)}}\right] = \left[\frac{\partial x^{(2)}}{\partial x^{(1)}}\right] \left[\frac{\partial \ell}{\partial x^{(2)}}\right] + \left[\frac{\partial x^{(3)}}{\partial x^{(1)}}\right] \left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(2)}|x^{(1)}}^{\top} \left[\frac{\partial \ell}{\partial x^{(2)}}\right] + J_{\phi^{(3)}|x^{(1)}}^{\top} \left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

$$\left[\frac{\partial \ell}{\partial x^{(0)}}\right] = \left[\frac{\partial x^{(1)}}{\partial x^{(0)}}\right] \left[\frac{\partial \ell}{\partial x^{(1)}}\right] + \left[\frac{\partial x^{(2)}}{\partial x^{(0)}}\right] \left[\frac{\partial \ell}{\partial x^{(2)}}\right] = J_{\phi^{(1)}|x^{(0)}}^{\top} \left[\frac{\partial \ell}{\partial x^{(1)}}\right] + J_{\phi^{(2)}|x^{(0)}}^{\top} \left[\frac{\partial \ell}{\partial x^{(2)}}\right]$$

Backward pass, derivatives w.r.t parameters



$$\left[\frac{\partial \ell}{\partial w^{(1)}}\right] = \left[\frac{\partial x^{(1)}}{\partial w^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(1)}}\right] + \left[\frac{\partial x^{(3)}}{\partial w^{(1)}}\right]\left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(1)}|w^{(1)}}^{\top}\left[\frac{\partial \ell}{\partial x^{(1)}}\right] + J_{\phi^{(3)}|w^{(1)}}^{\top}\left[\frac{\partial \ell}{\partial x^{(3)}}\right]$$

$$\left[\frac{\partial \ell}{\partial w^{(2)}}\right] = \left[\frac{\partial x^{(2)}}{\partial w^{(2)}}\right]\left[\frac{\partial \ell}{\partial x^{(2)}}\right] = J_{\phi^{(2)}|w^{(2)}}^{\top}\left[\frac{\partial \ell}{\partial x^{(2)}}\right]$$

So if we have a library of "tensor operators", and implementations of

$$
\begin{aligned}
& (x_1, \ldots, x_d, w) \mapsto \phi(x_1, \ldots, x_d; w) \\
\forall c, \quad & (x_1, \ldots, x_d, w) \mapsto J_{\phi|x_c}(x_1, \ldots, x_d; w) \\
& (x_1, \ldots, x_d, w) \mapsto J_{\phi|w}(x_1, \ldots, x_d; w),
\end{aligned}
$$

we can build any directed acyclic graph with these operators at the nodes, evaluate the resulting mapping, and compute its gradient with back-prop.

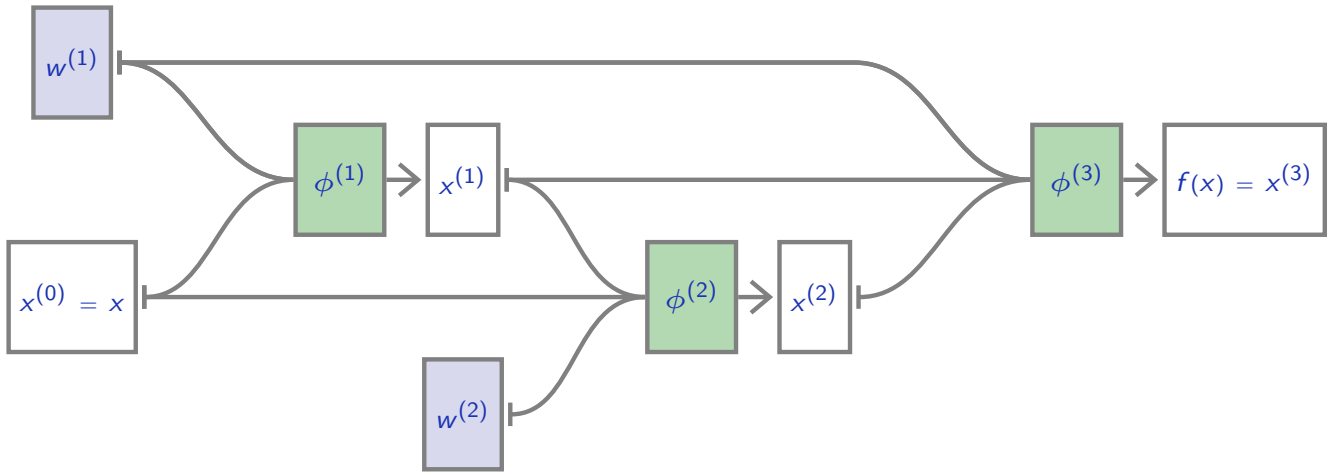
Writing from scratch a large neural network is complex and error-prone.

Multiple frameworks provide libraries of tensor operators and mechanisms to combine them into DAGs and automatically differentiate them.

	Language(s)	**License**	**Main backer**
PyTorch	**Python**, C++	BSD	Facebook
TensorFlow	Python, C++	Apache	Google
JAX	Python	Apache	Google
MXNet	Python, C++, R, Scala	Apache	Amazon
CNTK	Python, C++	MIT	Microsoft
Torch 7	Lua	BSD	Facebook
Theano	Python	BSD	U. of Montreal
Caffe	C++	BSD 2 clauses	U. of CA, Berkeley

One approach is to define the nodes and edges of such a DAG statically (TensorFlow, Torch 7, Caffe, Theano, etc.)

In TensorFlow, to run a forward/backward pass on



$$\phi^{(1)}\left(x^{(0)}; w^{(1)}\right) = w^{(1)}x^{(0)}$$

$$\phi^{(2)}\left(x^{(0)}, x^{(1)}; w^{(2)}\right) = x^{(0)} + w^{(2)}x^{(1)}$$

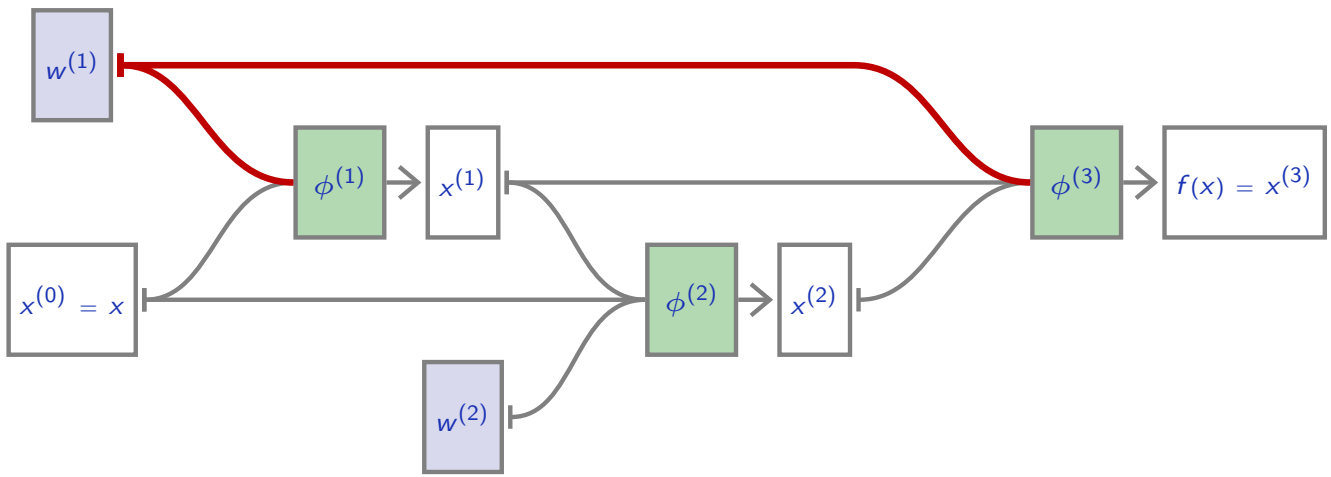
$$\phi^{(3)}\left(x^{(1)}, x^{(2)}; w^{(1)}\right) = w^{(1)}\left(x^{(1)} + x^{(2)}\right)$$

```
w1 = tf.Variable(tf.random_normal([5, 5]))
w2 = tf.Variable(tf.random_normal([5, 5]))
x = tf.Variable(tf.random_normal([5, 1]))
x0 = x
x1 = tf.matmul(w1, x0)
x2 = x0 + tf.matmul(w2, x1)
x3 = tf.matmul(w1, x1 + x2)
q = tf.norm(x3)

gw1, gw2 = tf.gradients(q, [w1, w2])

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    _gw1, _gw2 = sess.run([gw1, gw2])
```

Weight sharing

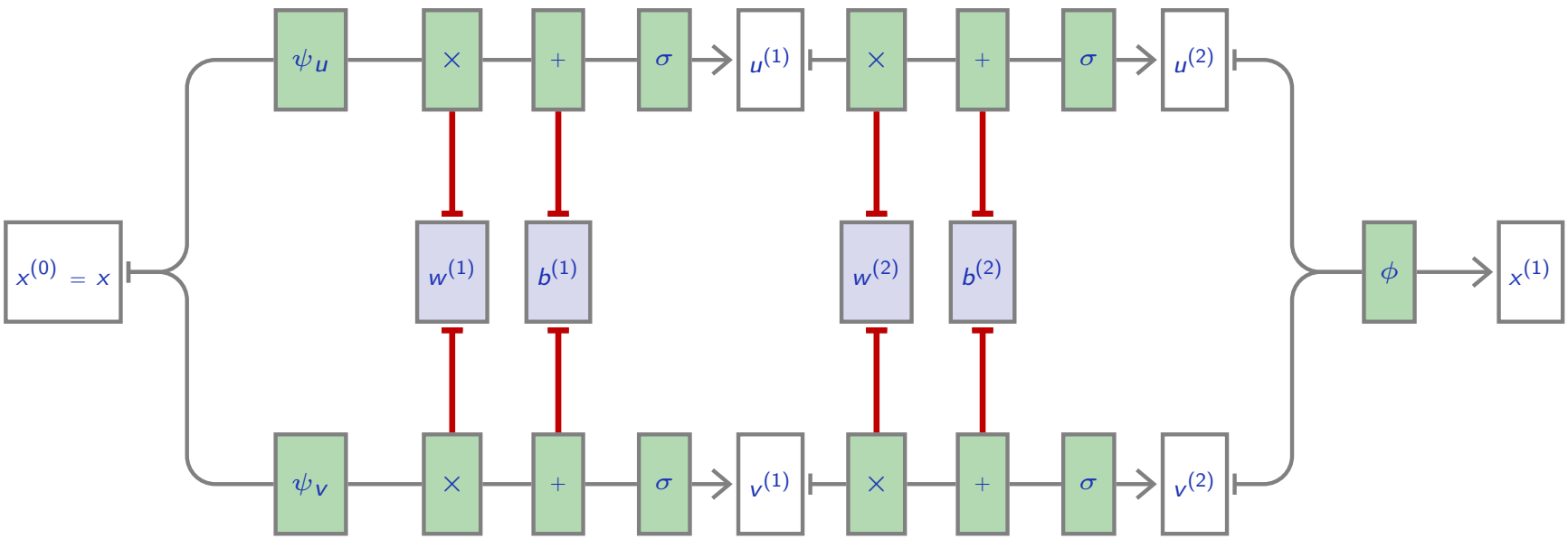
In our generalized DAG formulation, we have allowed the same parameters to modulate different parts of the processing.

For instance $w^{(1)}$ in our example parametrizes both $\phi^{(1)}$ and $\phi^{(3)}$.

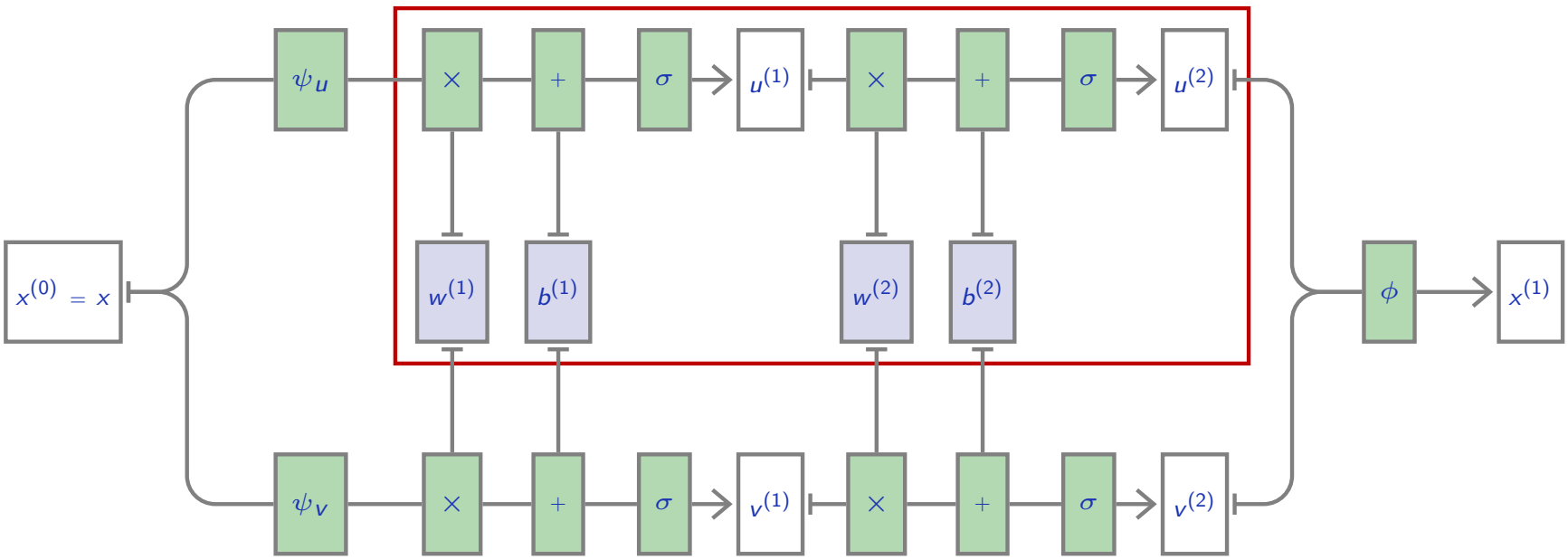


This is called **weight sharing**.

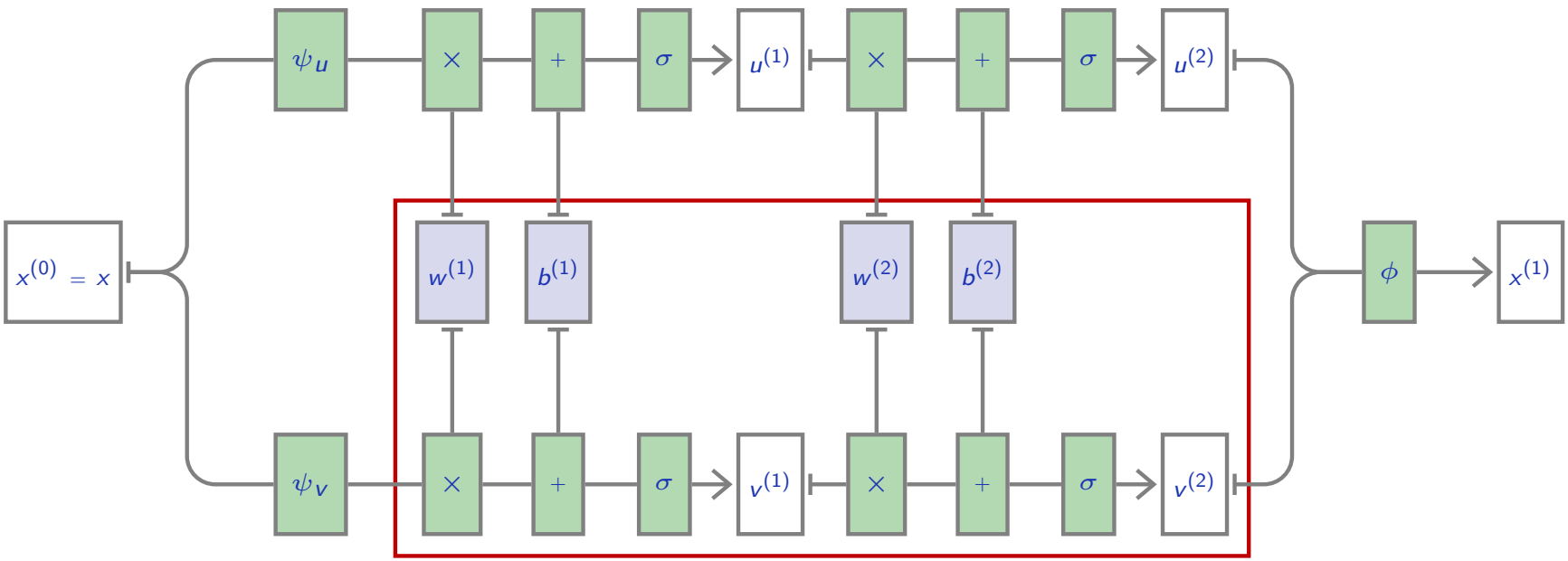
Weight sharing allows in particular to build **Siamese networks** where a full sub-network is replicated several times.



Weight sharing allows in particular to build **Siamese networks** where a full sub-network is replicated several times.



Weight sharing allows in particular to build **Siamese networks** where a full sub-network is replicated several times.



Deep learning

4.2. Autograd

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

Conceptually, the forward pass is a standard tensor computation, and the DAG of tensor operations is required only to compute derivatives.

When executing tensor operations, PyTorch can automatically construct on-the-fly the graph of operations to compute the gradient of any quantity with respect to any tensor involved.

This "autograd" mechanism (Paszke et al., 2017) has two main benefits:

- Simpler syntax: one just needs to write the forward pass as a standard sequence of Python operations,
- greater flexibility: since the graph is not static, the forward pass can be dynamically modulated.

A `Tensor` has a Boolean field `requires_grad`, set to `False` by default, which states if PyTorch should build the graph of operations so that gradients with respect to it can be computed.

The result of a tensorial operation has this flag to `True` if any of its operand has it to `True`.

```
>>> x = torch.tensor([ 1., 2. ])
>>> y = torch.tensor([ 4., 5. ])
>>> z = torch.tensor([ 7., 3. ])
>>> x.requires_grad
False
>>> (x + y).requires_grad
False
>>> z.requires_grad = True
>>> (x + z).requires_grad
True
```

Only floating point type tensors can have their gradient computed.

```
>>> x = torch.tensor([1., 10.])
>>> x.requires_grad = True
>>> x = torch.tensor([1, 10])
>>> x.requires_grad = True
Traceback (most recent call last):
/.../
RuntimeError: only Tensors of floating point dtype can require gradients
```

The method `requires_grad_(value = True)` set `requires_grad` to `value`, which is `True` by default.

`torch.autograd.grad(outputs, inputs)` computes and returns the gradient of `outputs` with respect to `inputs`.

```
>>> t = torch.tensor([1., 2., 4.]).requires_grad_()
>>> u = torch.tensor([10., 20.]).requires_grad_()
>>> a = t.pow(2).sum() + u.log().sum()
>>> torch.autograd.grad(a, (t, u))
(tensor([2., 4., 8.]), tensor([0.1000, 0.0500]))
```

`inputs` can be a single tensor, but the result is still a [one element] tuple.

If `outputs` is a tuple, the result is the sum of the gradients of its elements.

The function `Tensor.backward()` accumulates gradients in the `grad` fields of tensors which are not results of operations, the "leaves" in the autograd graph.

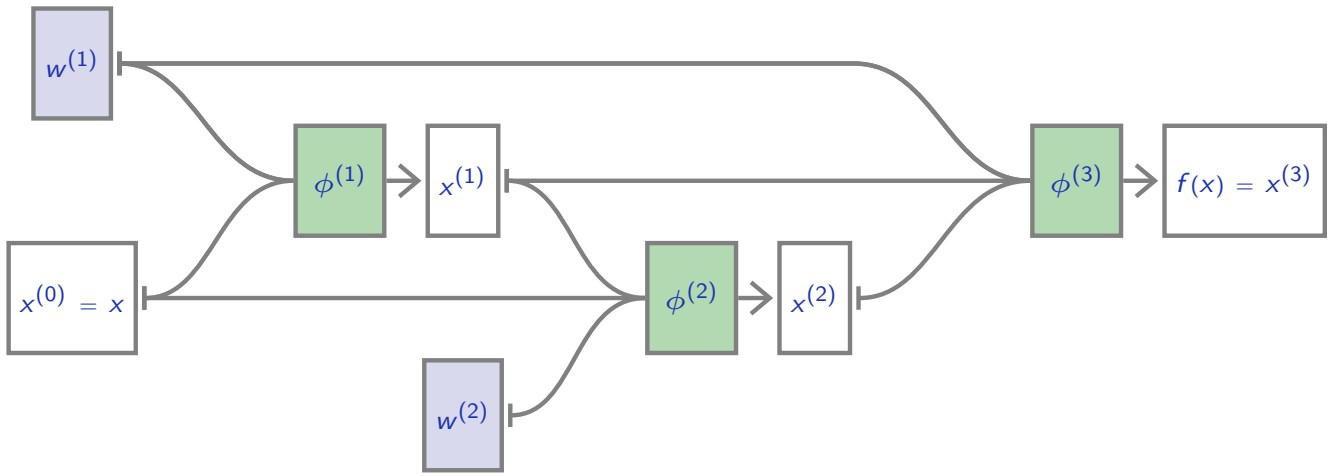
```
>>> x = torch.tensor([ -3., 2., 5. ]).requires_grad_()
>>> u = x.pow(3).sum()
>>> x.grad
>>> u.backward()
>>> x.grad
tensor([27., 12., 75.])
```

This function is an alternative to `torch.autograd.grad(...)` and standard for training models.

`Tensor.backward()` **accumulates** the gradients in the `grad` fields of tensors, so one may have to set them to zero before calling it.

This accumulating behavior is desirable in particular to compute the gradient of a loss summed over several "mini-batches," or the gradient of a sum of losses.

So we can run a forward/backward pass on



$$\phi^{(1)}\left(x^{(0)}; w^{(1)}\right) = w^{(1)}x^{(0)}$$

$$\phi^{(2)}\left(x^{(0)}, x^{(1)}; w^{(2)}\right) = x^{(0)} + w^{(2)}x^{(1)}$$

$$\phi^{(3)}\left(x^{(1)}, x^{(2)}; w^{(1)}\right) = w^{(1)}\left(x^{(1)} + x^{(2)}\right)$$

```
w1 = torch.rand(5, 5).requires_grad_()
w2 = torch.rand(5, 5).requires_grad_()
x = torch.randn(5)

x0 = x
x1 = w1 @ x0
x2 = x0 + w2 @ x1
x3 = w1 @ (x1 + x2)

q = x3.norm()

q.backward()
```

The autograd machinery

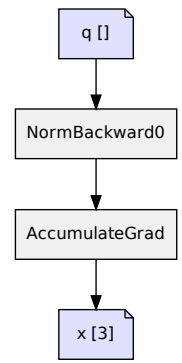
The autograd graph is encoded through the fields `grad_fn` of `Tensor`s, and the fields `next_functions` of `Function`s.

```
>>> x = torch.tensor([ 1.0, -2.0, 3.0, -4.0 ]).requires_grad_()
>>> a = x.abs()
>>> s = a.sum()
>>> s
tensor(10., grad_fn=<SumBackward0>)
>>> s.grad_fn.next_functions
((<AbsBackward object at 0x7ffb2b1462b0>, 0),)
>>> s.grad_fn.next_functions[0][0].next_functions
((<AccumulateGrad object at 0x7ffb2b146278>, 0),)
```

We will come back to this later to write our own `Function`s.

We can visualize the full graph built during a computation.

```
x = torch.tensor([1., 2., 2.]).requires_grad_()
q = x.norm()
```

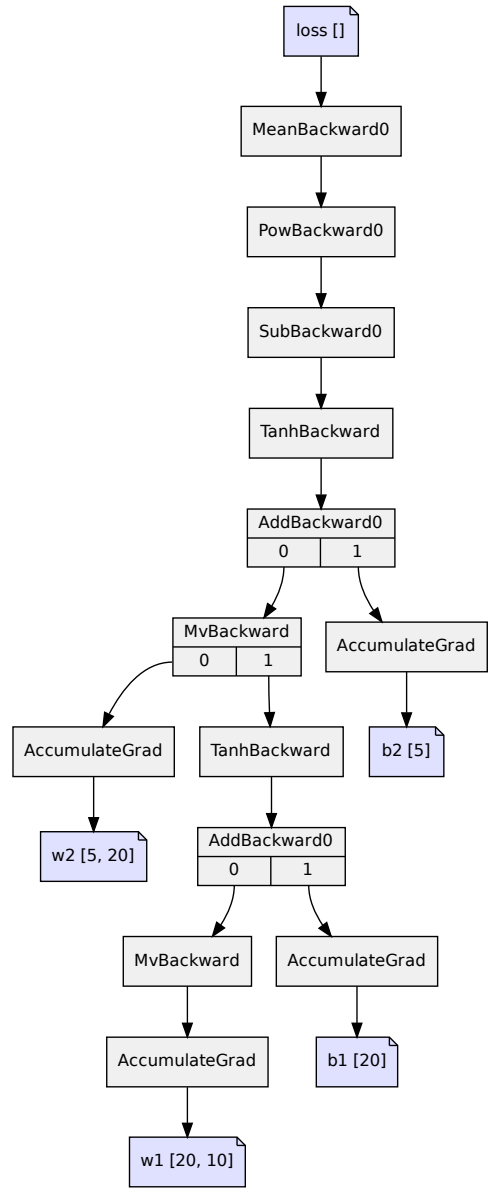


This graph was generated with

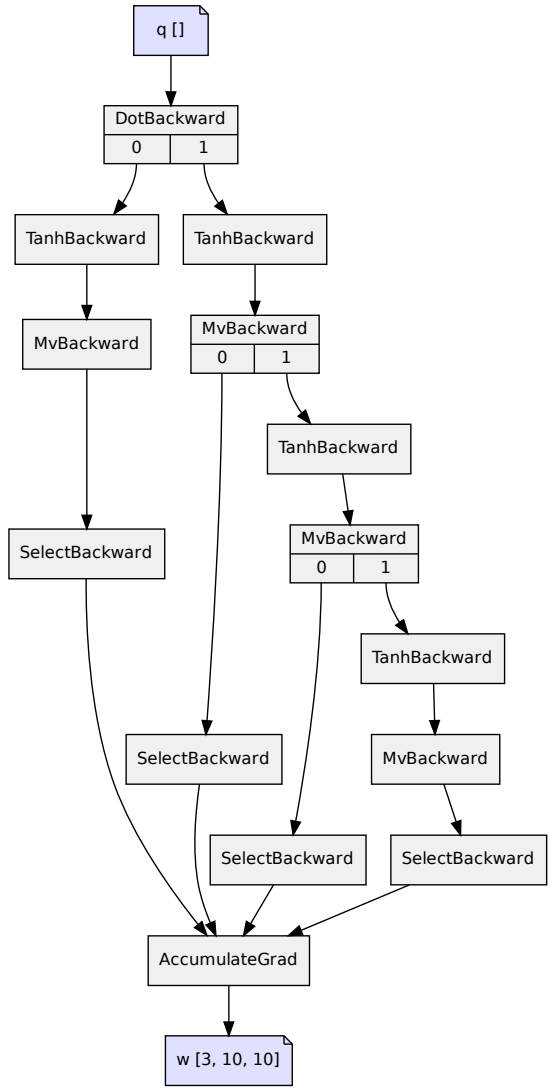
https://fleuret.org/git/agtree2dot

and Graphviz.

```
w1 = torch.rand(20, 10).requires_grad_()
b1 = torch.rand(20).requires_grad_()
w2 = torch.rand(5, 20).requires_grad_()
b2 = torch.rand(5).requires_grad_()

x = torch.rand(10)
h = torch.tanh(w1 @ x + b1)
y = torch.tanh(w2 @ h + b2)

targets = torch.rand(5)

loss = (y - targets).pow(2).mean()
```



```
w = torch.rand(3, 10, 10).requires_grad_()

def blah(k, x):
    for i in range(k):
        x = torch.tanh(w[i] @ x)
    return x

u = blah(1, torch.rand(10))
v = blah(3, torch.rand(10))
q = u.dot(v)
```



Although they are related, **the autograd graph is not the network's structure**, but the graph of operations to compute the gradient. It can be data-dependent and miss or replicate sub-parts of the network.

The `torch.no_grad()` context switches off the autograd machinery, and can be used for operations such as parameter updates.

```
w = torch.empty(10, 784).normal_(0, 1e-3).requires_grad_()
b = torch.empty(10).normal_(0, 1e-3).requires_grad_()

for k in range(10001):
    y_hat = x @ w.t() + b
    loss = (y_hat - y).pow(2).mean()

    w.grad, b.grad = None, None
    loss.backward()

    with torch.no_grad():
        w -= eta * w.grad
        b -= eta * b.grad
```

The `detach()` method creates a tensor which shares the data, but does not require gradient computation, and is not connected to the current graph.

This method should be used when the gradient should not be propagated beyond a variable, or to update leaf tensors.

```
a = torch.tensor( 0.5).requires_grad_()
b = torch.tensor(-0.5).requires_grad_()

for k in range(100):
    l = (a - 1)**2 + (b + 1)**2 + (a - b)**2
    ga, gb = torch.autograd.grad(l, (a, b))
    with torch.no_grad():
        a -= eta * ga
        b -= eta * gb

print(a, b)
```

prints

```
tensor(0.3333, requires_grad=True) tensor(-0.3333, requires_grad=True)
```

```
a = torch.tensor( 0.5).requires_grad_()
b = torch.tensor(-0.5).requires_grad_()

for k in range(100):
    l = (a - 1)**2 + (b + 1)**2 + (a.detach() - b)**2
    ga, gb = torch.autograd.grad(l, (a, b))
    with torch.no_grad():
        a -= eta * ga
        b -= eta * gb

print(a, b)
```

prints

```
tensor(1.0000, requires_grad=True) tensor(-8.2480e-08, requires_grad=True)
```

By default, autograd deletes the computational graph when it is used.

```
>>> x = torch.tensor([1.]).requires_grad_()
>>> z = 1/x
>>> torch.autograd.grad(z, x)
(tensor([-1.]),)
>>> torch.autograd.grad(z * z, x)
Traceback (most recent call last):
/.../
RuntimeError: Trying to backward through the graph a second time, but
the buffers have already been freed.
```

The flag `retain_graph` indicates to keep it.

```
>>> x = torch.tensor([1.]).requires_grad_()
>>> z = 1/x
>>> torch.autograd.grad(z, x, retain_graph = True)
(tensor([-1.]),)
>>> torch.autograd.grad(z * z, x)
(tensor([-2.]),)
```

Autograd can also track the computation of the gradient itself, to allow **higher-order derivatives**. This is specified with `create_graph = True`.

$$\psi(x_1, x_2) = \log(x_1) + x_2^2$$

$$\|\nabla\psi\|_2^2 = \left(\frac{1}{x_1}\right)^2 + (2x_2)^2$$

$$\nabla\|\nabla\psi\|_2^2 = \left(-\frac{2}{x_1^3}, 8x_2\right)$$

```
>>> x = torch.tensor([2., 3.]).requires_grad_()
>>> psi = x[0].log() + x[1].pow(2)
>>> g, = torch.autograd.grad(psi, x, create_graph = True)
>>> torch.autograd.grad(g.pow(2).sum(), x)
(tensor([-0.2500, 24.0000]),)
```

In-place operations may corrupt values required to compute the gradient, and this is tracked down by autograd.

```
>>> x = torch.tensor([1., 2., 3.]).requires_grad_()
>>> y = x.sin()
>>> y *= y
>>> l = y.sum()
>>> l.backward()
Traceback (most recent call last):
/.../
RuntimeError: one of the variables needed for gradient computation
has been modified by an inplace operation
```

They are also prohibited on so-called "leaf" tensors, which are not the results of operations but the initial inputs to the whole computation.

Deep learning

4.3. PyTorch modules and batch processing

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

Elements from `torch.nn.functional` are autograd-compliant functions which compute a result from provided arguments alone.

Subclasses of `torch.nn.Module` are losses and network components. The latter embed parameters to be optimized during training.

Parameters are of the type `torch.nn.Parameter` which is a `Tensor` with `requires_grad` to `True`, and known to be a model parameter by various utility functions, in particular `torch.nn.Module.parameters()`.

Usually `torch.nn.functional` is imported as `F`, and `torch.nn` as `nn`.

Functions and modules from `nn` process **batches** of inputs stored in a tensor whose first dimension indexes them, and produce a corresponding tensor with the same additional dimension.

E.g. a fully connected layer $\mathbb{R}^C \to \mathbb{R}^D$ expects as input a tensor of size $N \times C$ and computes a tensor of size $N \times D$, where N is the number of samples and can vary from a call to another. We come back to this in a second.

The autograd-compliant function

```
F.relu(input, inplace=False)
```

takes a tensor of any size as input, applies ReLU on each value to produce a result tensor of same size.

```
>>> x
tensor([[ 0.8008, -0.2586,  0.5019, -0.2002, -0.7416],
        [ 0.0557,  0.6046,  0.0864, -0.5929,  1.2606]])
>>> F.relu(x)
tensor([[ 0.8008,  0.0000,  0.5019,  0.0000,  0.0000],
        [ 0.0557,  0.6046,  0.0864,  0.0000,  1.2606]])
```

`inplace` indicates if the operation should modify the argument itself. This may be desirable to reduce the memory footprint of the processing.

The module

```
nn.Linear(in_features, out_features, bias=True)
```

implements a $\mathbb{R}^C \to \mathbb{R}^D$ fully-connected layer. It takes as input a tensor of size $N \times C$ and produce a tensor of size $N \times D$.

```
>>> f = nn.Linear(in_features = 10, out_features = 4)
>>> for n, p in f.named_parameters(): print(n, p.size())
...
weight torch.Size([4, 10])
bias torch.Size([4])
>>> x = torch.randn(523, 10)
>>> y = f(x)
>>> y.size()
torch.Size([523, 4])
```

The weights and biases are automatically randomized at creation. We will come back to that later.

The module

```
nn.MSELoss()
```

implements the Mean Square Error loss: the sum of the component-wise squared difference, **divided by the total number of components in the tensors**.

```
>>> f = nn.MSELoss()
>>> x = torch.tensor([[ 3. ]])
>>> y = torch.tensor([[ 0. ]])
>>> f(x, y)
tensor(9.)
>>> x = torch.tensor([[ 3., 0., 0., 0. ]])
>>> y = torch.tensor([[ 0., 0., 0., 0. ]])
>>> f(x, y)
tensor(2.2500)
```

The first parameter of a loss is traditionally called the **input** and the second the **target**. These two quantities may be of different dimensions or even types for some losses (e.g. for classification).

Criteria do not accept a target with `requires_grad` to `True`.

```
>>> import torch
>>> f = nn.MSELoss()
>>> x = torch.tensor([ 3., 2. ]).requires_grad_()
>>> y = torch.tensor([ 0., -2. ]).requires_grad_()
>>> f(x, y)
Traceback (most recent call last):
/.../
AssertionError: nn criterions don't compute the gradient w.r.t.
targets - please mark these tensors as not requiring gradients
```

Batch processing

Functions and modules from `nn` process samples by batches. This is motivated by the computational speed-up it induces.

Training a large network on CIFAR10:

Batch size	Time per epoch
1	4h22min
64	4min50s

speed up of $\times 54$.

To evaluate a module on a sample, both the module's parameters and the sample have to be first copied into **cache memory**, which is fast but small.

For any model of reasonable size, only a fraction of its parameters can be kept in cache, so a module's parameters have to be copied there every time they are used.

Memory transfers are slower than computation. Batch processing cuts down to one copy of the parameters to the cache per batch.

It also cuts down the use of Python loops, which are awfully slow.

Consider a model composed of three modules

$$f = f_3 \circ f_2 \circ f_1,$$

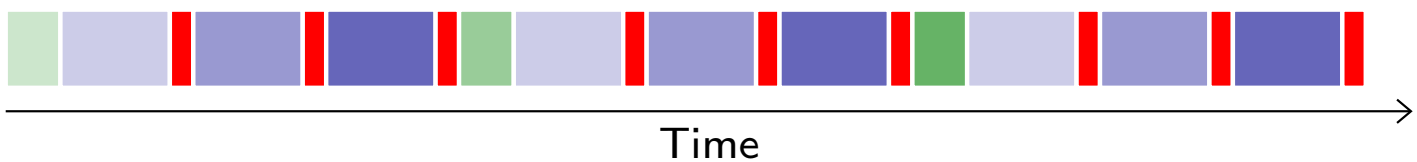
and we want to compute $f(x_1), f(x_2), f(x_3)$.

Copying the x_ns to cache memory

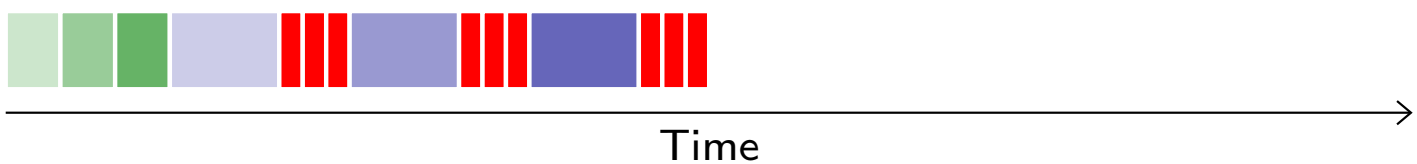
Copying the f_ds' parameters to cache memory

Computing a $f_d(.)$

Processing samples one by one:



Batch processing:



With

```
def timing(x, w, batch = False, nb = 101):
    t = torch.zeros(nb)

    for u in range(nb):
        t0 = time.perf_counter()
        if batch:
            y = x.mm(w.t())
        else:
            y = torch.empty(x.size(0), w.size(0))
            for k in range(y.size(0)): y[k] = w.mv(x[k])
        y.is_cuda and torch.cuda.synchronize()
        t[u] = time.perf_counter() - t0

    return t.median().item()
```

```
x = torch.randn(2500, 1000)
w = torch.randn(1500, 1000)
print('Batch-processing speed-up on CPU %.1f' %
      (timing(x, w, batch = False) / timing(x, w, batch = True)))

x, w = x.to('cuda'), w.to('cuda')
print('Batch-processing speed-up on GPU %.1f' %
      (timing(x, w, batch = False) / timing(x, w, batch = True)))
```

prints

```
Batch-processing speed-up on CPU 4.6
Batch-processing speed-up on GPU 144.4
```

Formally, we have to revisit a bit some expressions we saw previously for fully connected layers. We had

$$\forall l, n, \;\; w^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}, \;\; x_n^{(l-1)} \in \mathbb{R}^{d_{l-1}}, \quad s_n^{(l)} = w^{(l)} \, x_n^{(l-1)}.$$

From now on, we will use row vectors, so that we can represent a series of samples as a 2d array with the first index being the sample's index.

$$x = \begin{pmatrix} x_{1,1} & \cdots & x_{1,D} \\ \vdots & \ddots & \vdots \\ x_{N,1} & \cdots & x_{N,D} \end{pmatrix} = \begin{pmatrix} (x_1)^\top \\ \vdots \\ (x_N)^\top \end{pmatrix},$$

which is an element of $\mathbb{R}^{N \times D}$.

To make all sample row vectors and apply a linear operator, we want

$$\forall n,\ s_n^{(l)} = \left(w^{(l)} \left(x_n^{(l-1)}\right)^\top\right)^\top = x_n^{(l-1)} \left(w^{(l)}\right)^\top$$

which gives a tensorial expression for the full batch

$$s^{(l)} = x^{(l-1)} \left(w^{(l)}\right)^\top.$$

And in `torch/nn/functional.py`

```
def linear(input, weight, bias=None):
    if input.dim() == 2 and bias is not None:
        # fused op is marginally faster
        return torch.addmm(bias, input, weight.t())

    output = input.matmul(weight.t())
    if bias is not None:
        output += bias
    return output
```

Similarly for the backward pass of a linear layer we get

$$\left[\!\left[\frac{\partial \mathscr{L}}{\partial w^{(l)}} \right]\!\right] = \left[\!\left[\frac{\partial \mathscr{L}}{\partial s^{(l)}} \right]\!\right]^{\top} x^{(l-1)},$$

and

$$\left[\!\left[\frac{\partial \mathscr{L}}{\partial x^{(l)}} \right]\!\right] = \left[\!\left[\frac{\partial \ell}{\partial s^{(l+1)}} \right]\!\right] w^{(l+1)}.$$

Deep learning

4.4. Convolutions

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

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

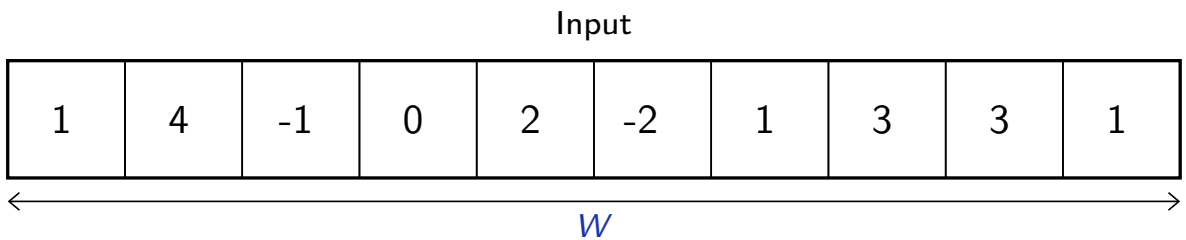
For instance a linear layer taking a 256×256 RGB image as input, and producing an image of same size would require

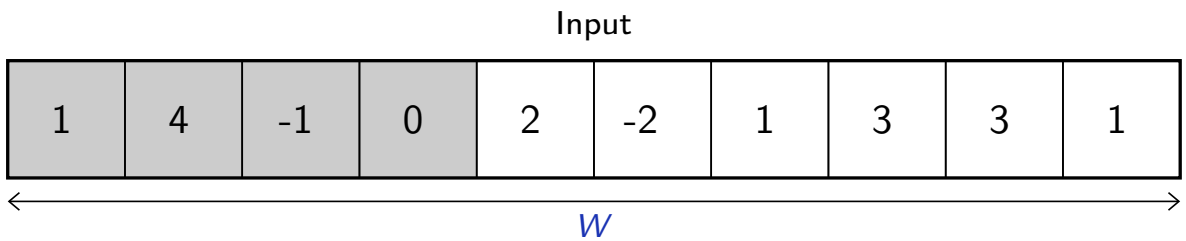
$$(256 \times 256 \times 3)^2 \simeq 3.87e{+}10$$

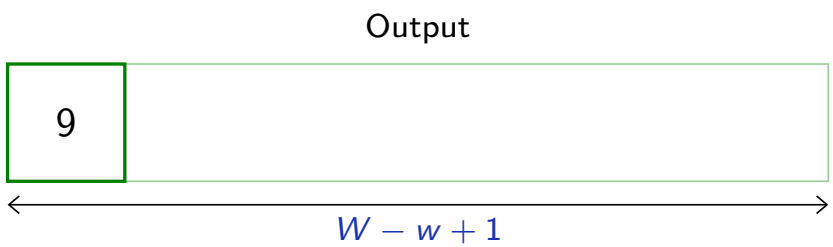
parameters, with the corresponding memory footprint ($\simeq$150Gb !), and excess of capacity.

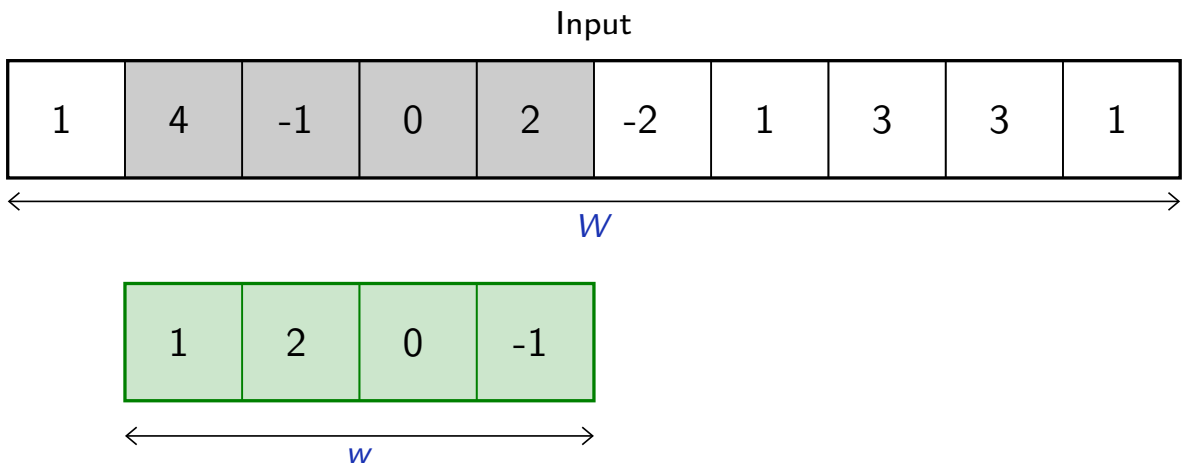
Moreover, this requirement is inconsistent with the intuition that such large signals have some "invariance in translation". **A transformation meaningful at a certain location can / should be used everywhere.**

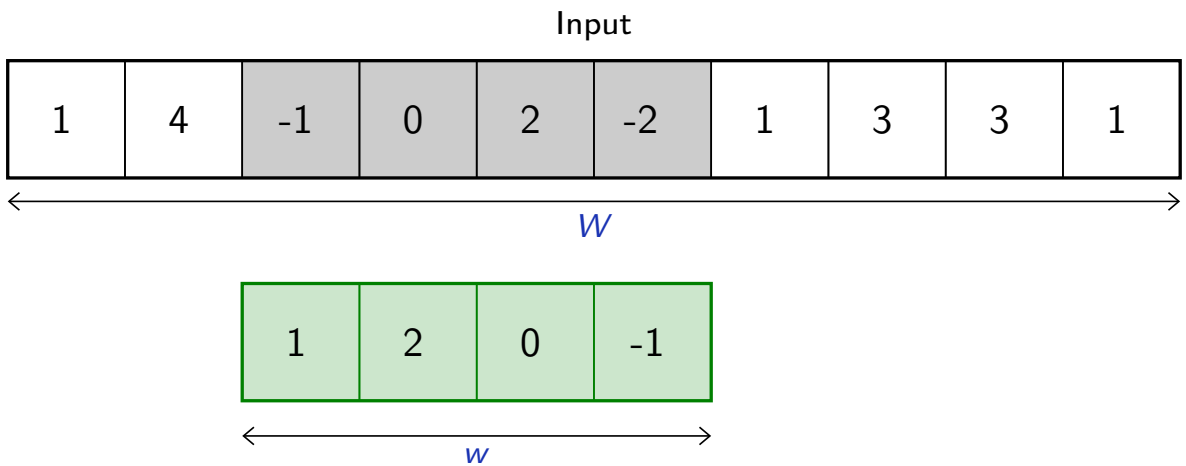
A convolution layer embodies this idea. **It applies the same linear transformation locally, everywhere**

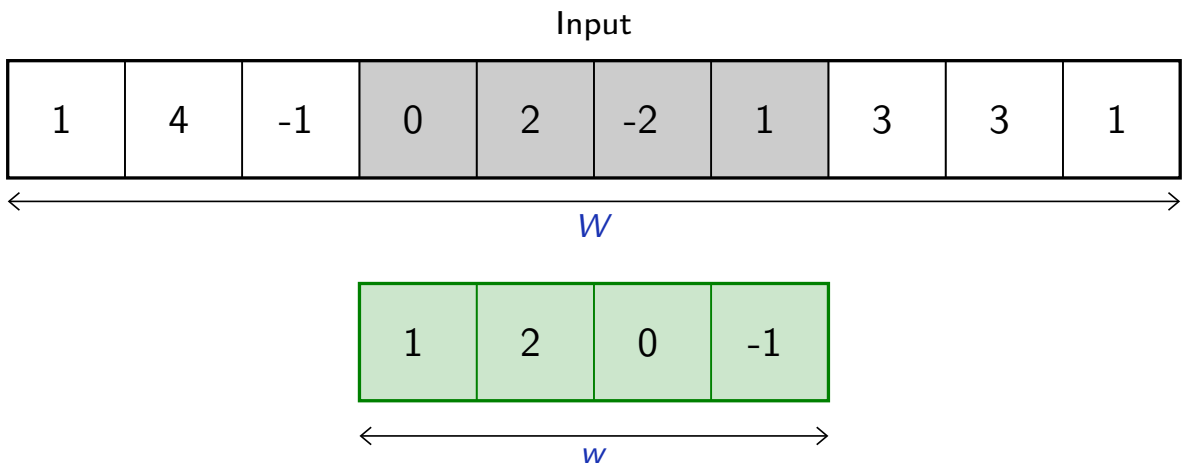

Input
1
4
-1
0
2
-2
1
3
3
1
W
Kernel
1
2
0
-1
w
Output
W − w + 1

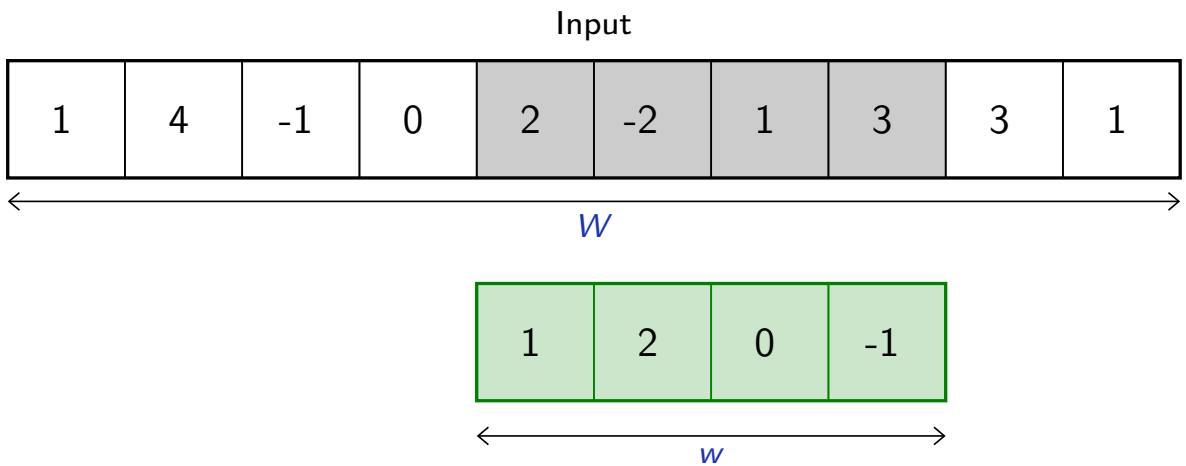

Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w

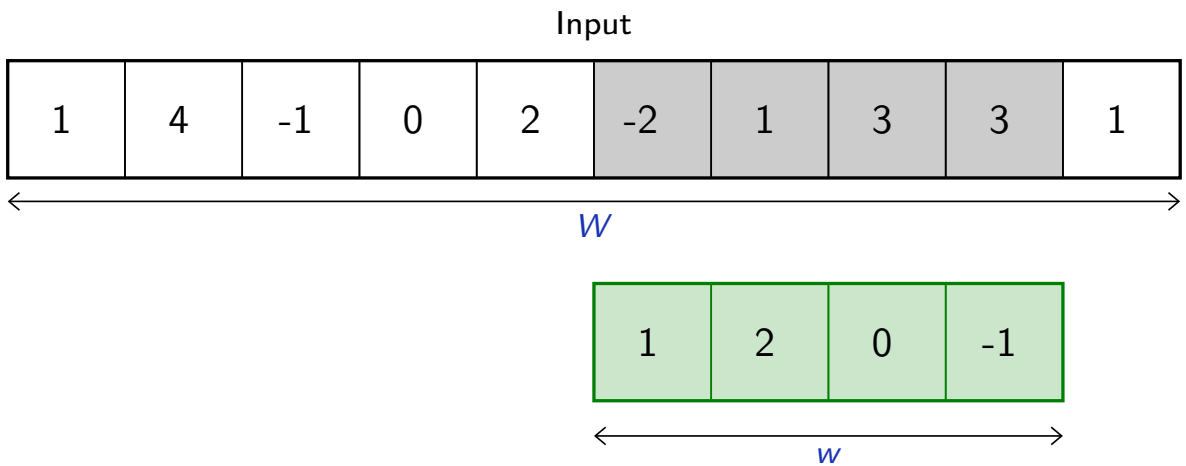

Output
9
W − w + 1

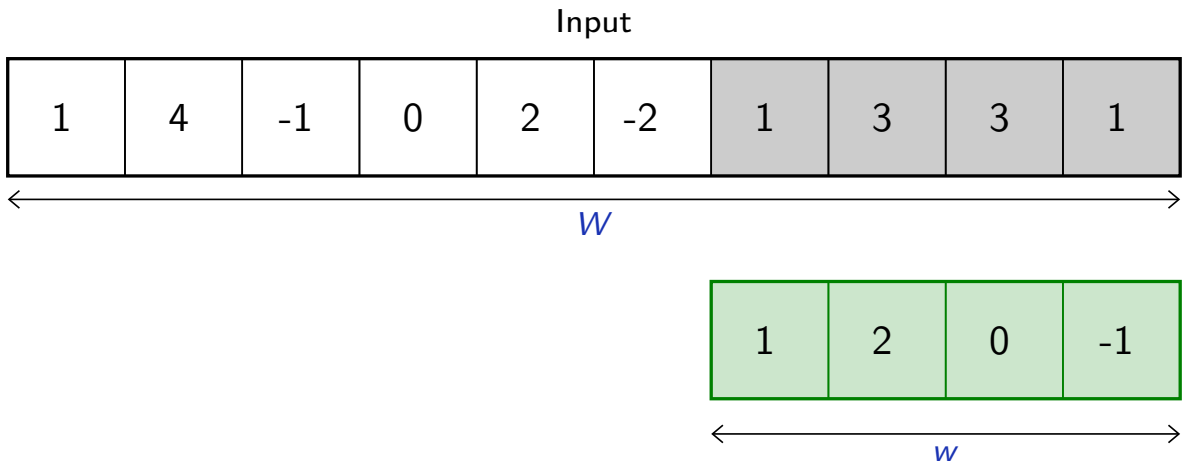

Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w
Output
9
0
W − w + 1

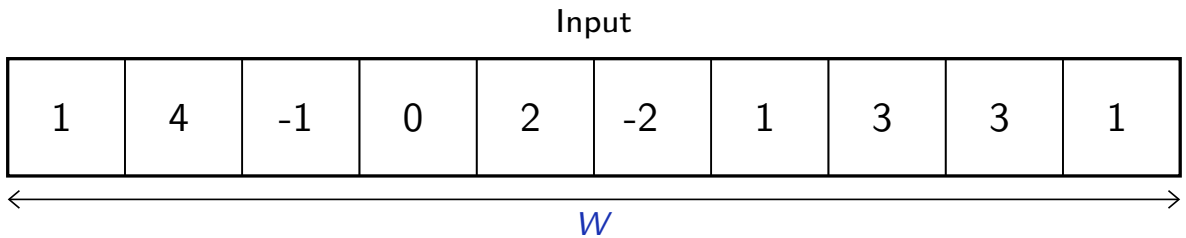

Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w
Output
9
0
1
W − w + 1


Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w
Output
9
0
1
3
W − w + 1


Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w
Output
9
0
1
3
-5
W − w + 1


Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w
Output
9
0
1
3
-5
-3
W − w + 1


Input
1
4
-1
0
2
-2
1
3
3
1
W
1
2
0
-1
w
Output
9
0
1
3
-5
-3
6
W − w + 1


Input
1
4
-1
0
2
-2
1
3
3
1
W
Kernel
1
2
0
-1
w
Output
9
0
1
3
-5
-3
6
W − w + 1

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

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

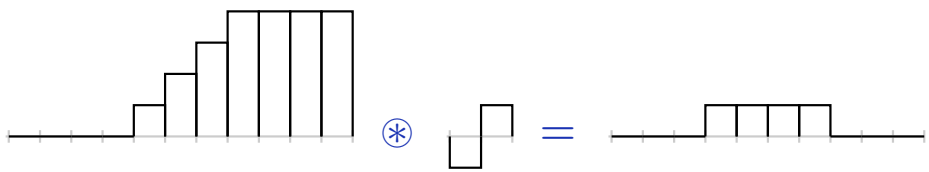
for instance

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

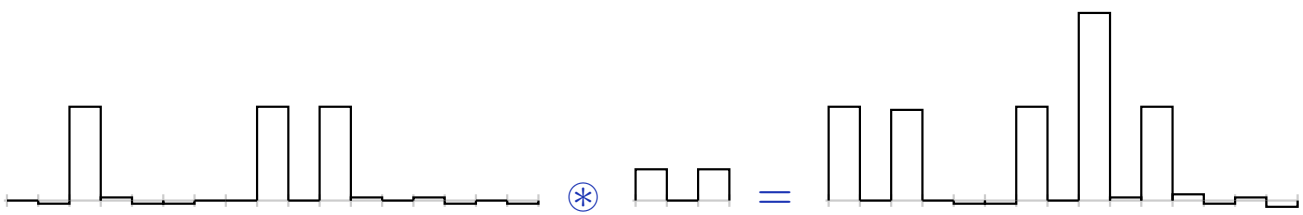
This differs from the usual convolution since the kernel and the signal are both visited in increasing index order.

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.



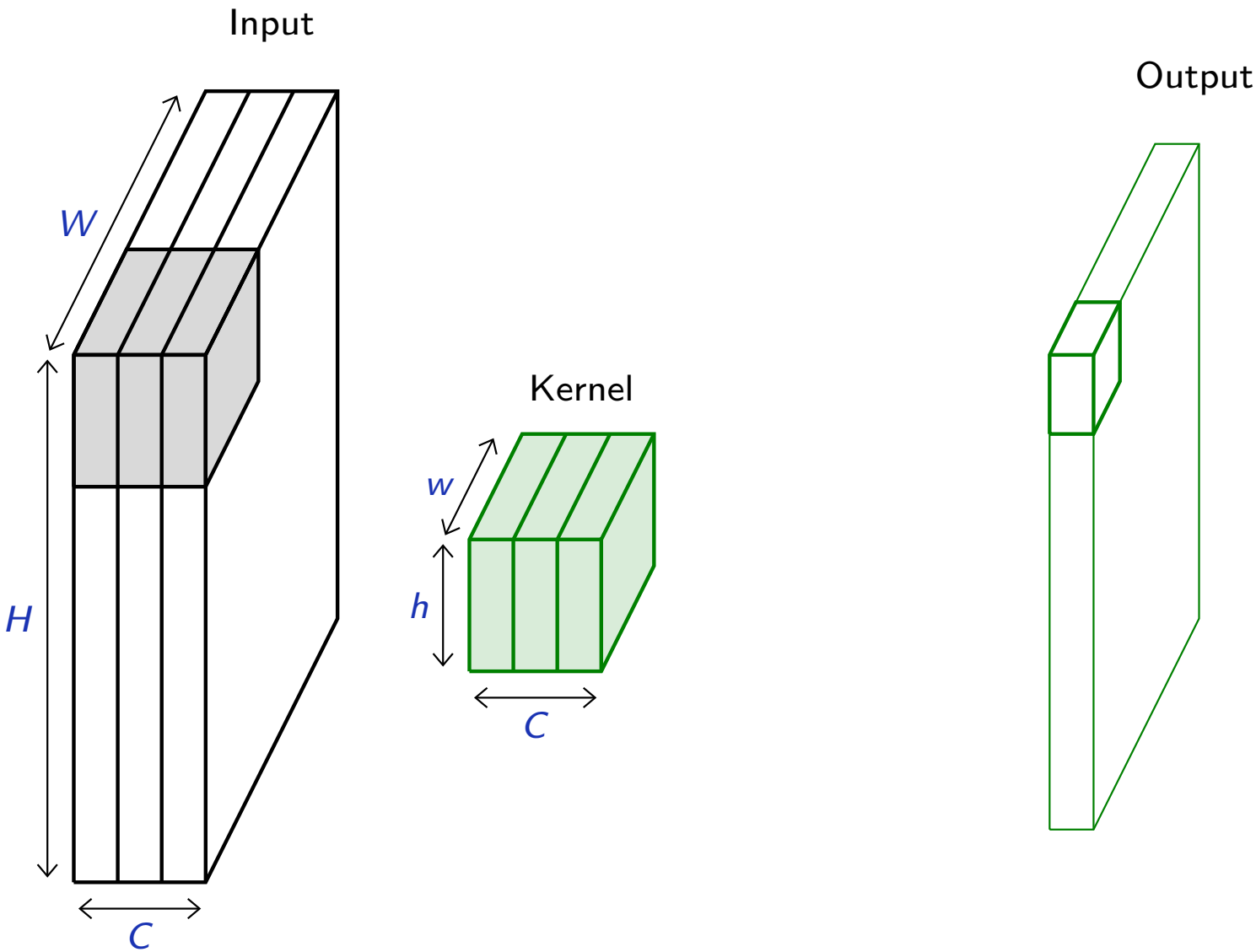
It generalizes naturally to a multi-dimensional input, although specification can become complicated.

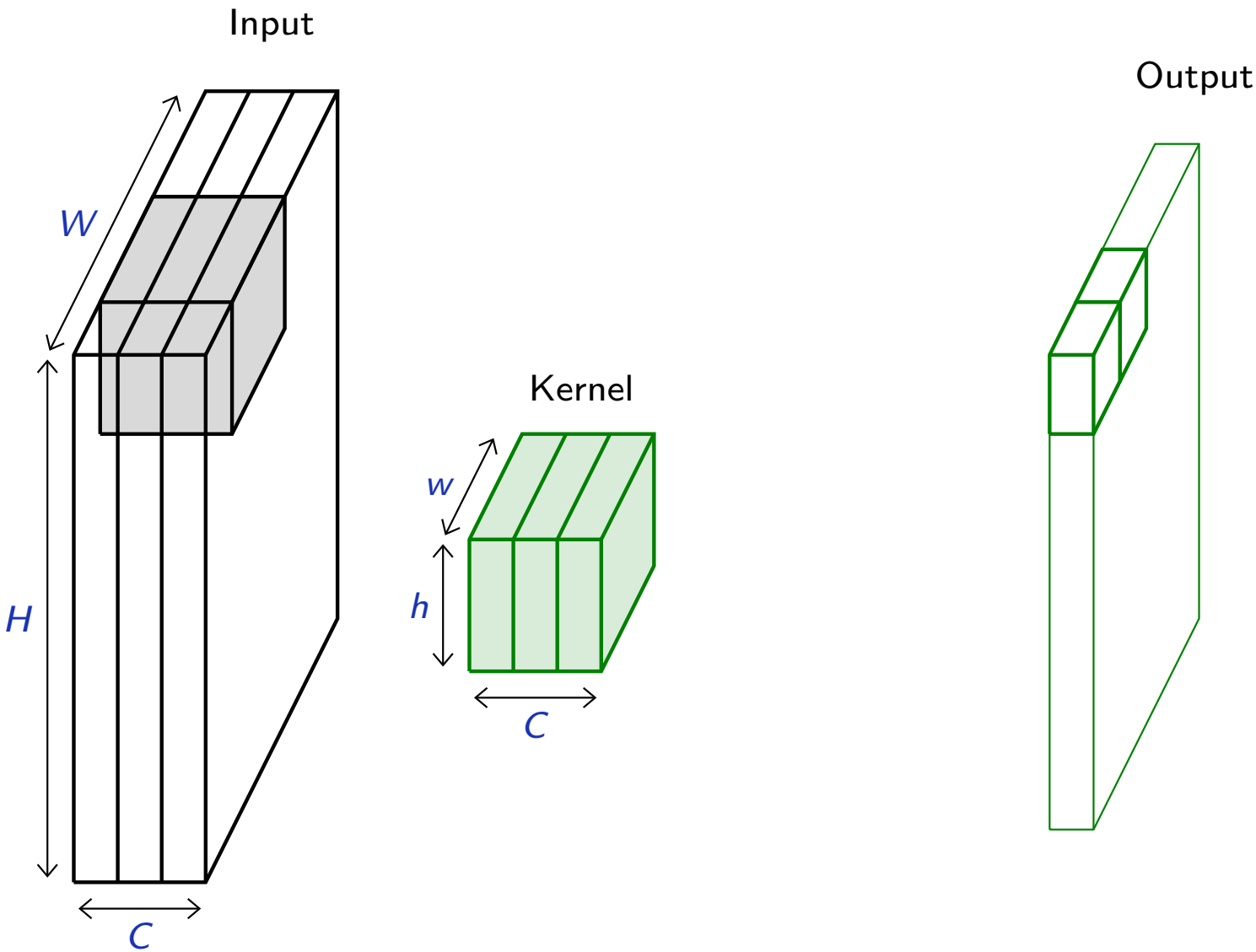
Its most usual form for "convolutional networks" processes a 3d tensor as input (i.e. a multi-channel 2d signal) to output a 2d tensor. The kernel is not swiped across channels, just across rows and columns.

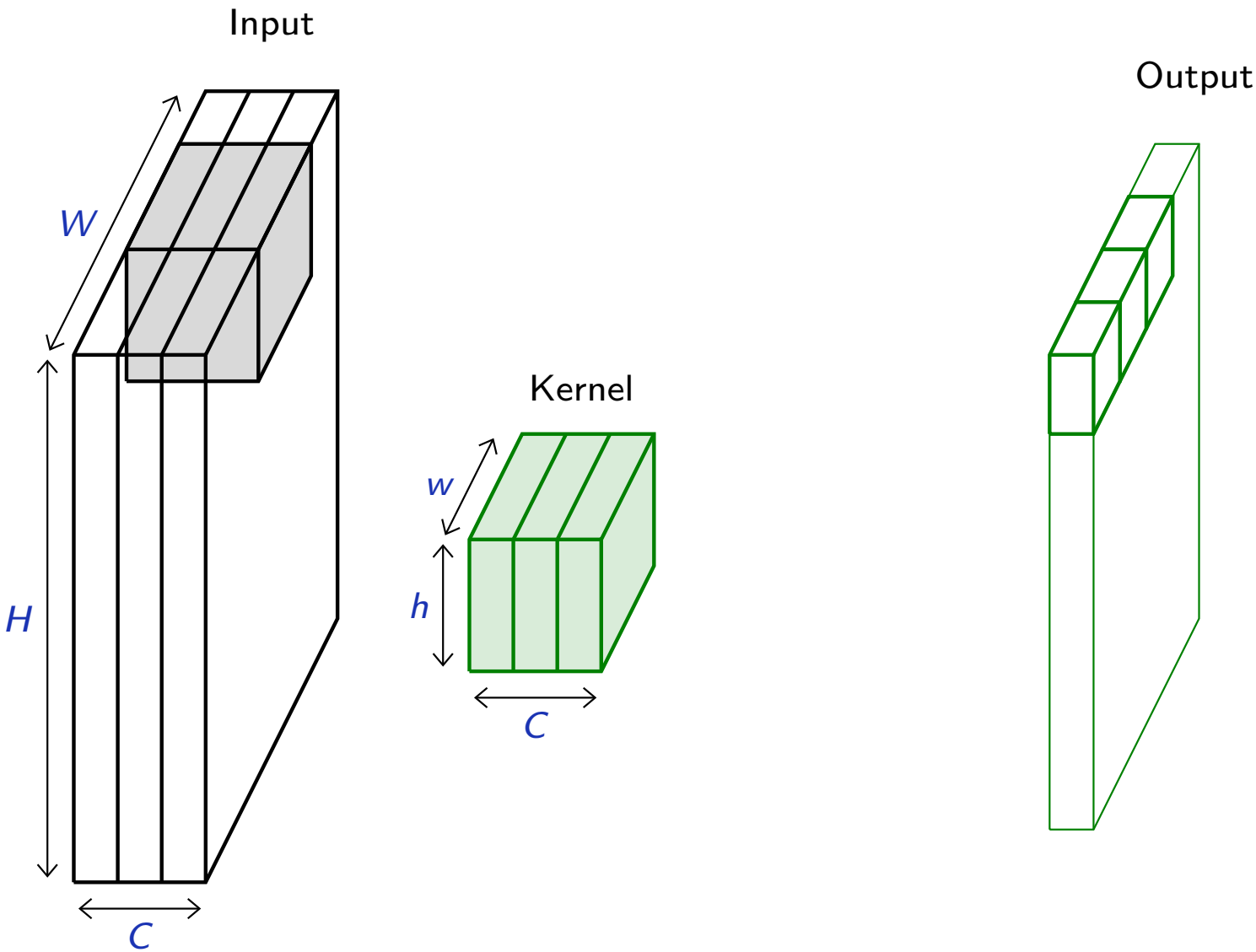
In this case, if the input tensor is of size $C \times H \times W$, and the kernel is $C \times h \times w$, the output is $(H - h + 1) \times (W - w + 1)$.

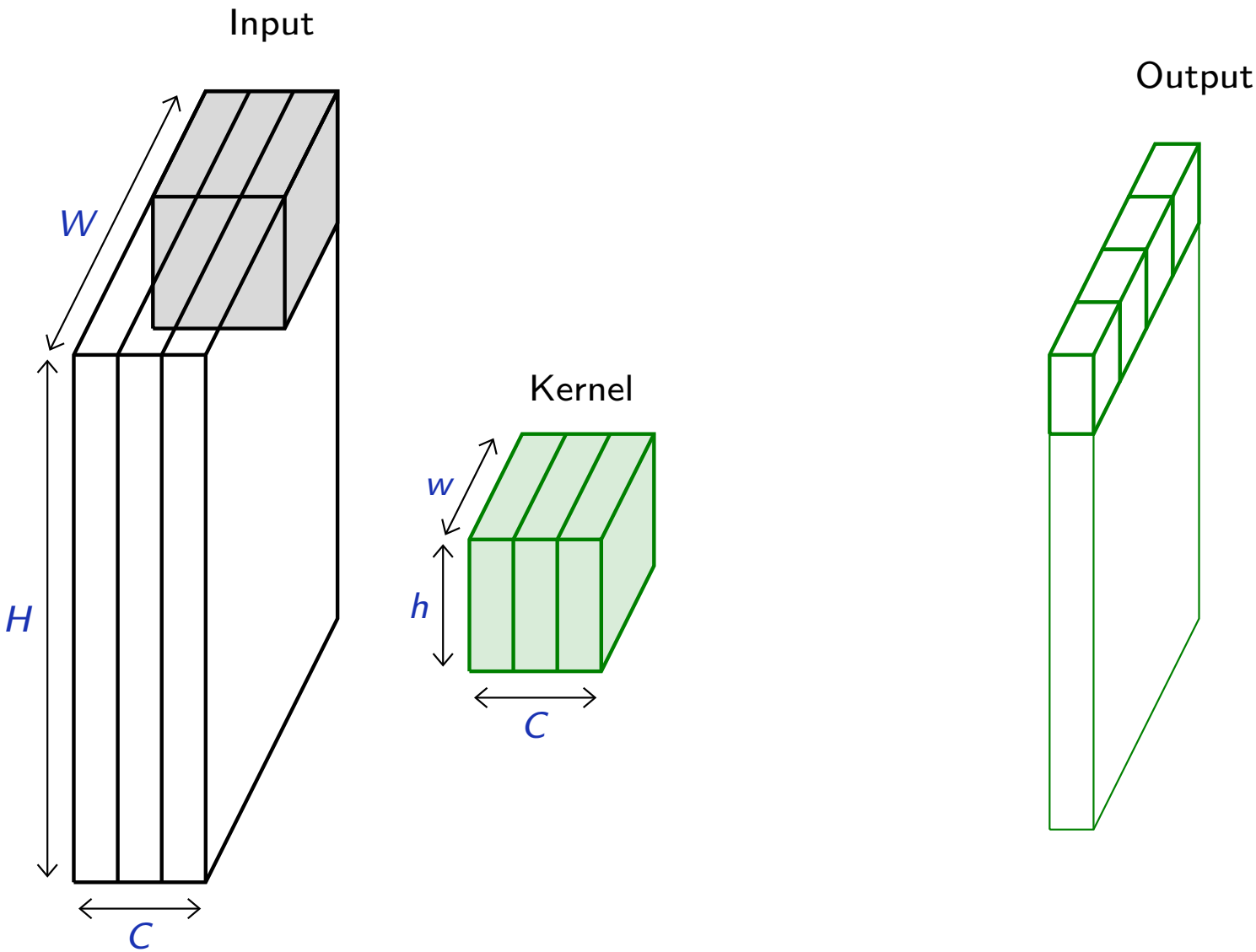
We say "2d signal" even though it has C channels, since it is a feature vector indexed by a 2d location without structure on the feature indexes.

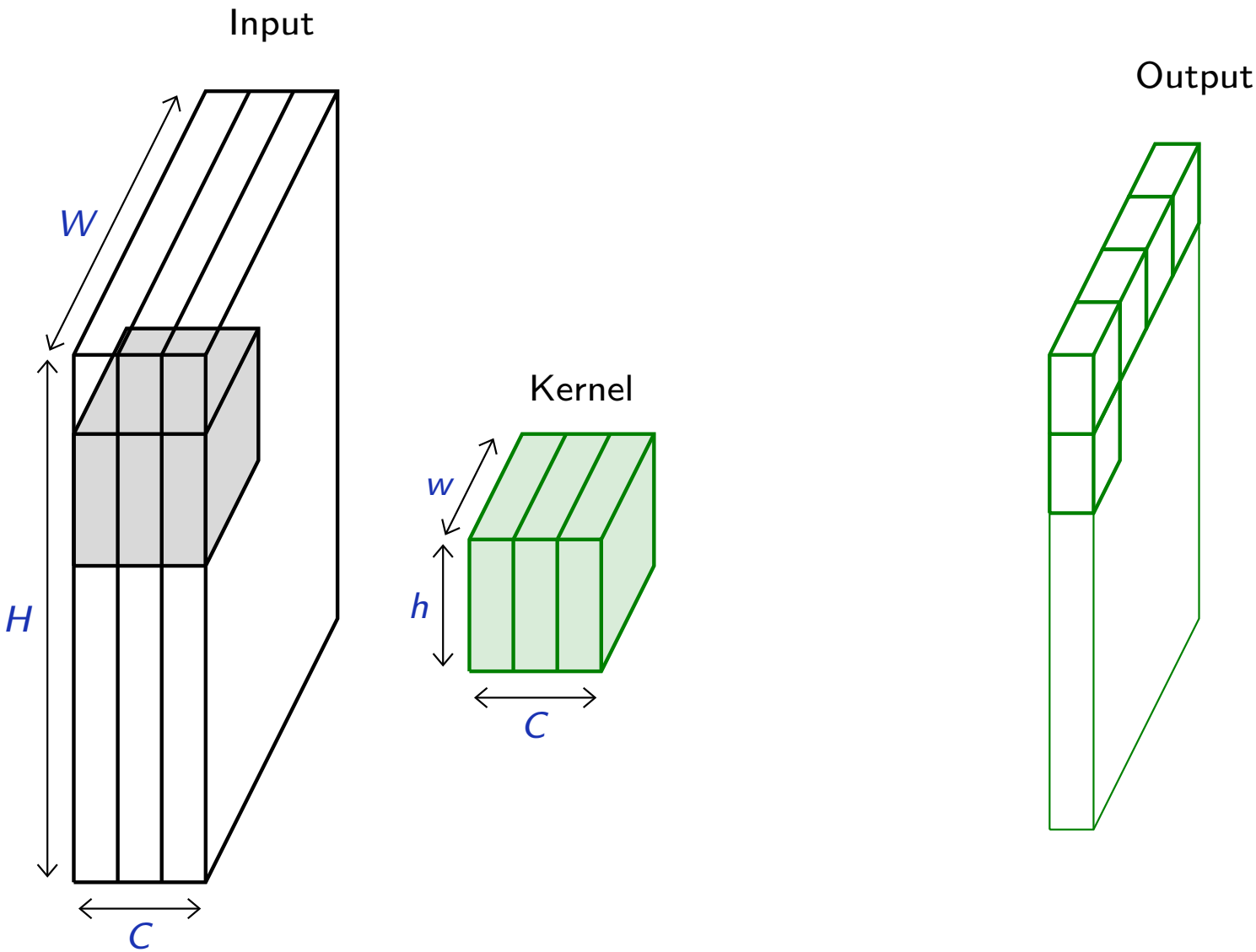
In a standard convolution layer, D such convolutions are combined to generate a $D \times (H - h + 1) \times (W - w + 1)$ output.

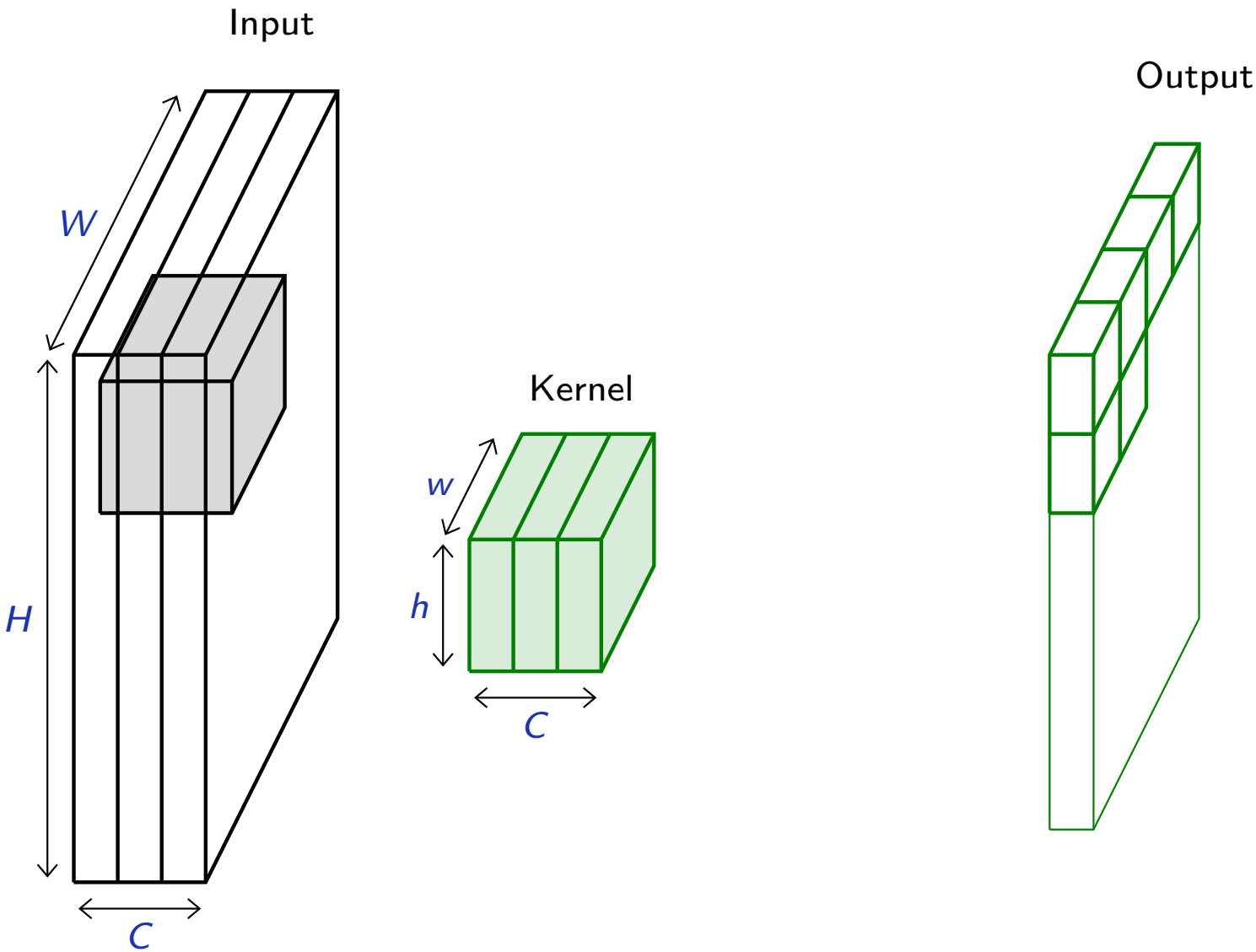

Input
W
H
C
Kernel
w
h
C
Output

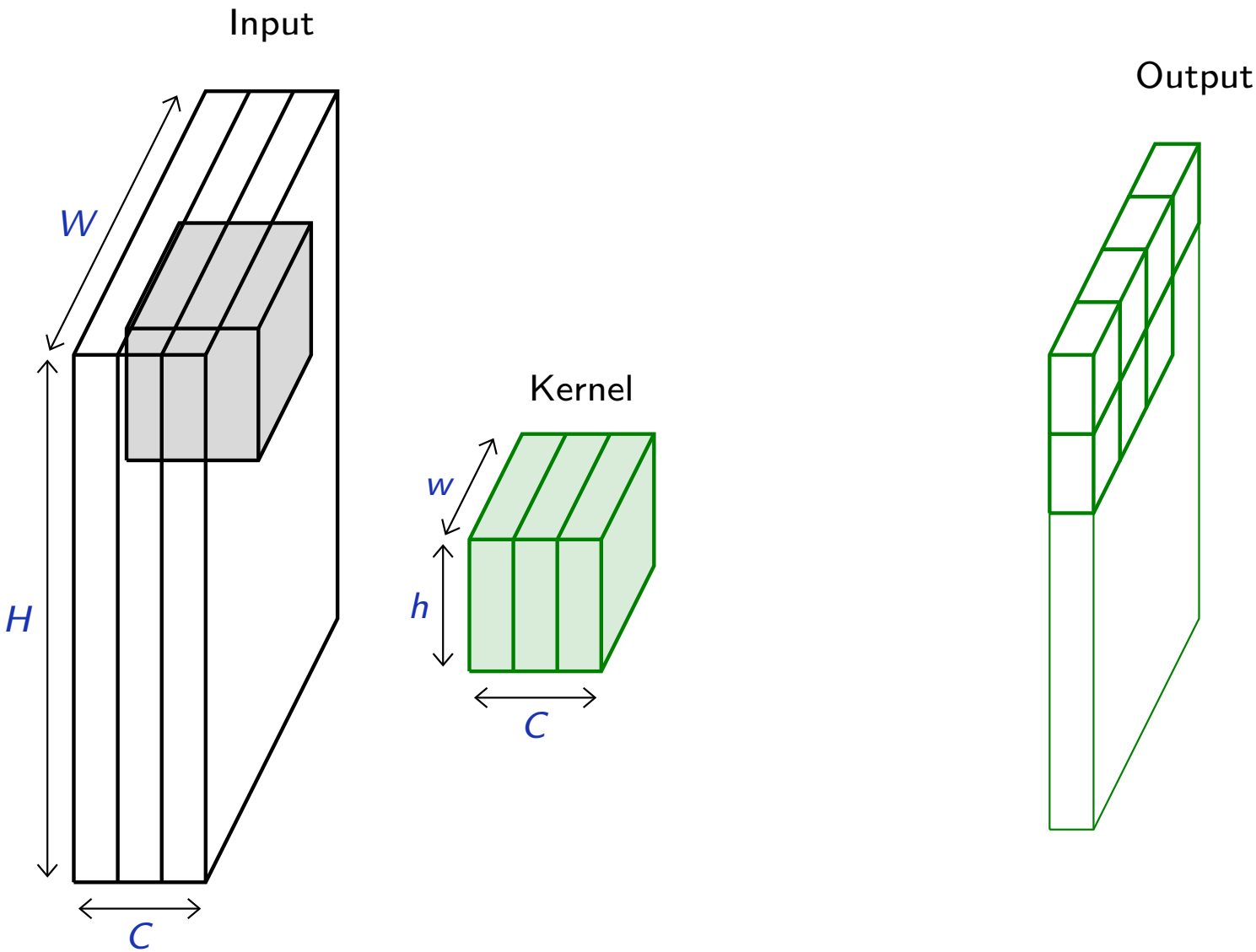

Input
W
H
C
Kernel
w
h
C
Output

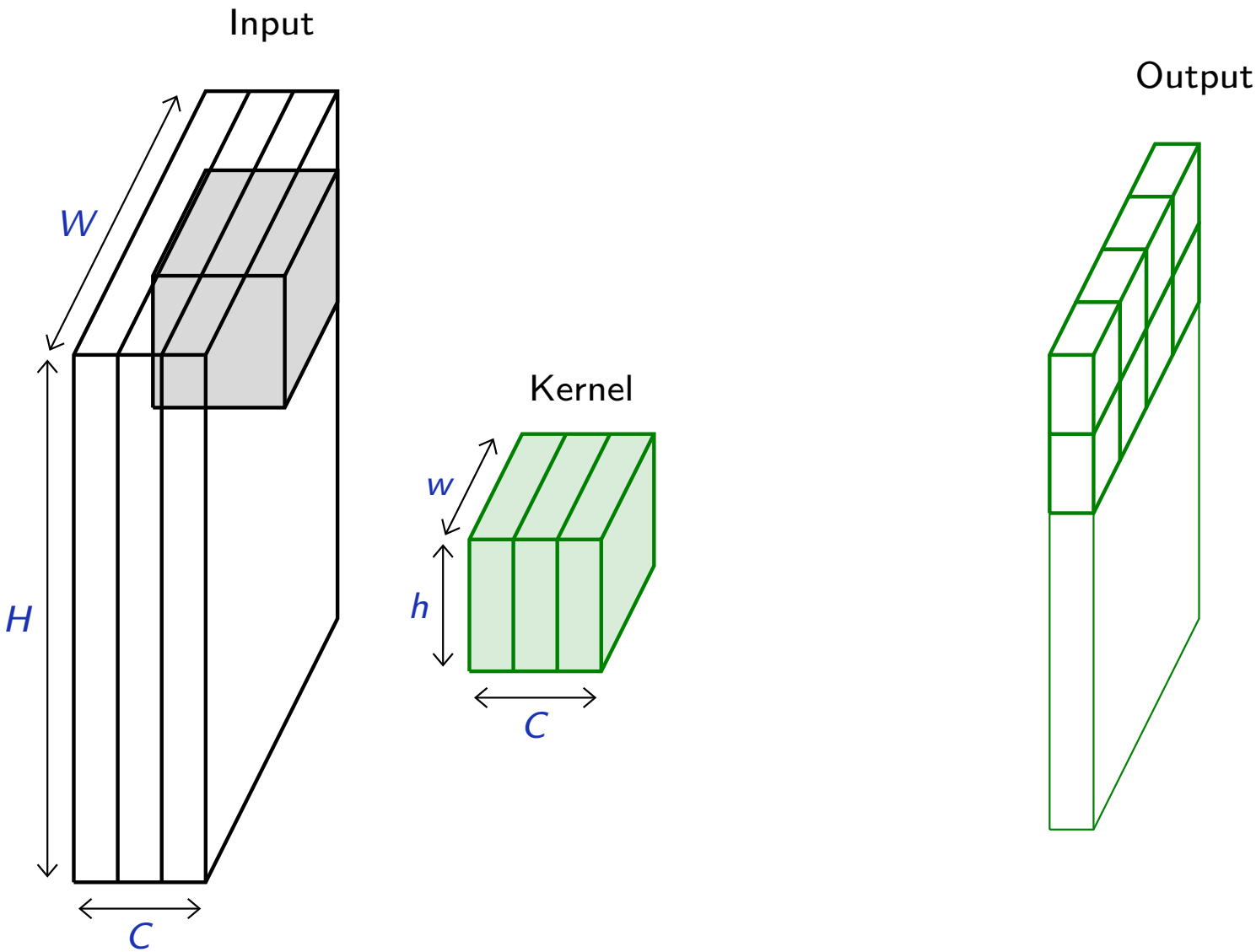

Input
W
H
C
Kernel
w
h
C
Output

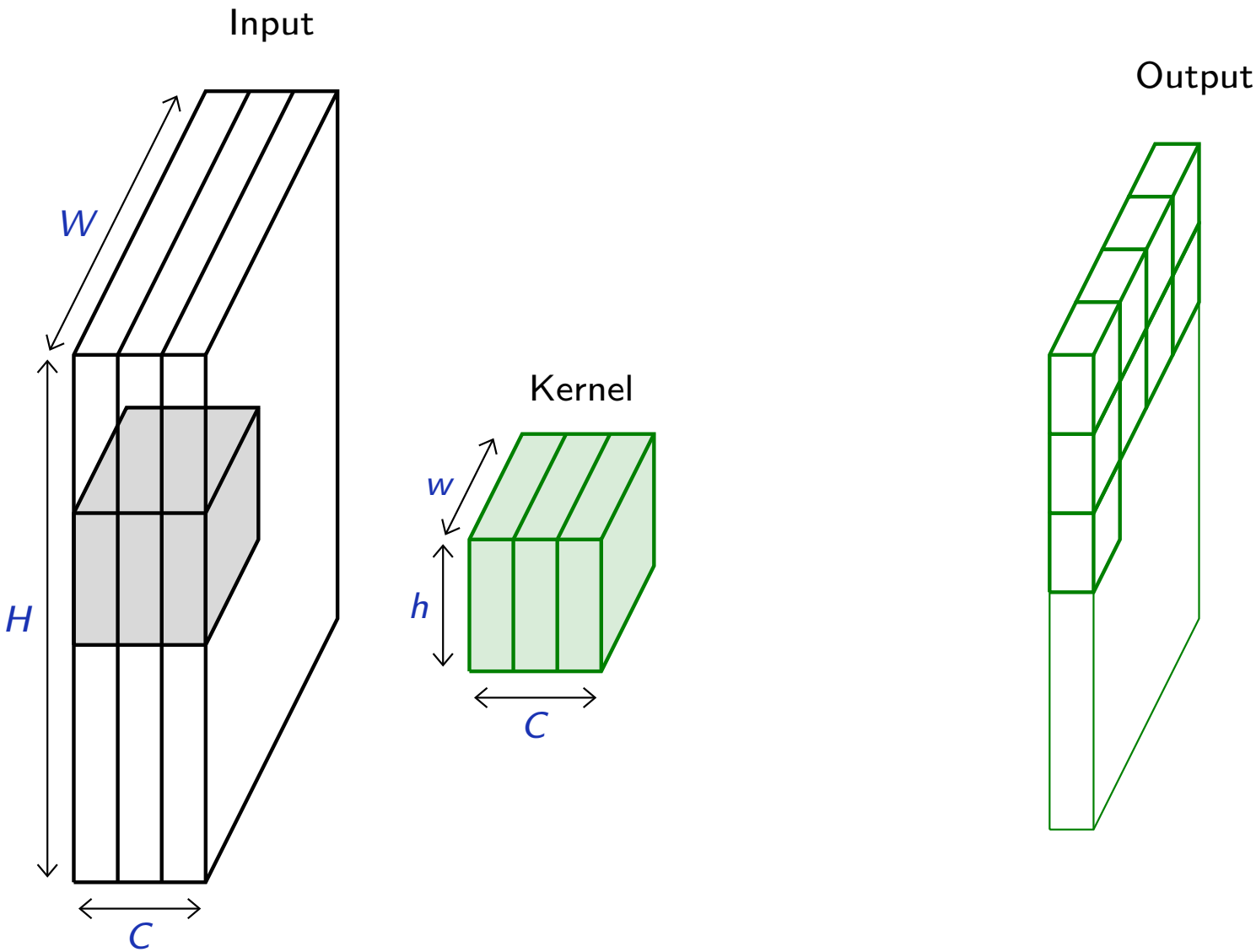

Input
W
H
C
Kernel
w
h
C
Output

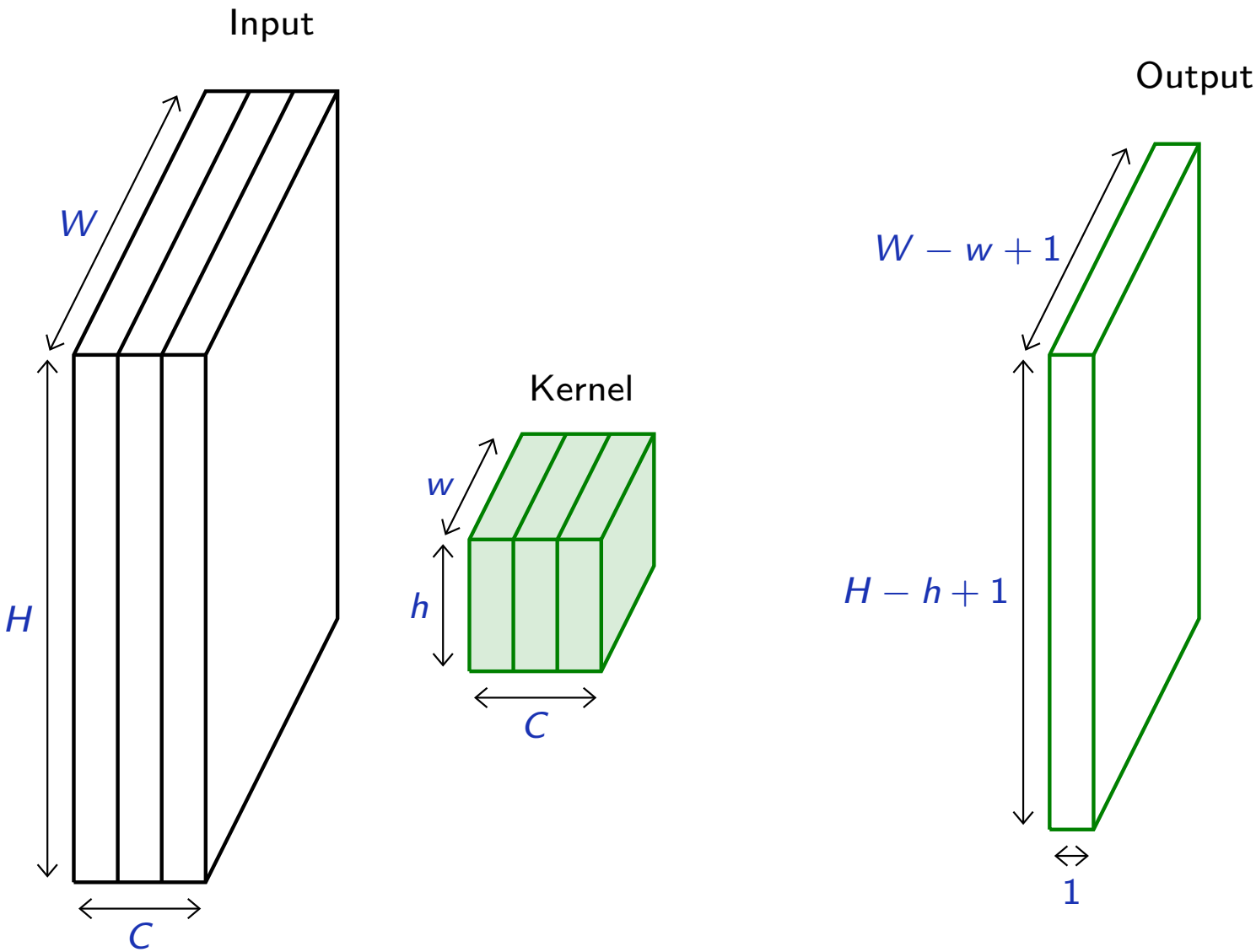

Input
W
H
C
Kernel
w
h
C
Output

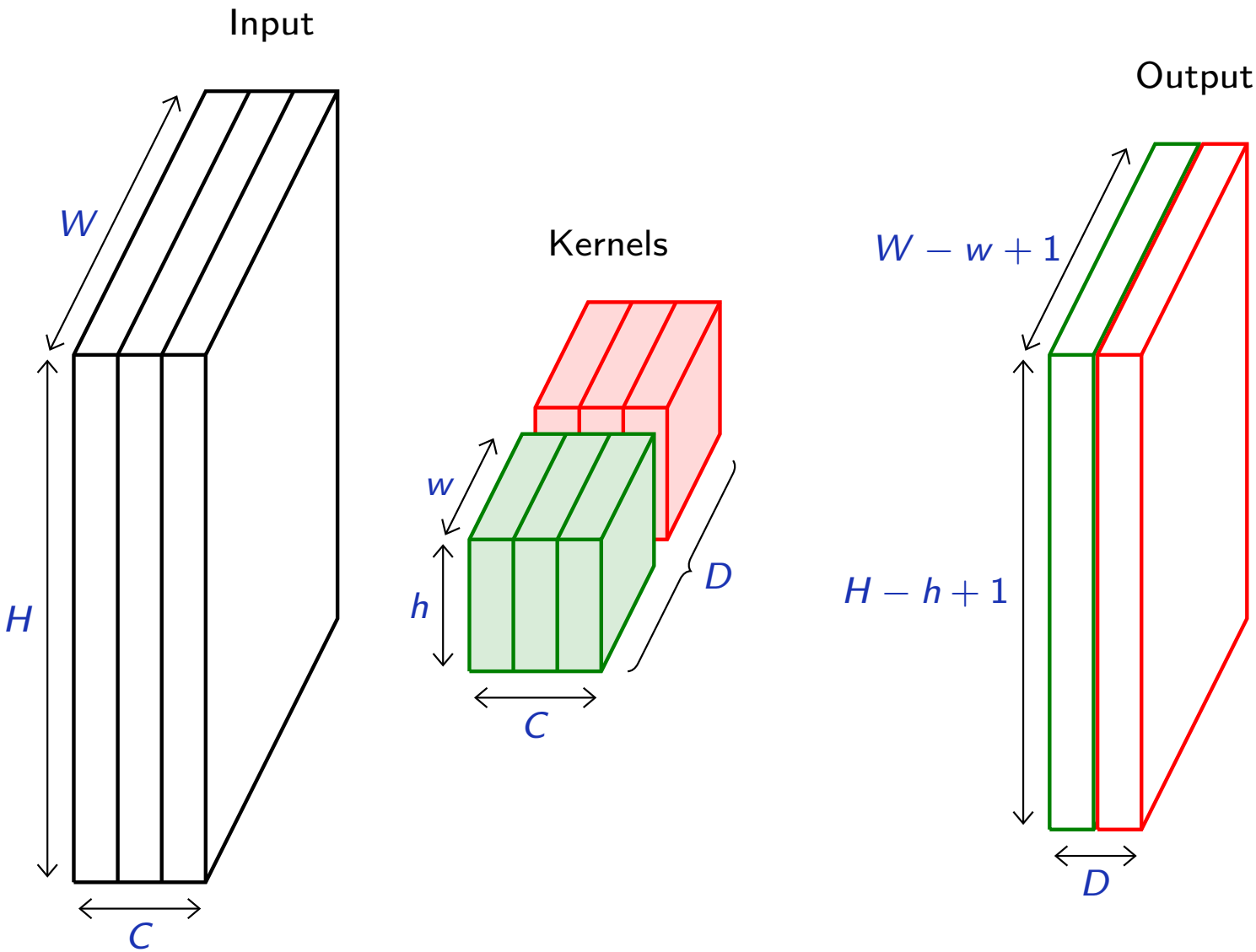

Input
W
H
C
Kernel
w
h
C
Output

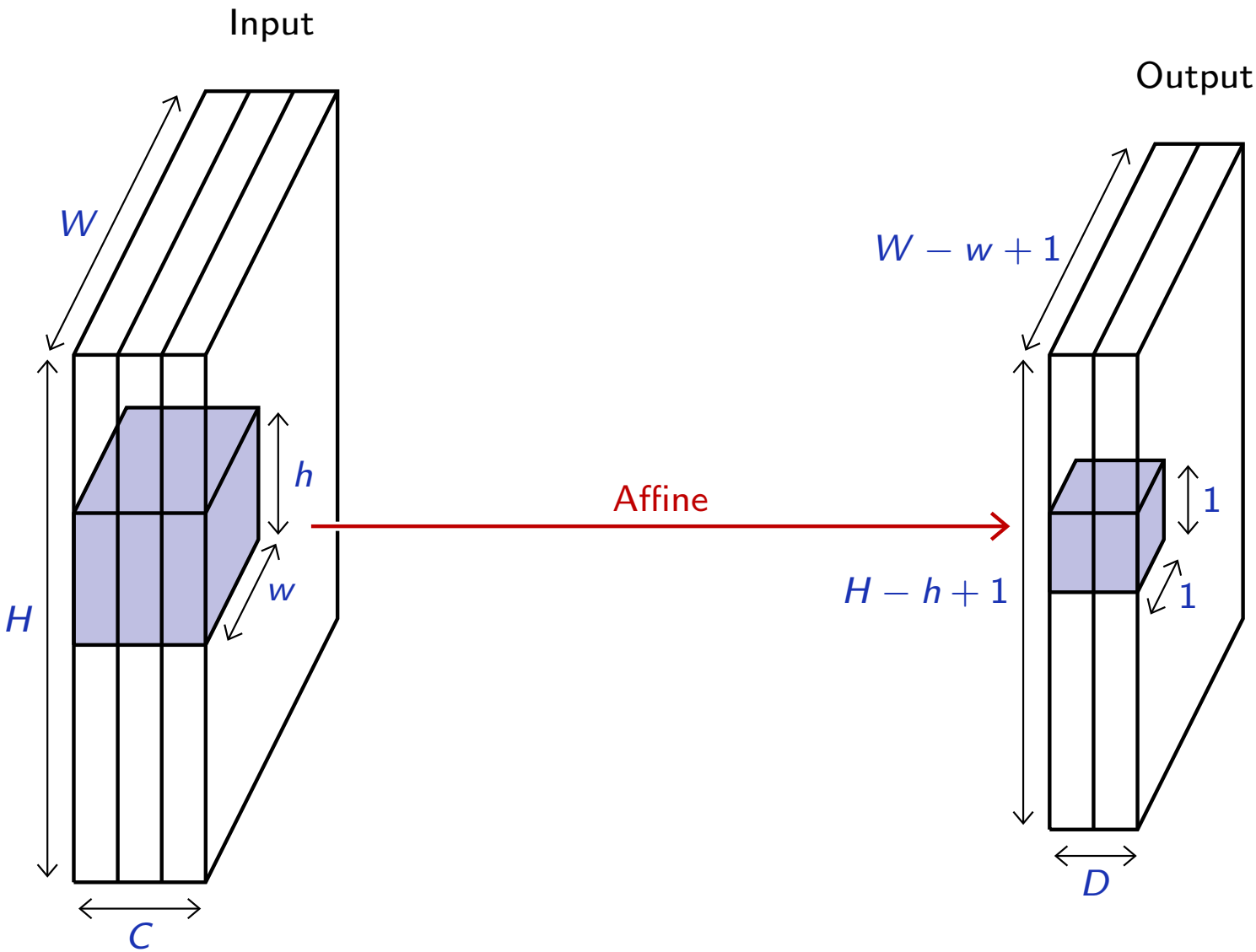

Input
W
H
C
Kernel
w
h
C
Output


Input
W
H
C
Kernel
w
h
C
Output


Input
W
H
C
Kernel
w
h
C
Output


Input
W
H
C
Kernel
w
h
C
Output
W − w + 1
H − h + 1
1


Input
W
H
C
Kernels
w
h
D
C
Output
W − w + 1
H − h + 1
D


Input
Output
W
h
w
H
C
Affine
W − w + 1
H − h + 1
1
1
D

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

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

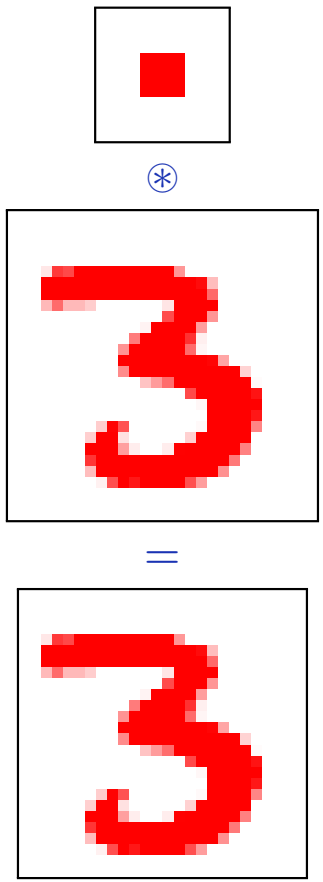
and the result is of dimension

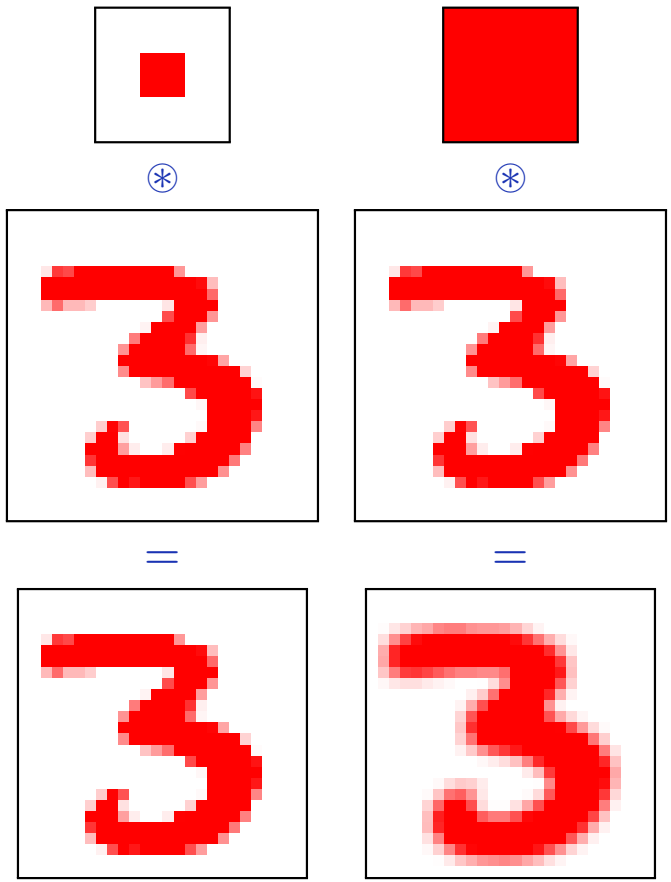
$$N \times D \times (H - h + 1) \times (W - w + 1).$$

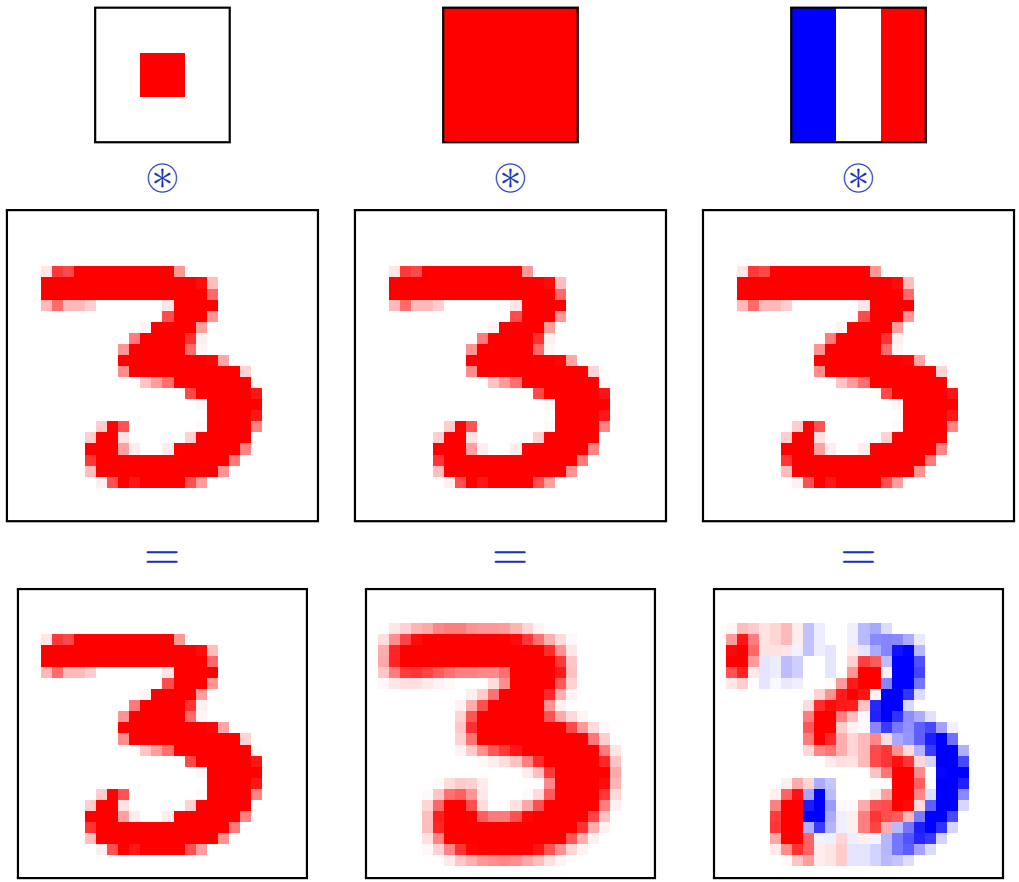
```
>>> weight = torch.randn(5, 4, 2, 3)
>>> bias = torch.randn(5)
>>> input = torch.randn(117, 4, 10, 3)
>>> output = F.conv2d(input, weight, bias)
>>> output.size()
torch.Size([117, 5, 9, 1])
```

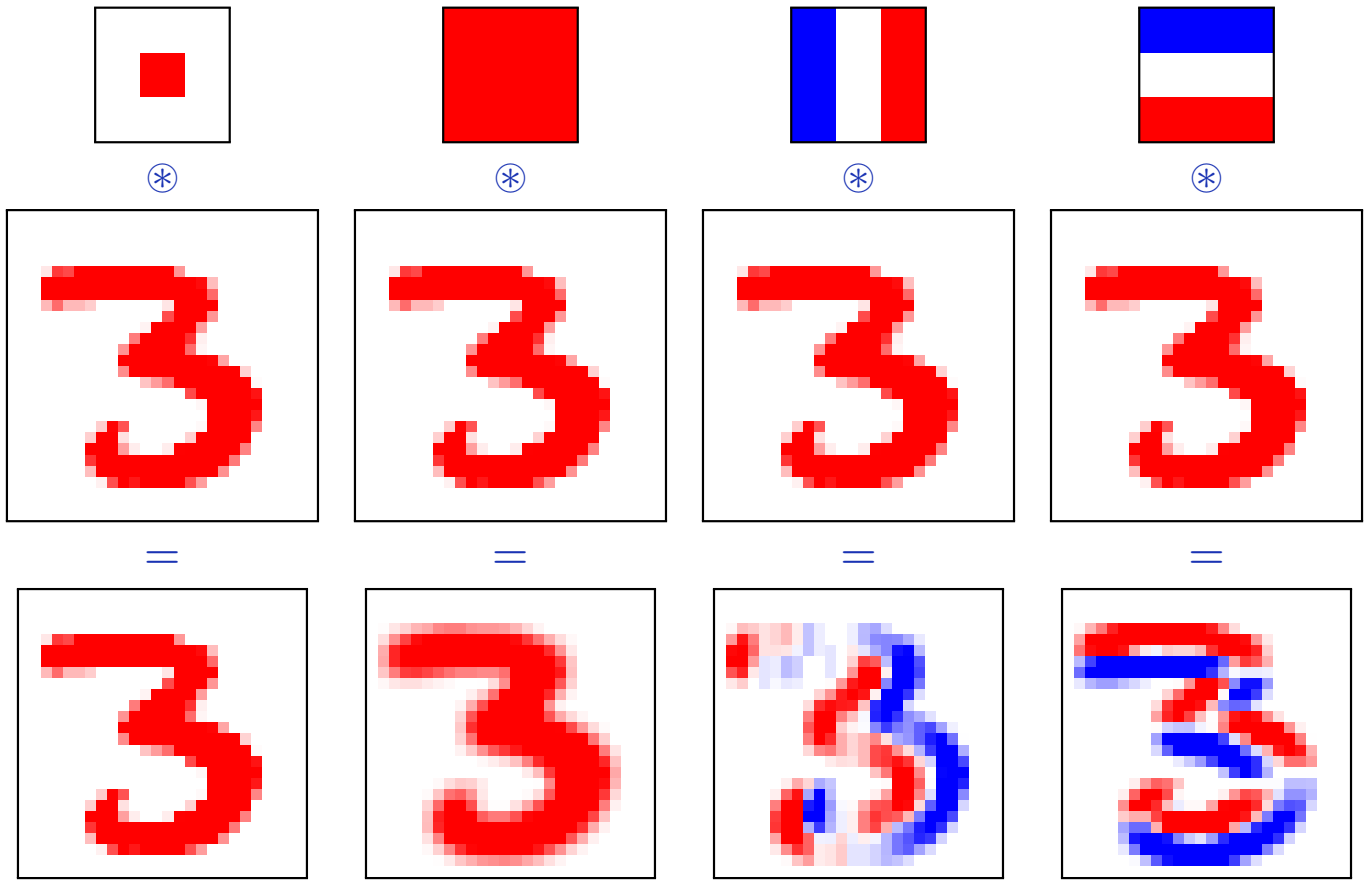
Similar functions implement 1d and 3d convolutions.

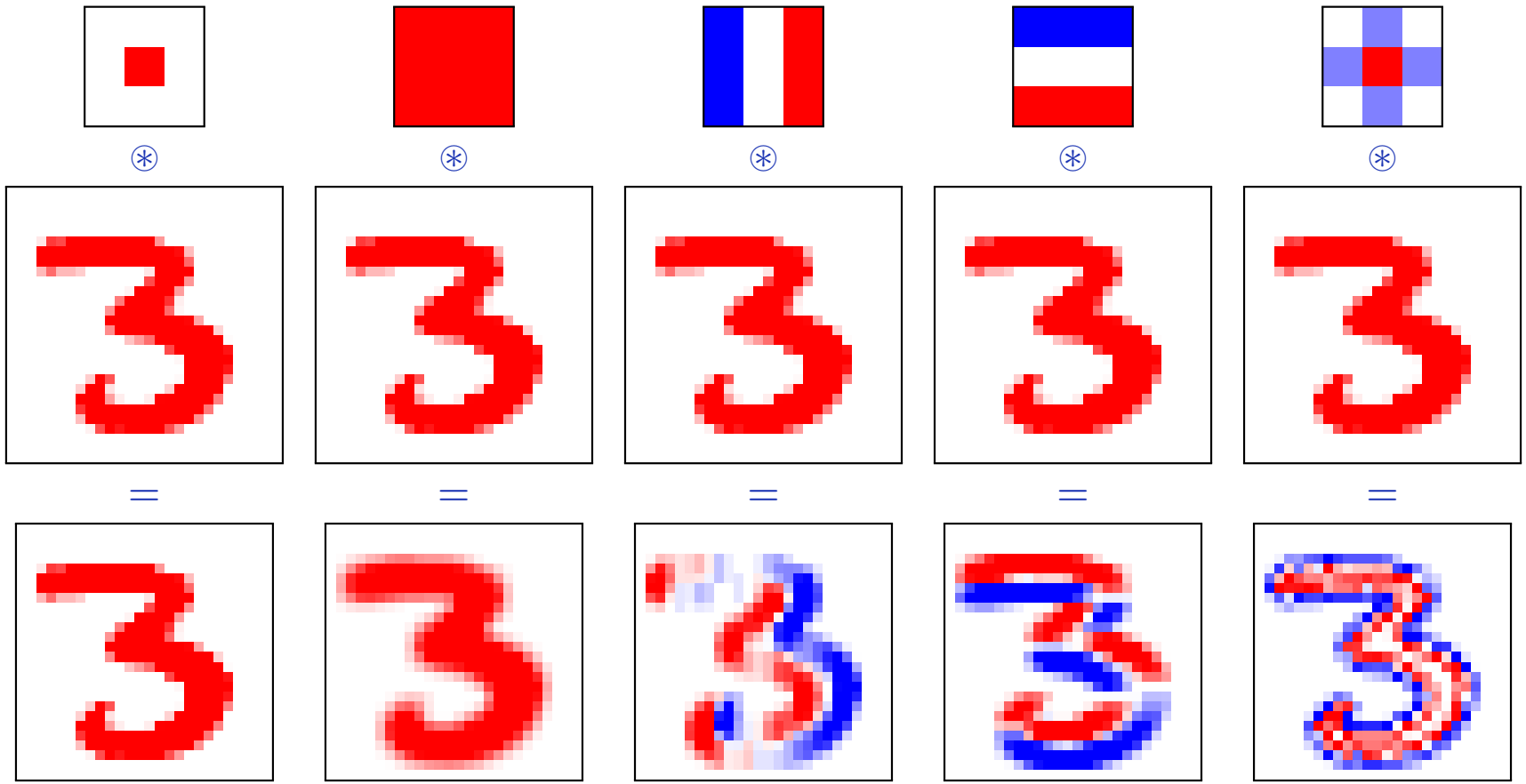
```
x = mnist_train.data[12].float().view(1, 1, 28, 28)

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)
```











```
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).

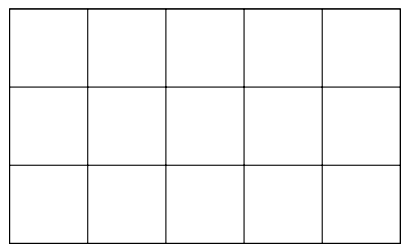
```
>>> f = nn.Conv2d(in_channels = 4, out_channels = 5, kernel_size = (2, 3))
>>> for n, p in f.named_parameters(): print(n, p.size())
...
weight torch.Size([5, 4, 2, 3])
bias torch.Size([5])
>>> x = torch.randn(117, 4, 10, 3)
>>> y = f(x)
>>> y.size()
torch.Size([117, 5, 9, 1])
```

Padding, stride, and dilation

Convolutions have three additional parameters:

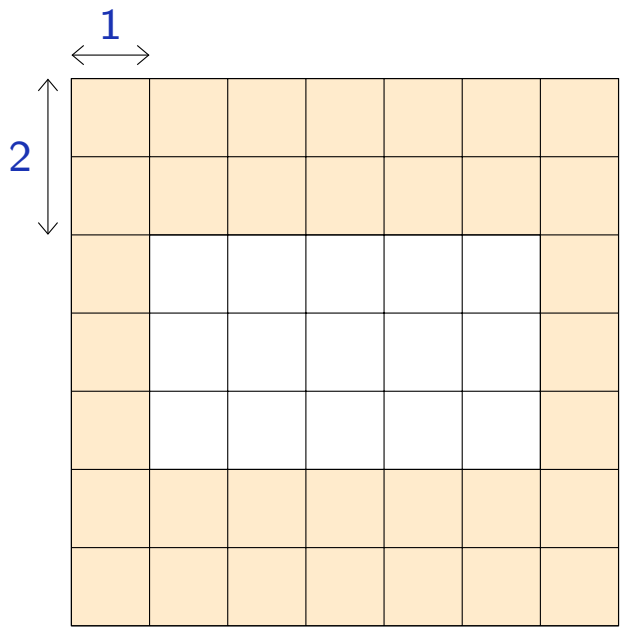
- The **padding** specifies the size of a zeroed frame added around the input,
- the **stride** specifies a step size when moving the kernel across the signal,
- the **dilation** modulates the expansion of the filter without adding weights.

Here with $C \times 3 \times 5$ as input



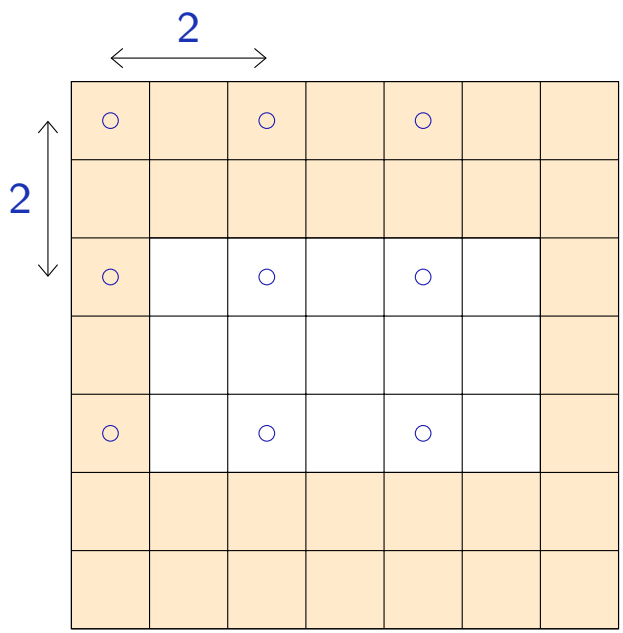
Input

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$



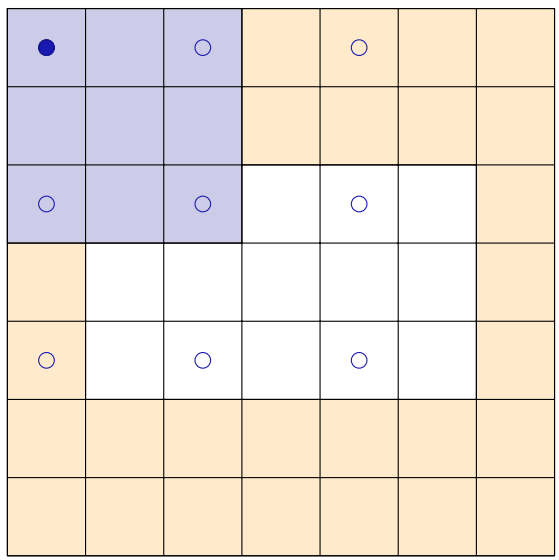
Input

Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$

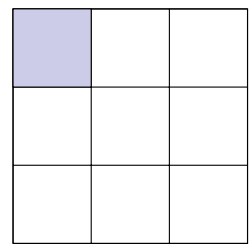


Input

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

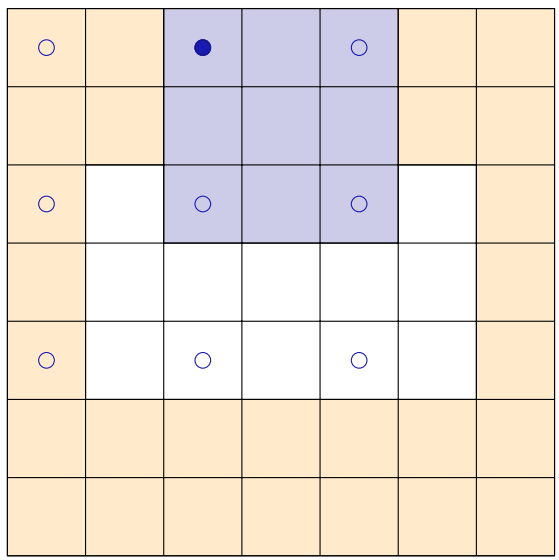


Input

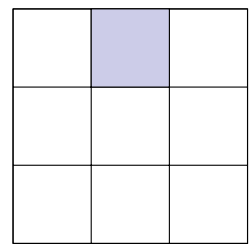


Output

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

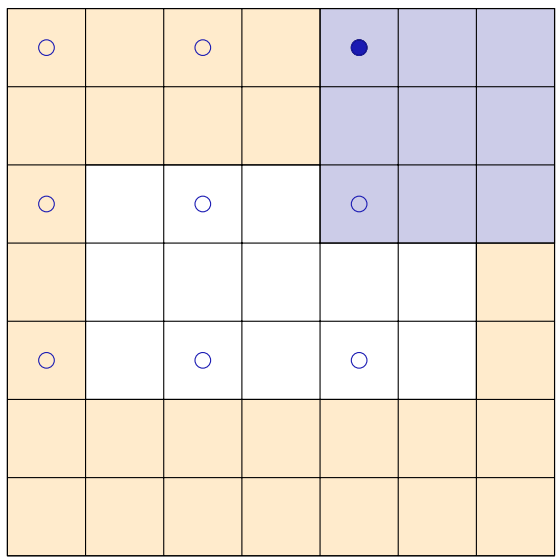


Input

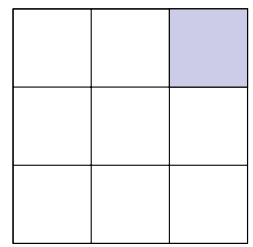


Output

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

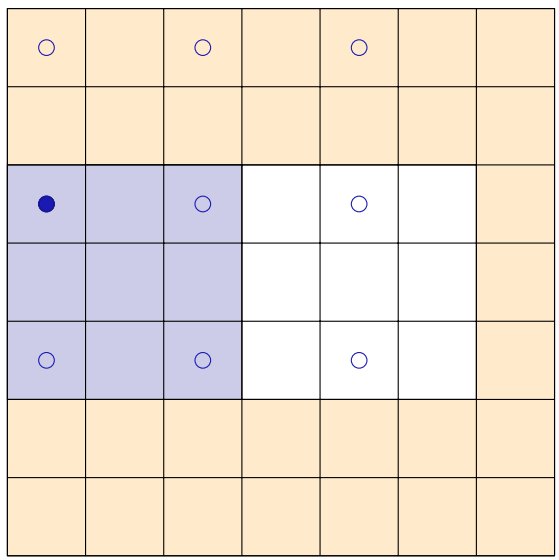


Input

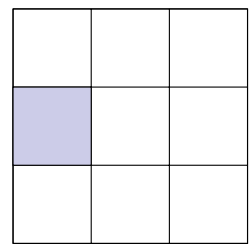


Output

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

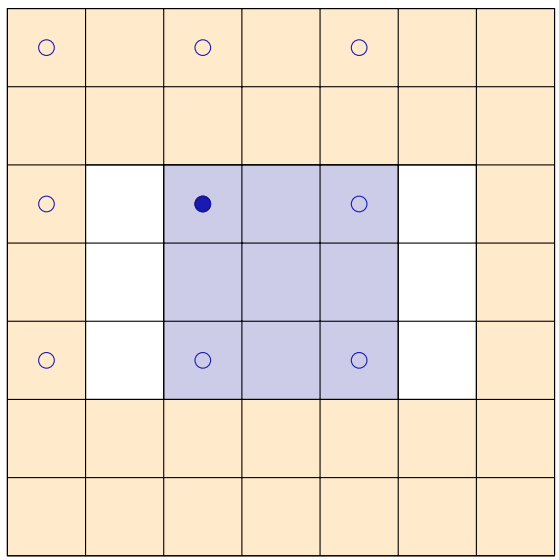


Input

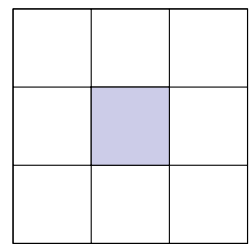


Output

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

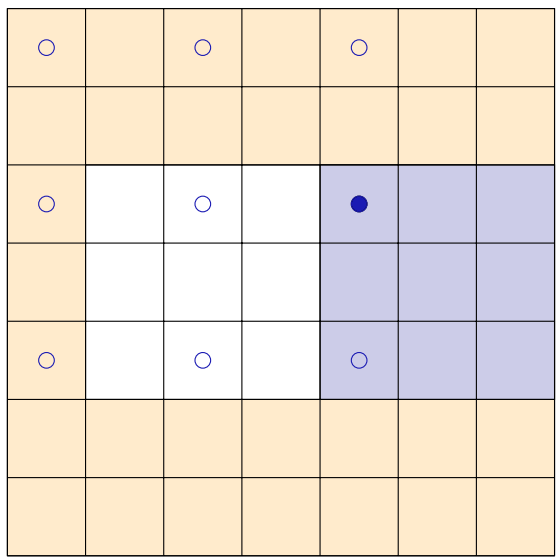


Input

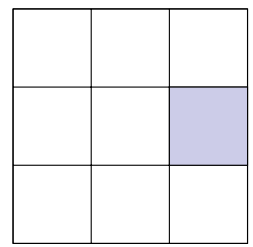


Output

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

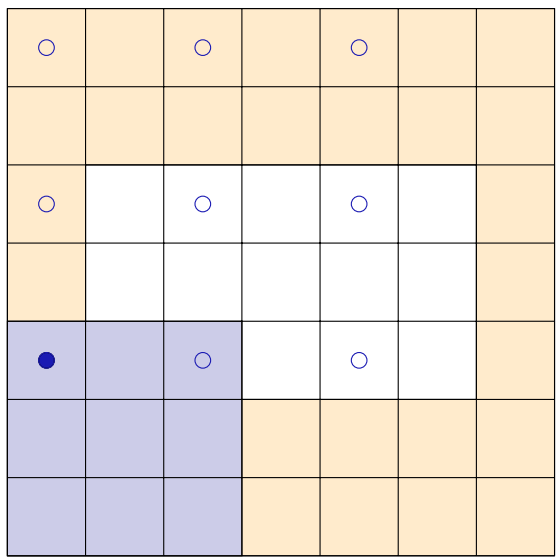


Input

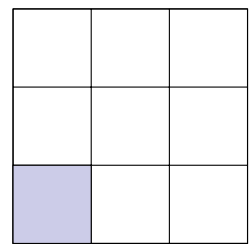


Output

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

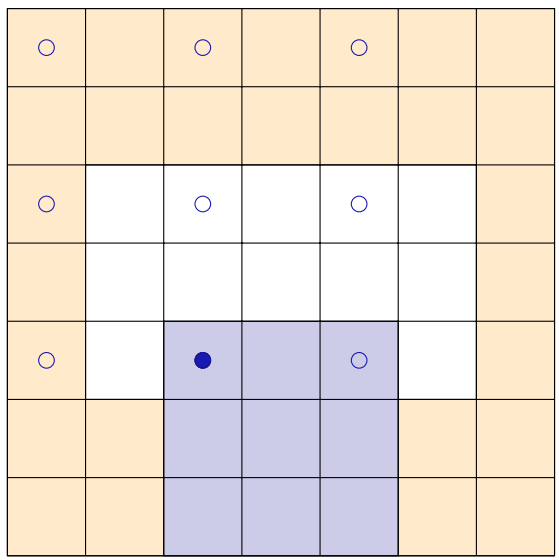


Input

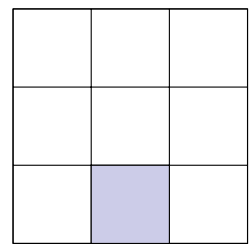


Output

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

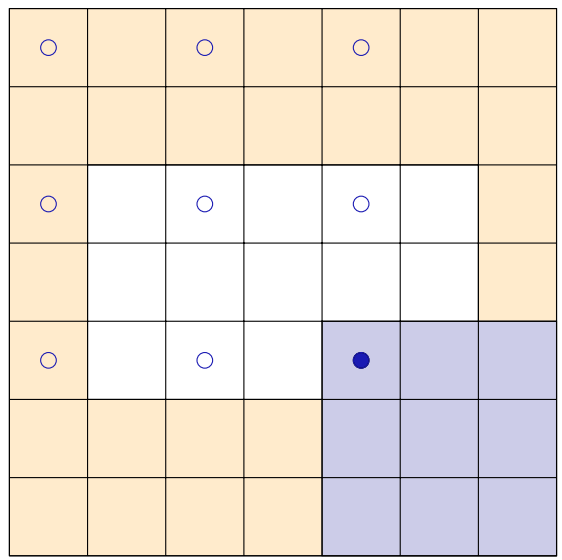


Input

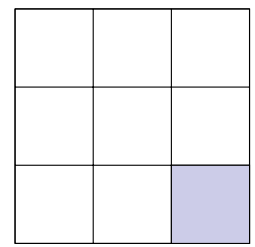


Output

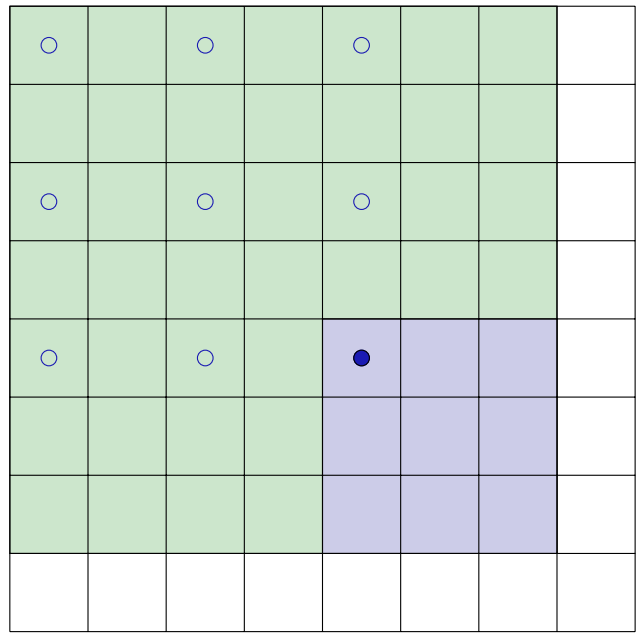
Here with $C \times 3 \times 5$ as input, a padding of $(2, 1)$, a stride of $(2, 2)$, and a kernel of size $C \times 3 \times 3$, the output is $1 \times 3 \times 3$.



Input



Output

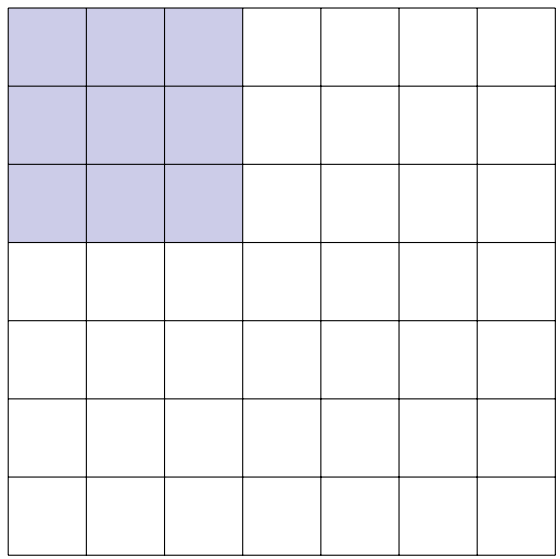


A convolution with a stride greater than 1 may not cover the input map entirely, hence may ignore some of the input values.

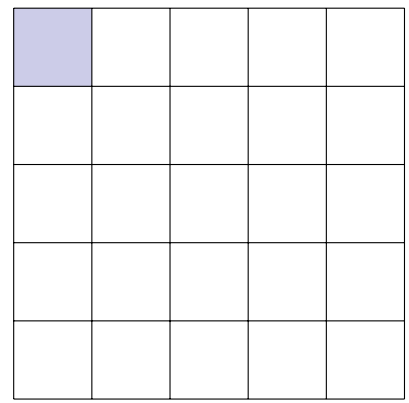
The dilation modulates the expansion of the filter support by adding rows and columns of zeros between coefficients (Yu and Koltun, 2015).

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.

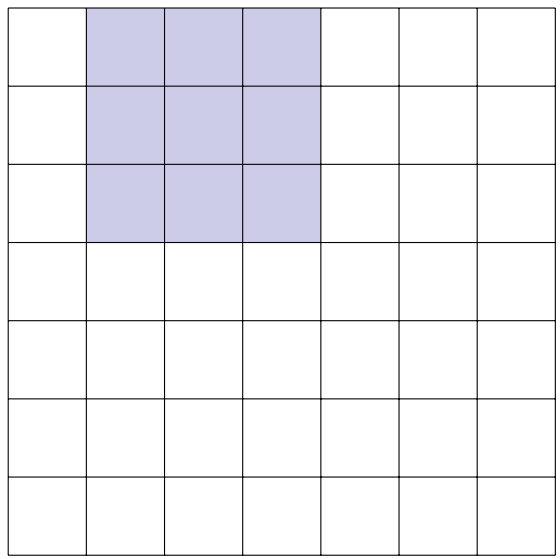
This notion comes from signal processing, where it is referred to as *algorithme à trous*, hence the term sometime used of "convolution à trous".



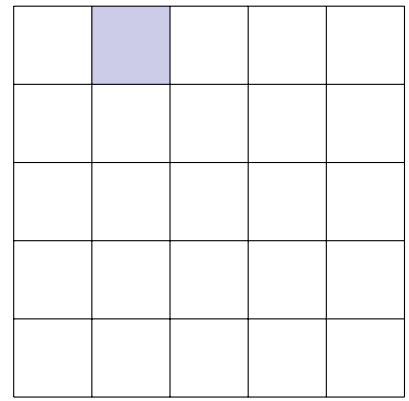
Input



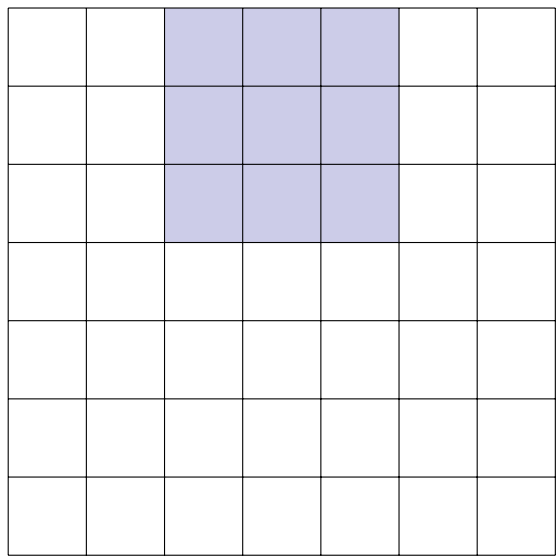
Output



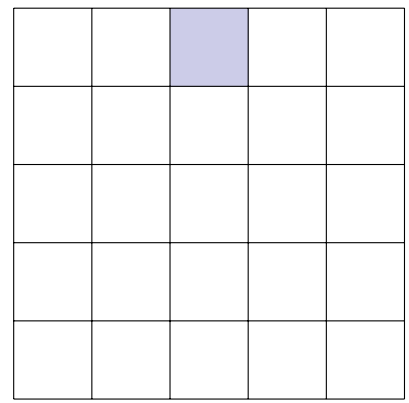
Input



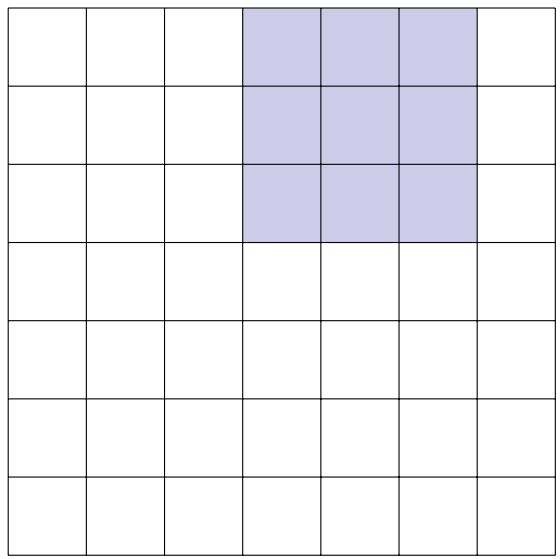
Output



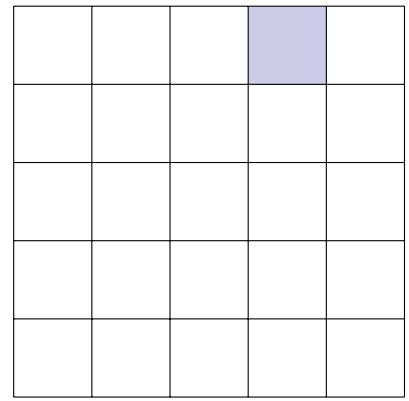
Input



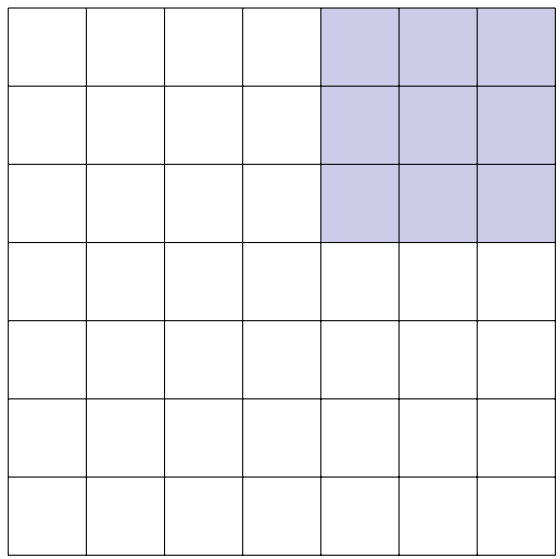
Output



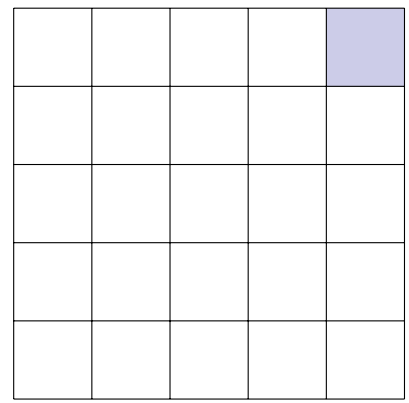
Input



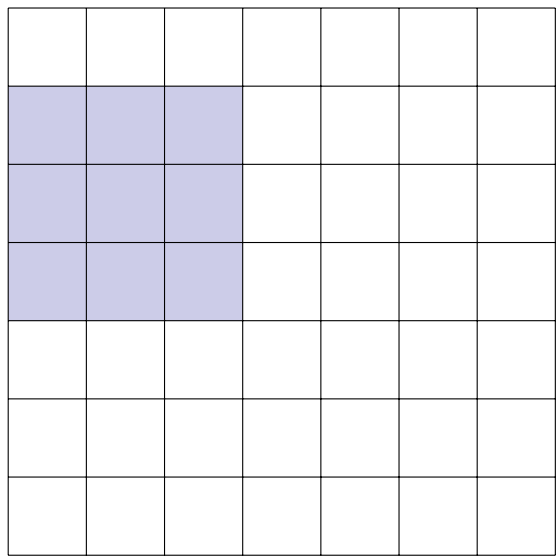
Output



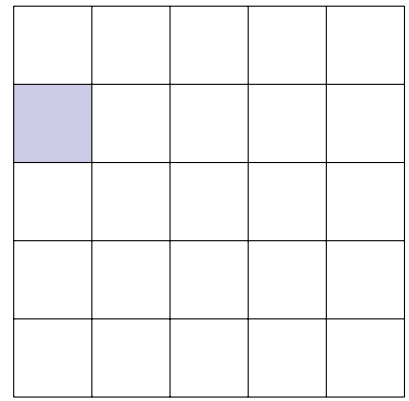
Input



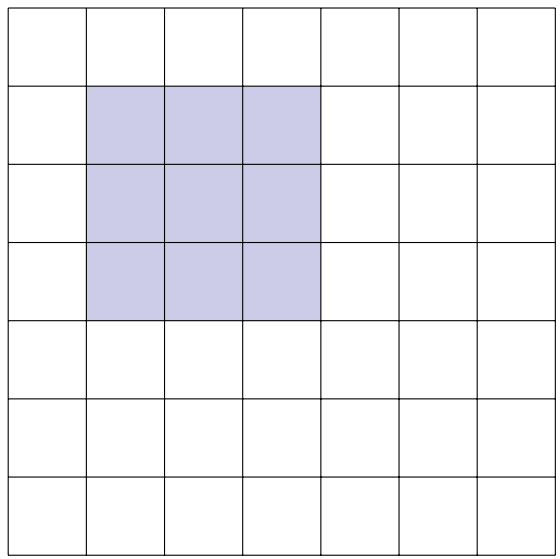
Output



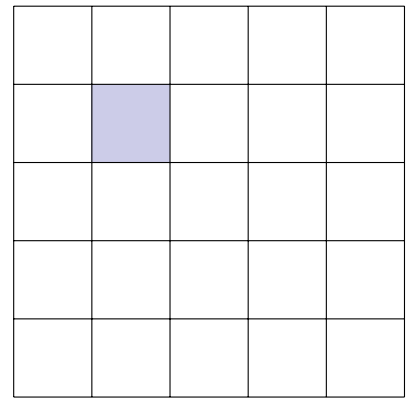
Input



Output

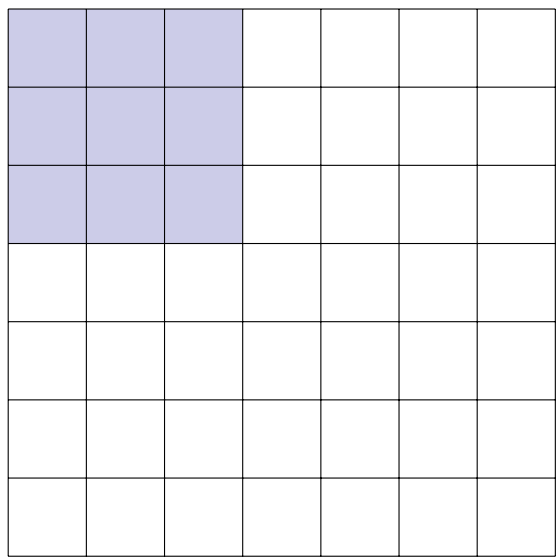


Input

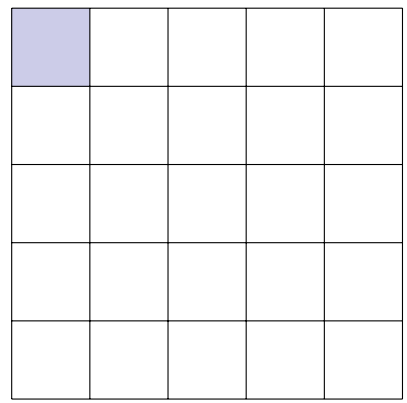


Output

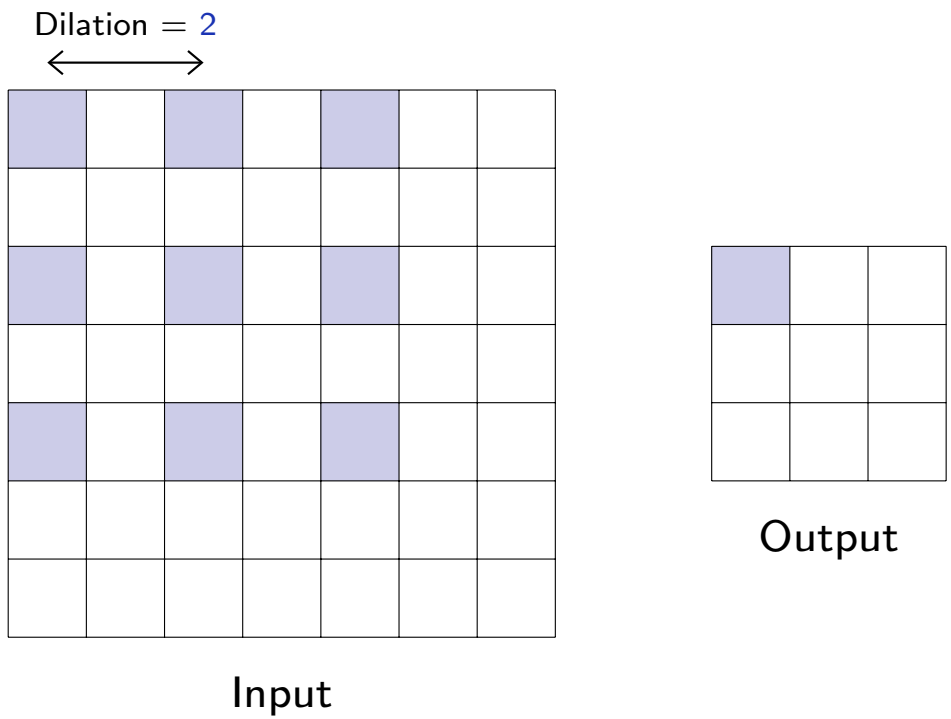
Dilation $= 1$

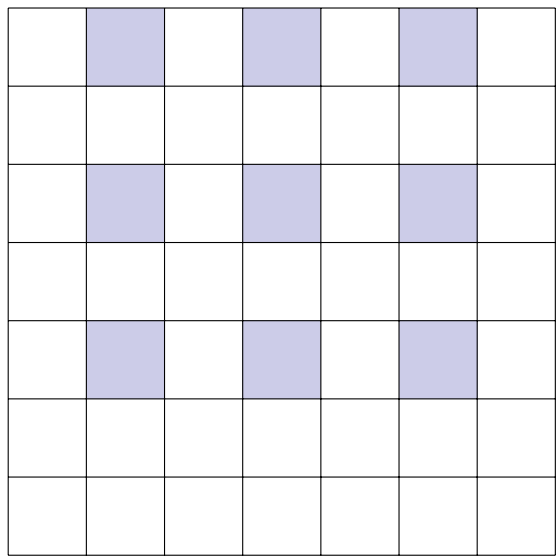


Input

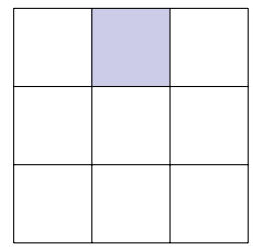


Output

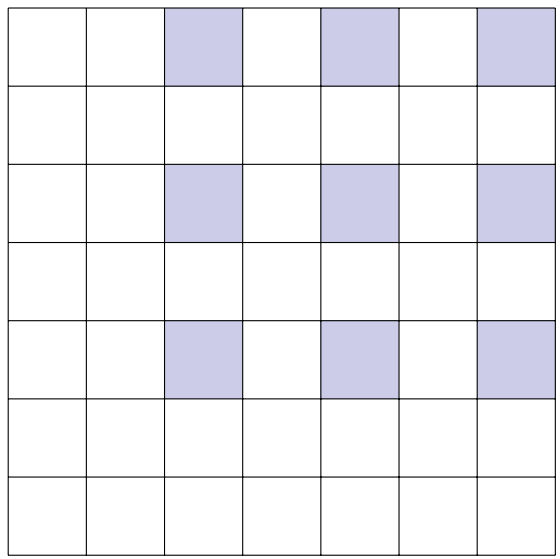

Dilation = 2
Input
Output



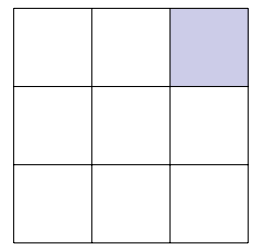
Input



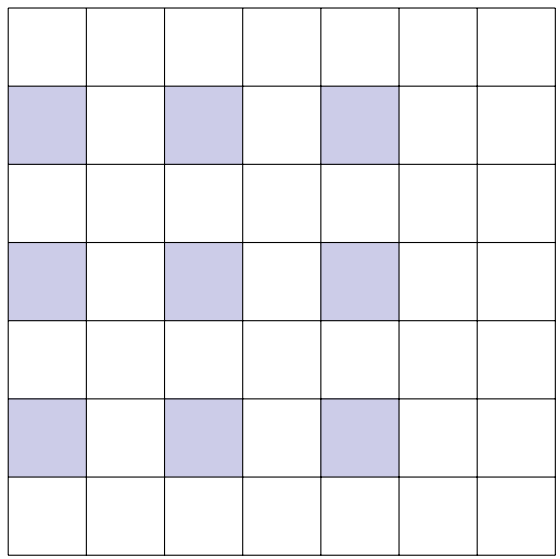
Output



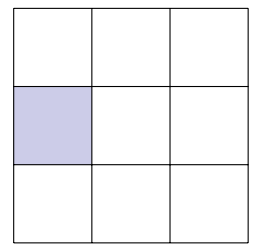
Input



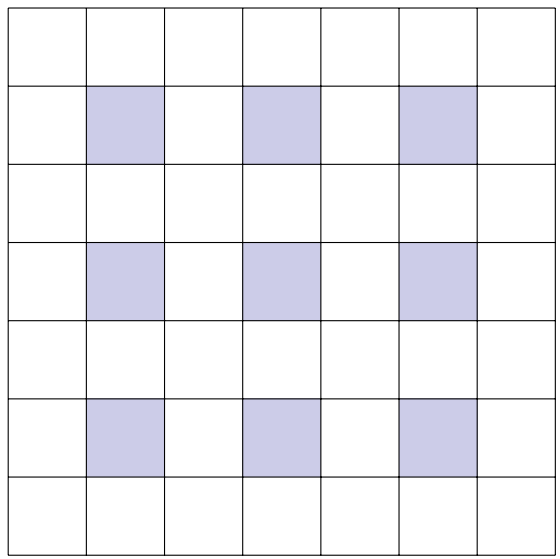
Output



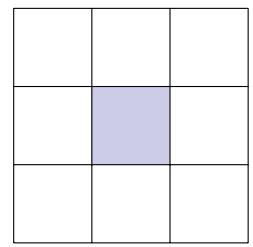
Input



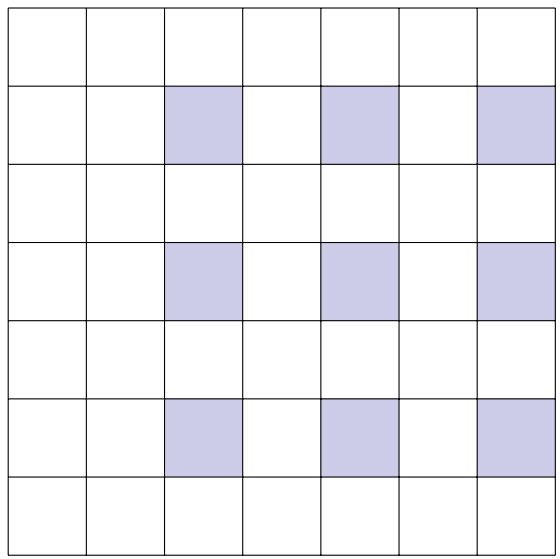
Output



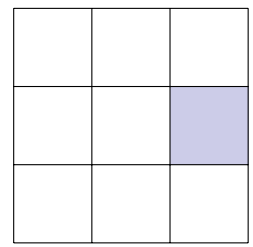
Input



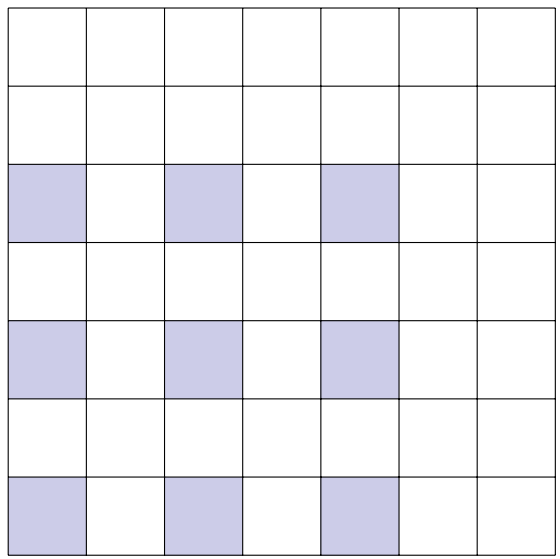
Output



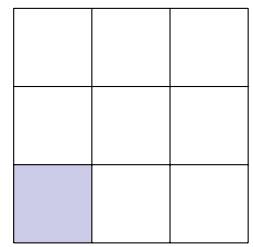
Input



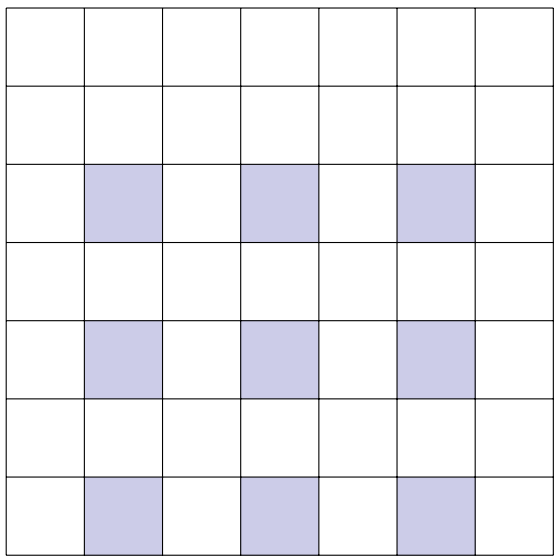
Output



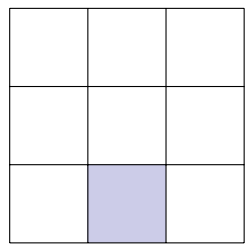
Input



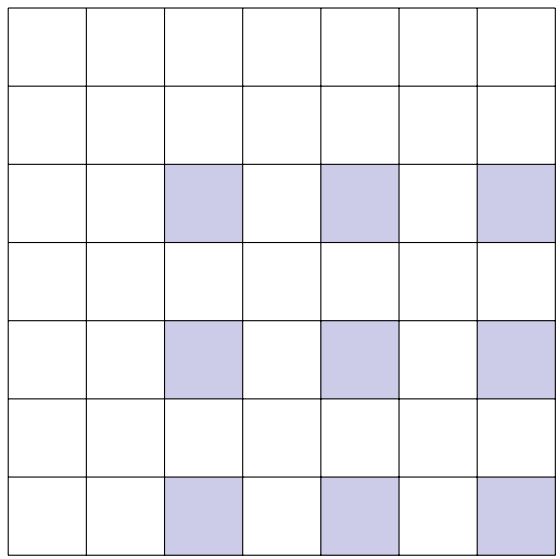
Output



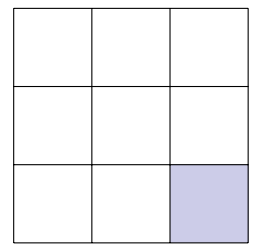
Input



Output



Input



Output

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.

Deep learning

4.5. Pooling

François Fleuret

https://fleuret.org/dlc/

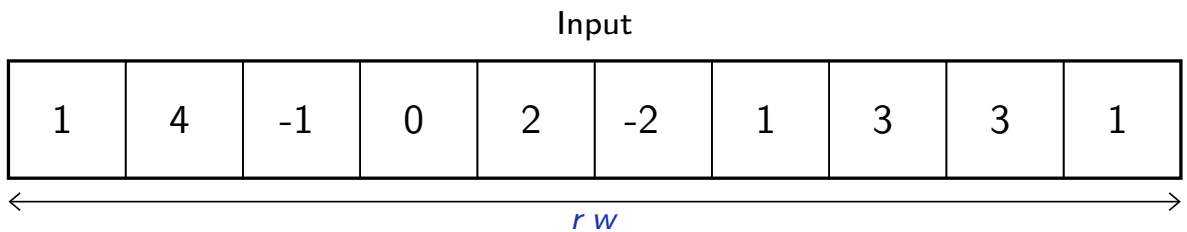
UNIVERSITÉ
DE GENÈVE

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.

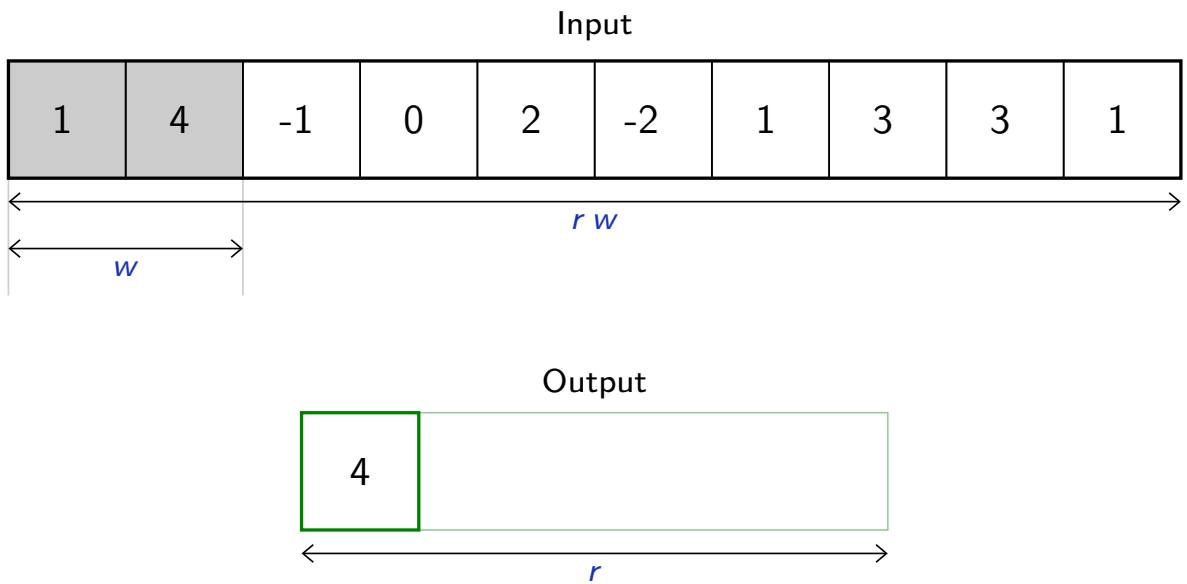
The most standard type of pooling is the **max-pooling**, which computes max values over non-overlapping blocks.

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



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

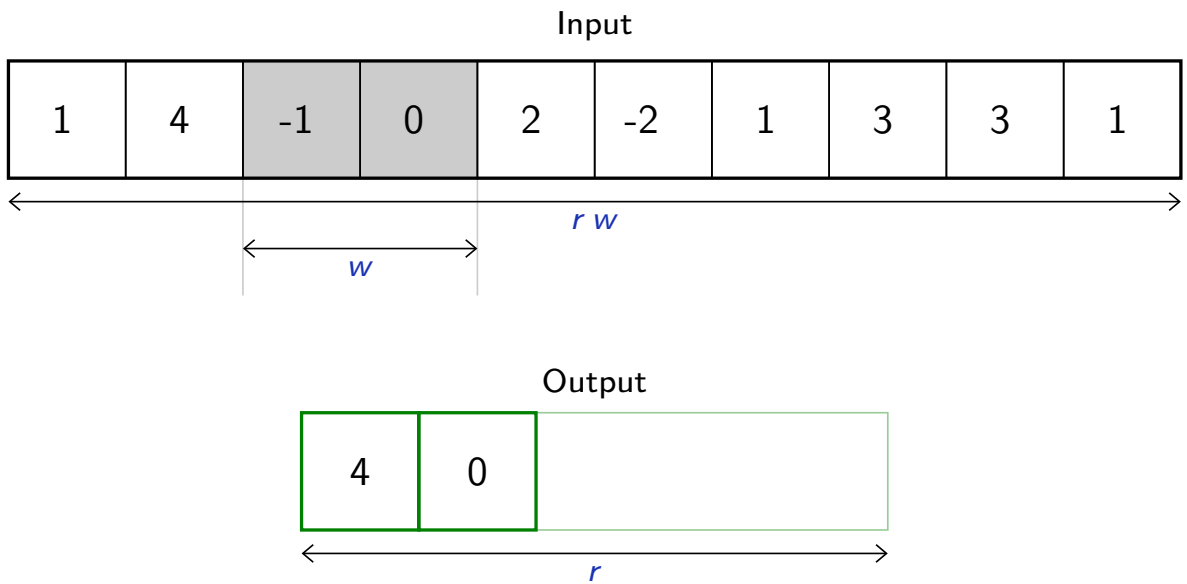
For instance in 1d with a kernel of size 2:



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

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

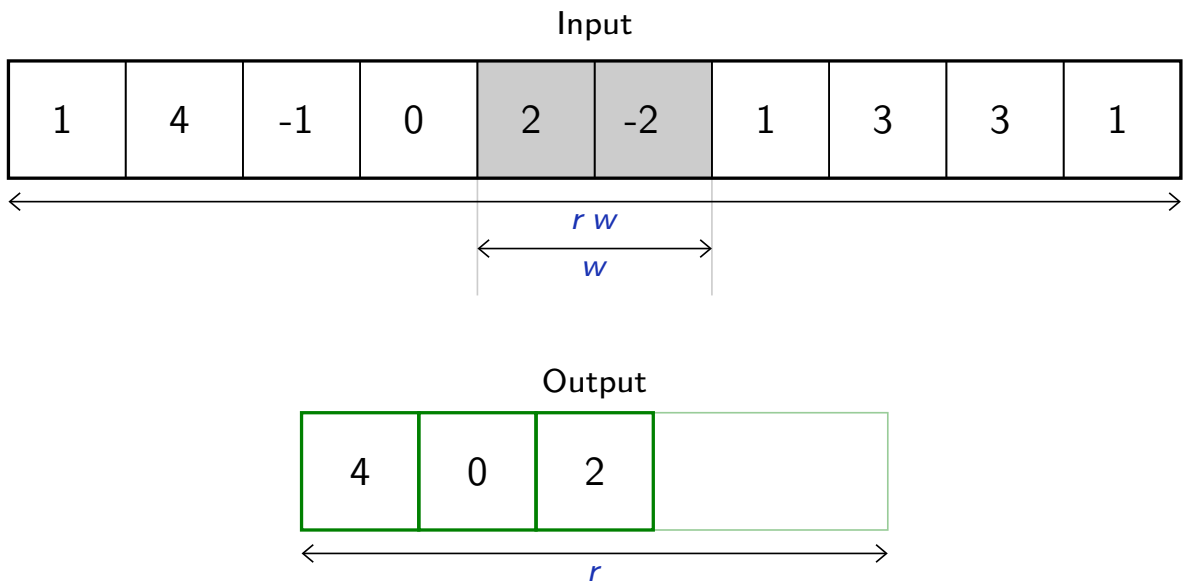
For instance in 1d with a kernel of size 2:



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

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

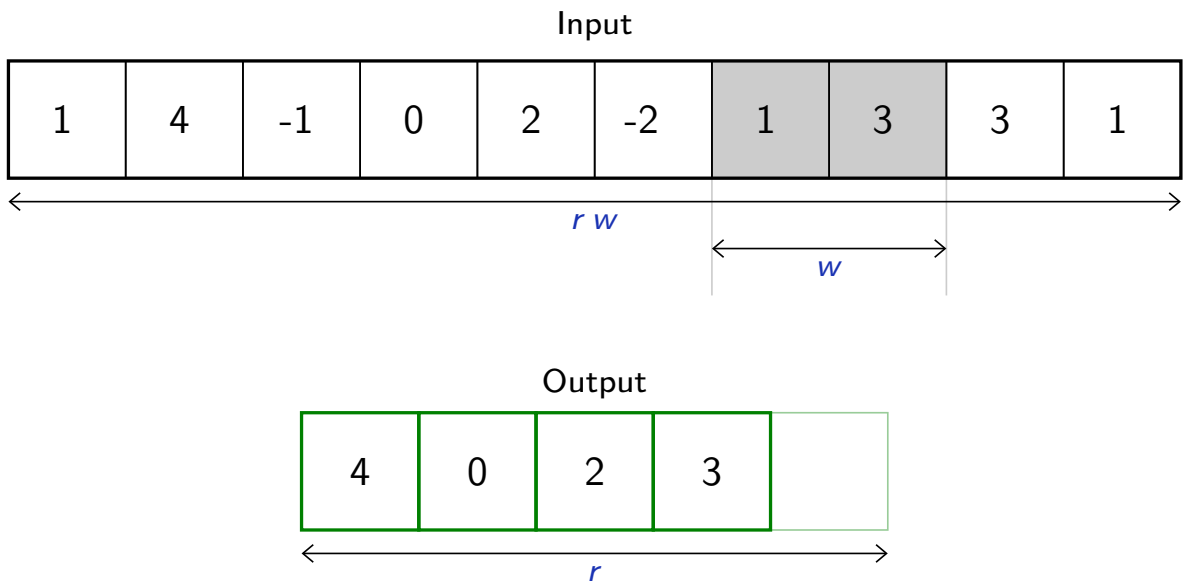
For instance in 1d with a kernel of size 2:



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

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

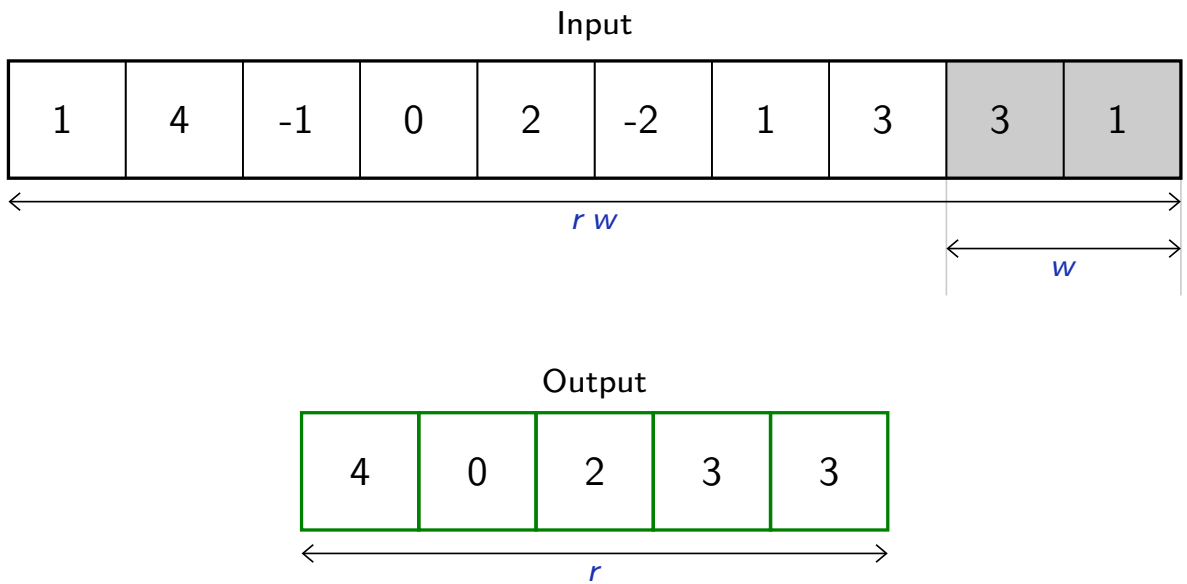
For instance in 1d with a kernel of size 2:



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

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

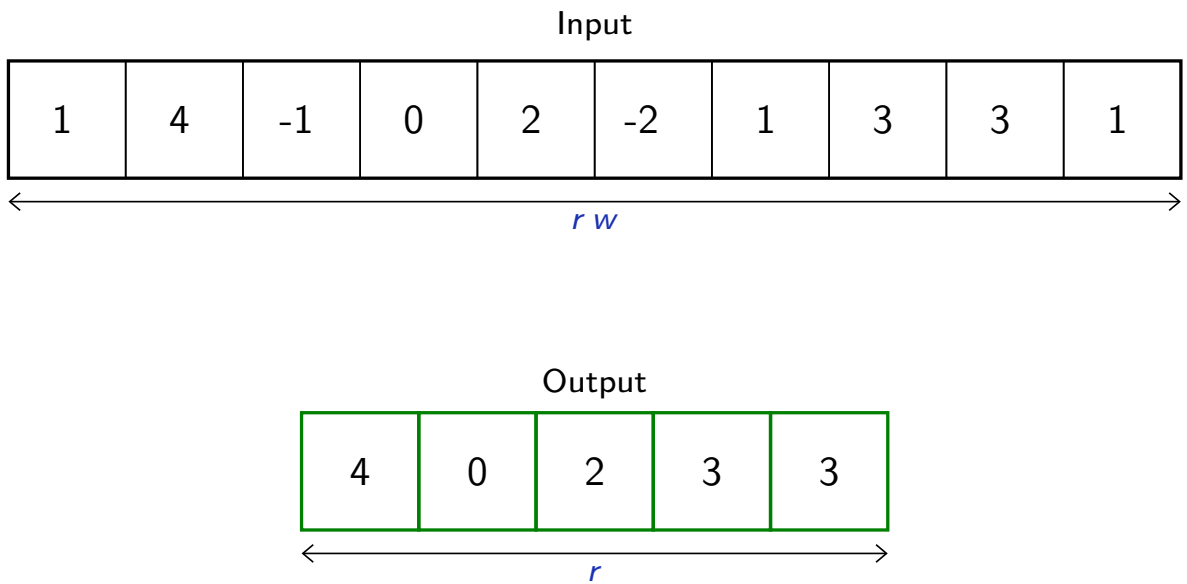
For instance in 1d with a kernel of size 2:



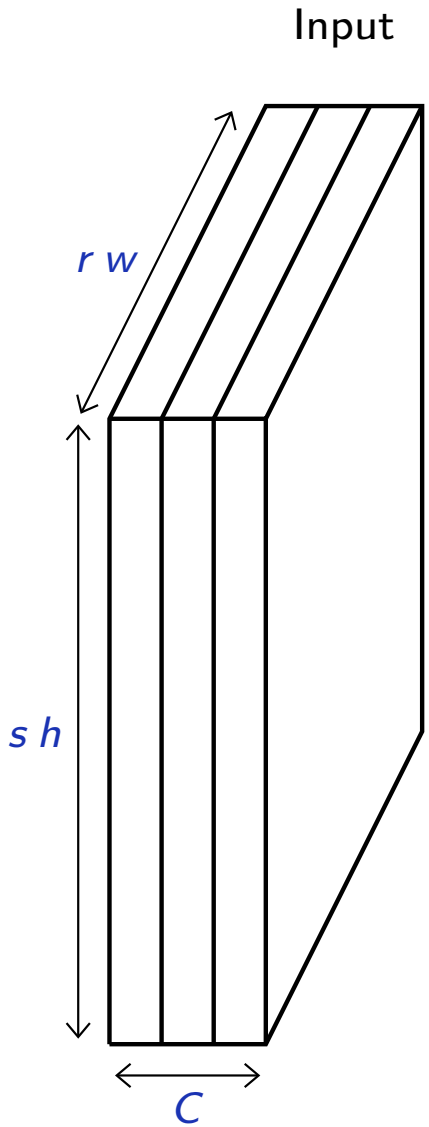
The **average pooling** computes average values per block instead of max values.

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

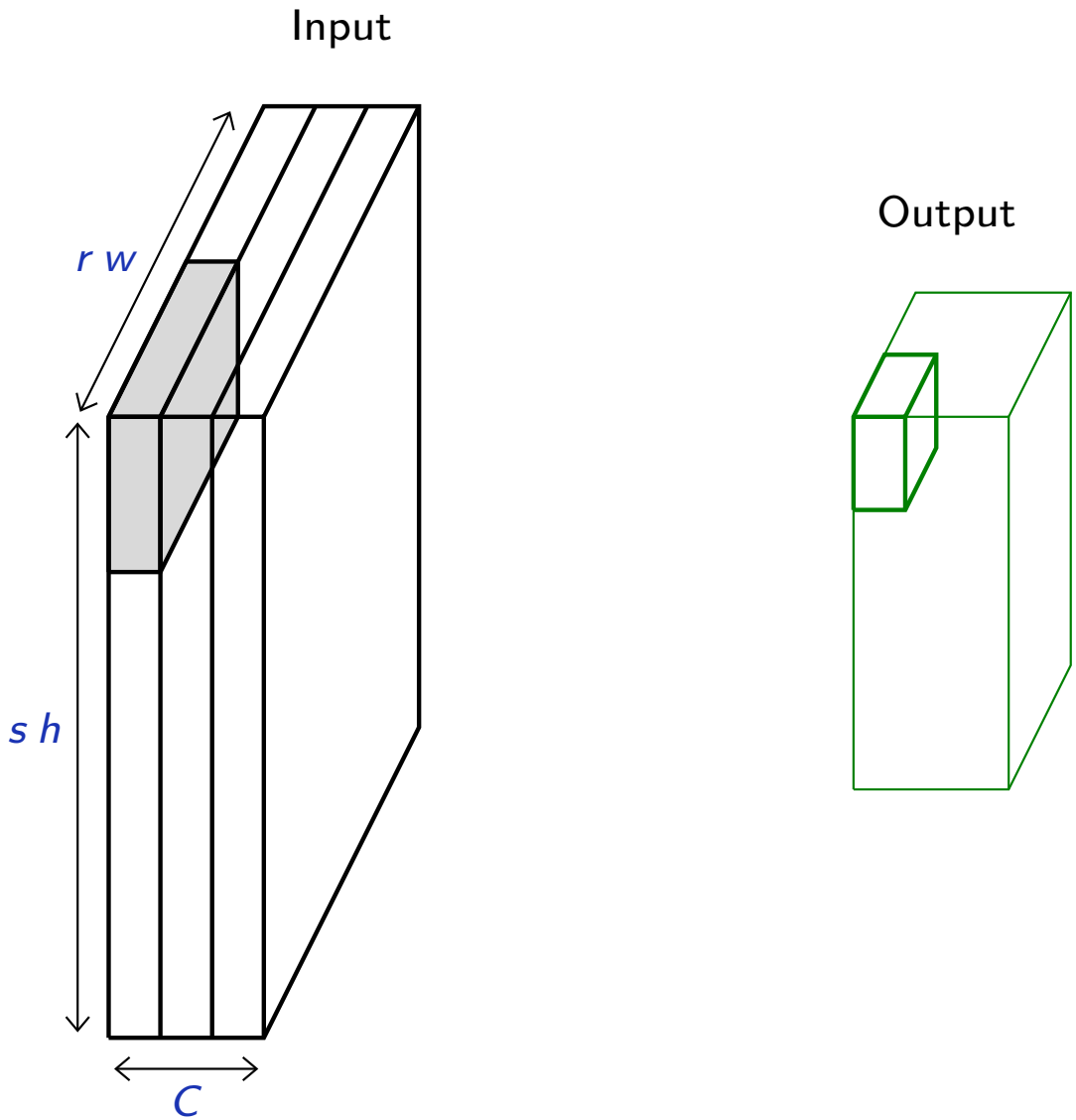
For instance in 1d with a kernel of size 2:



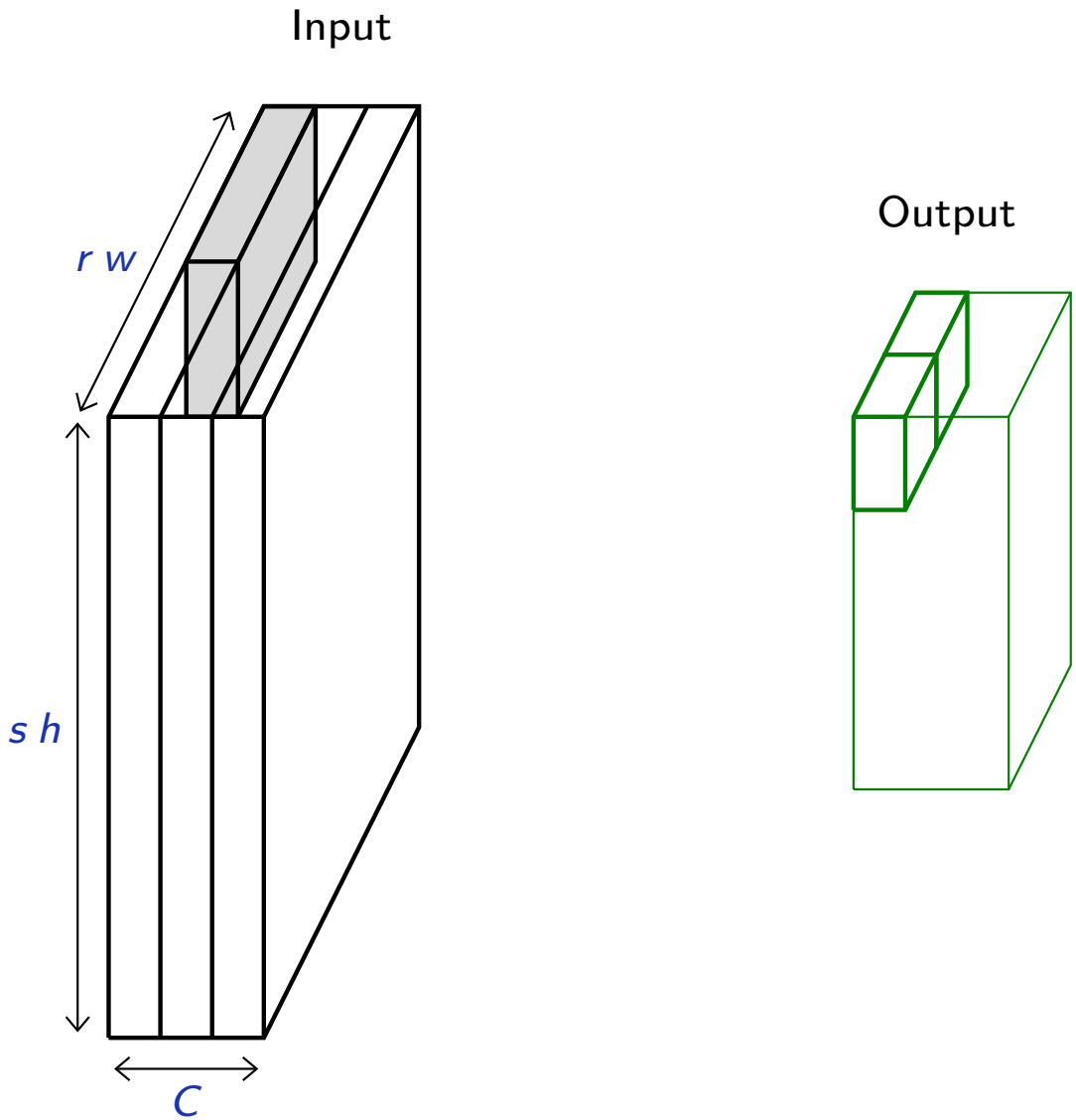
The **average pooling** computes average values per block instead of max values.



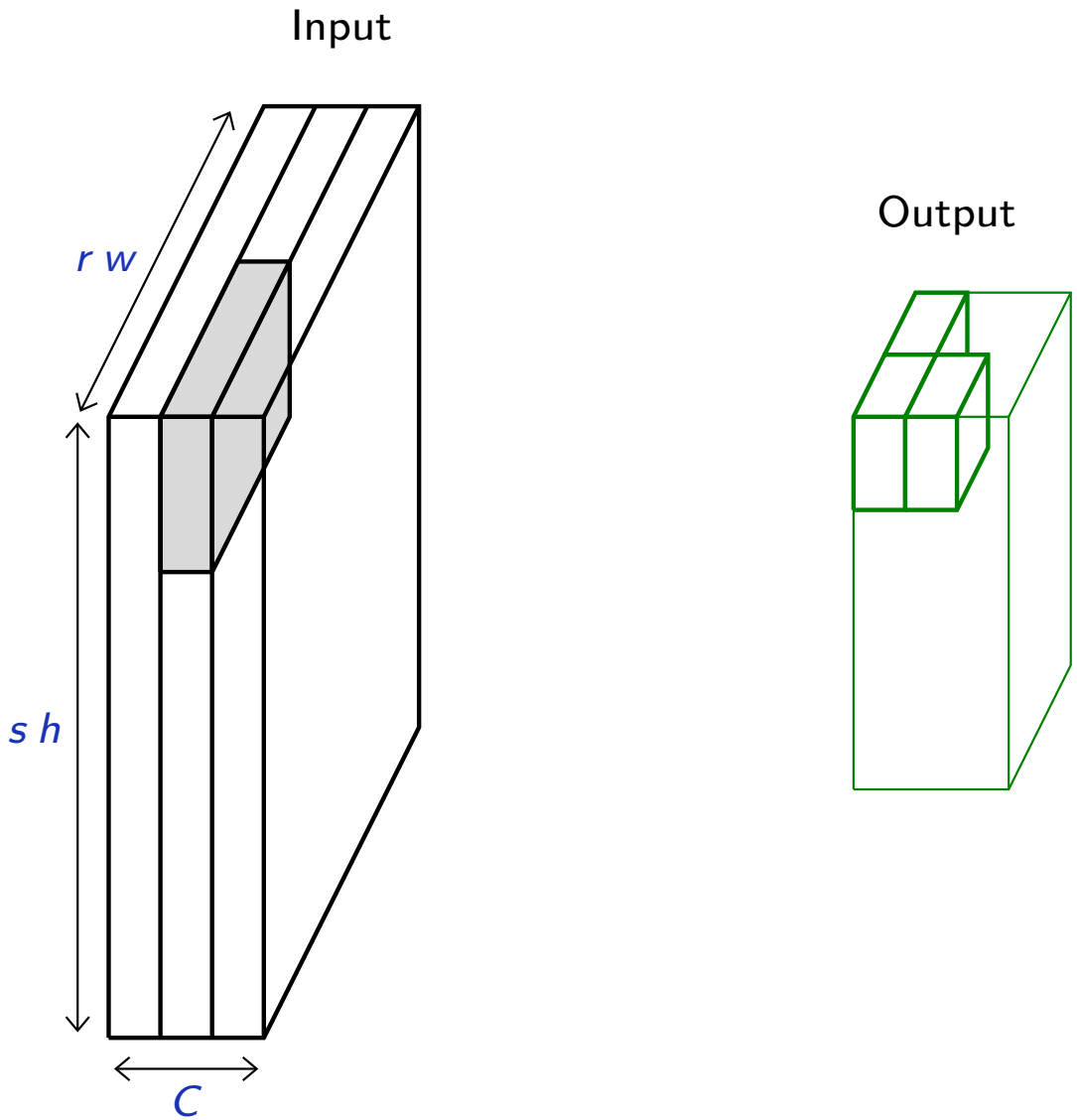
Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



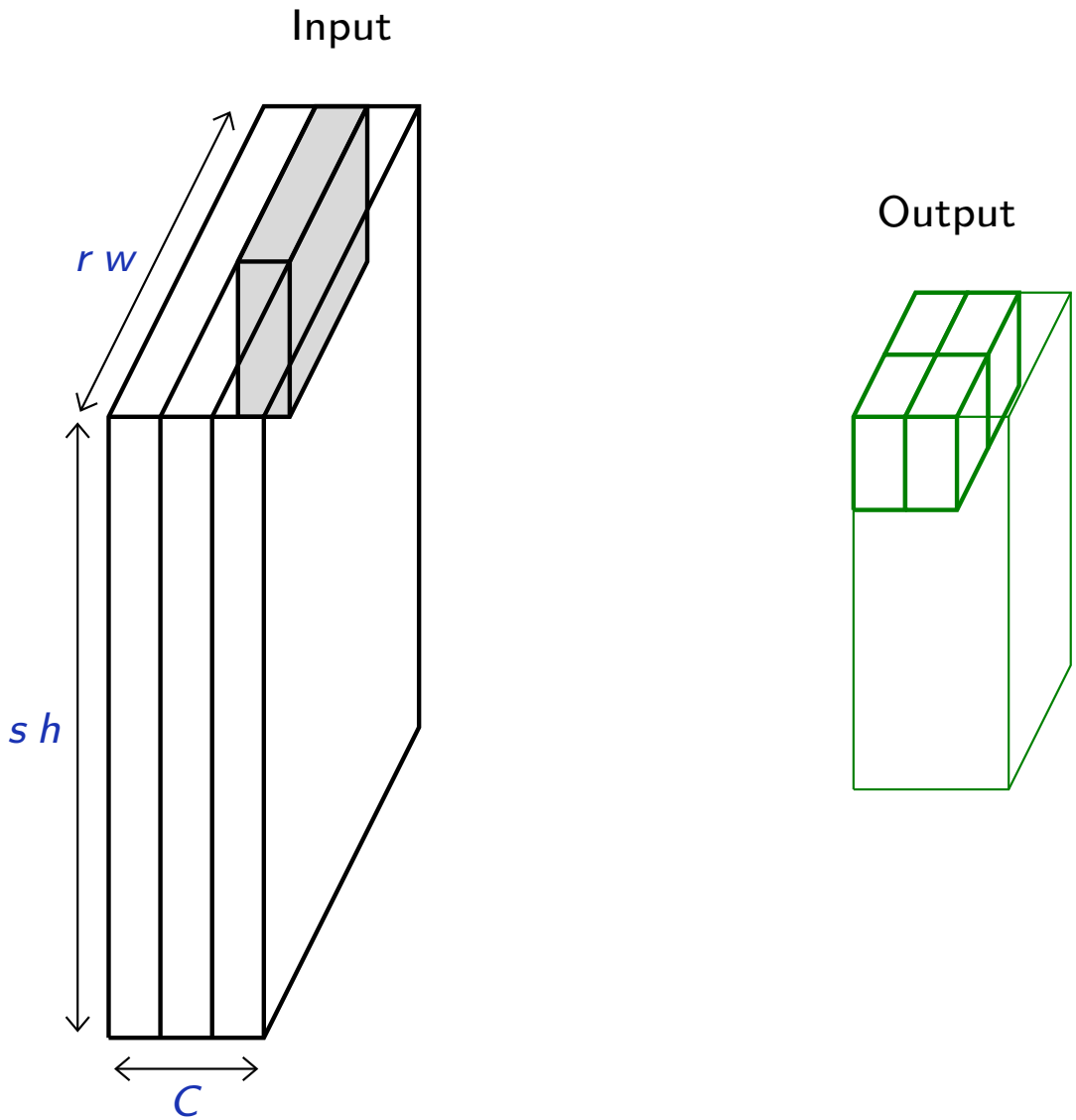
Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



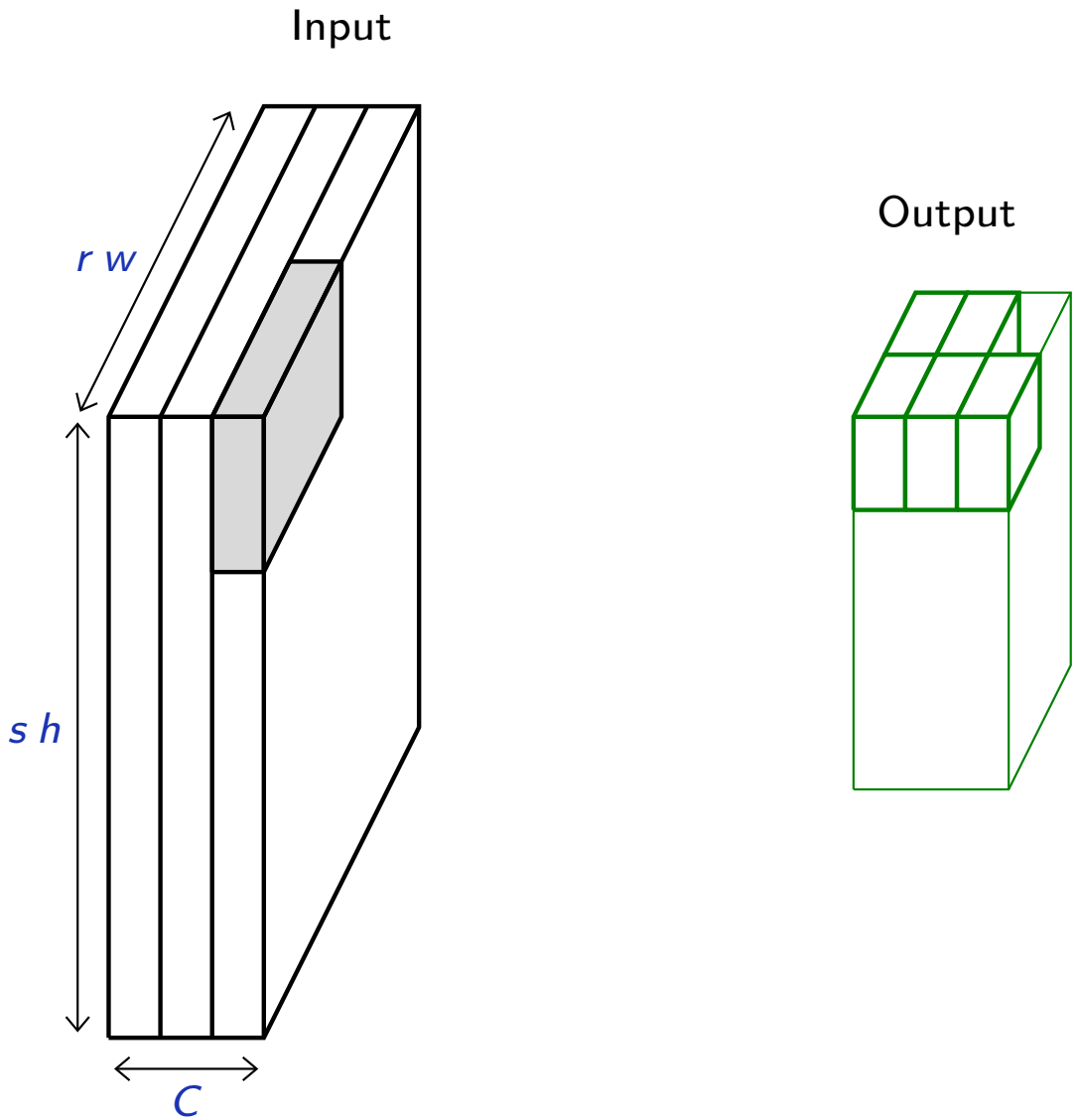
Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



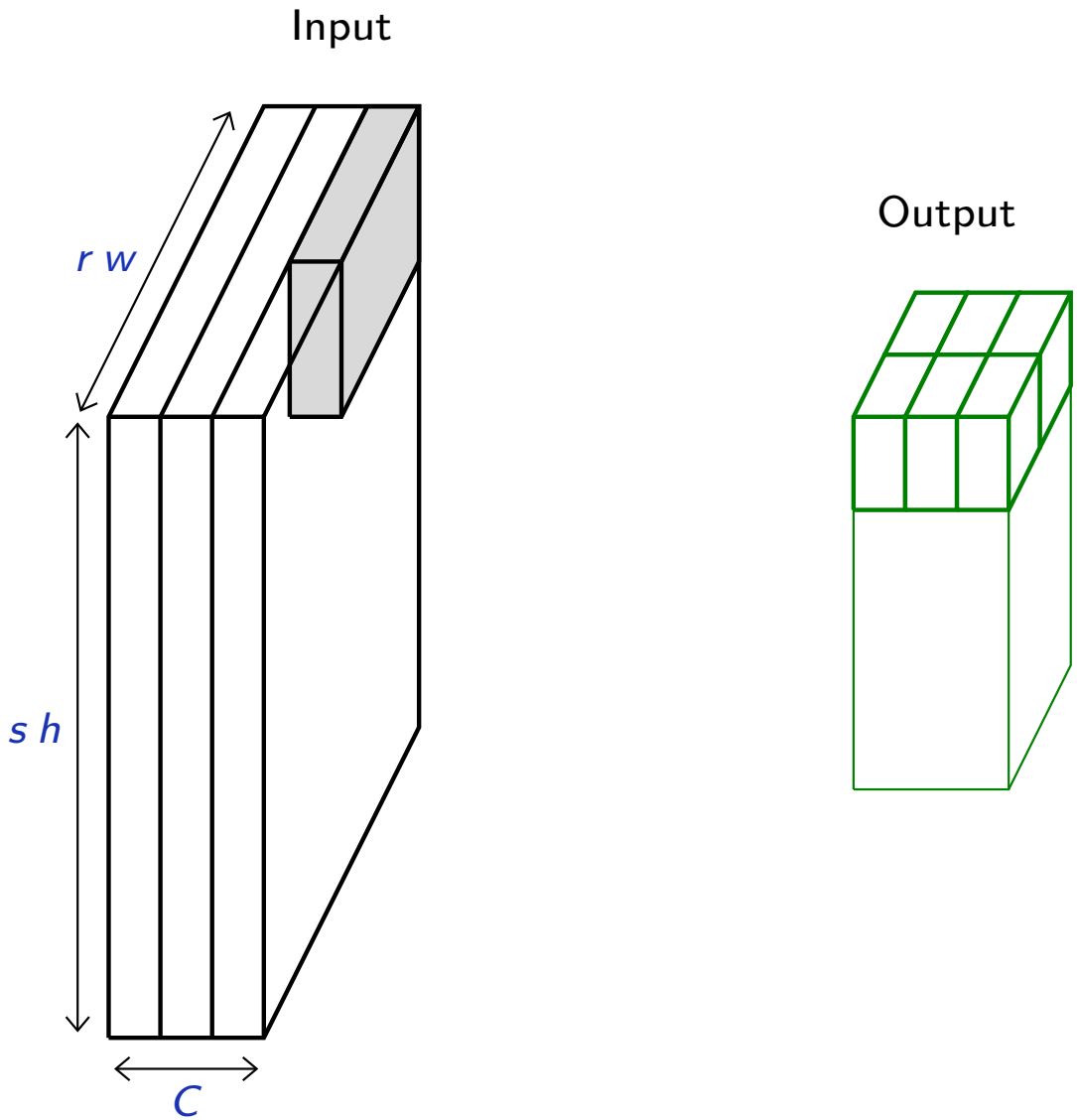
Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



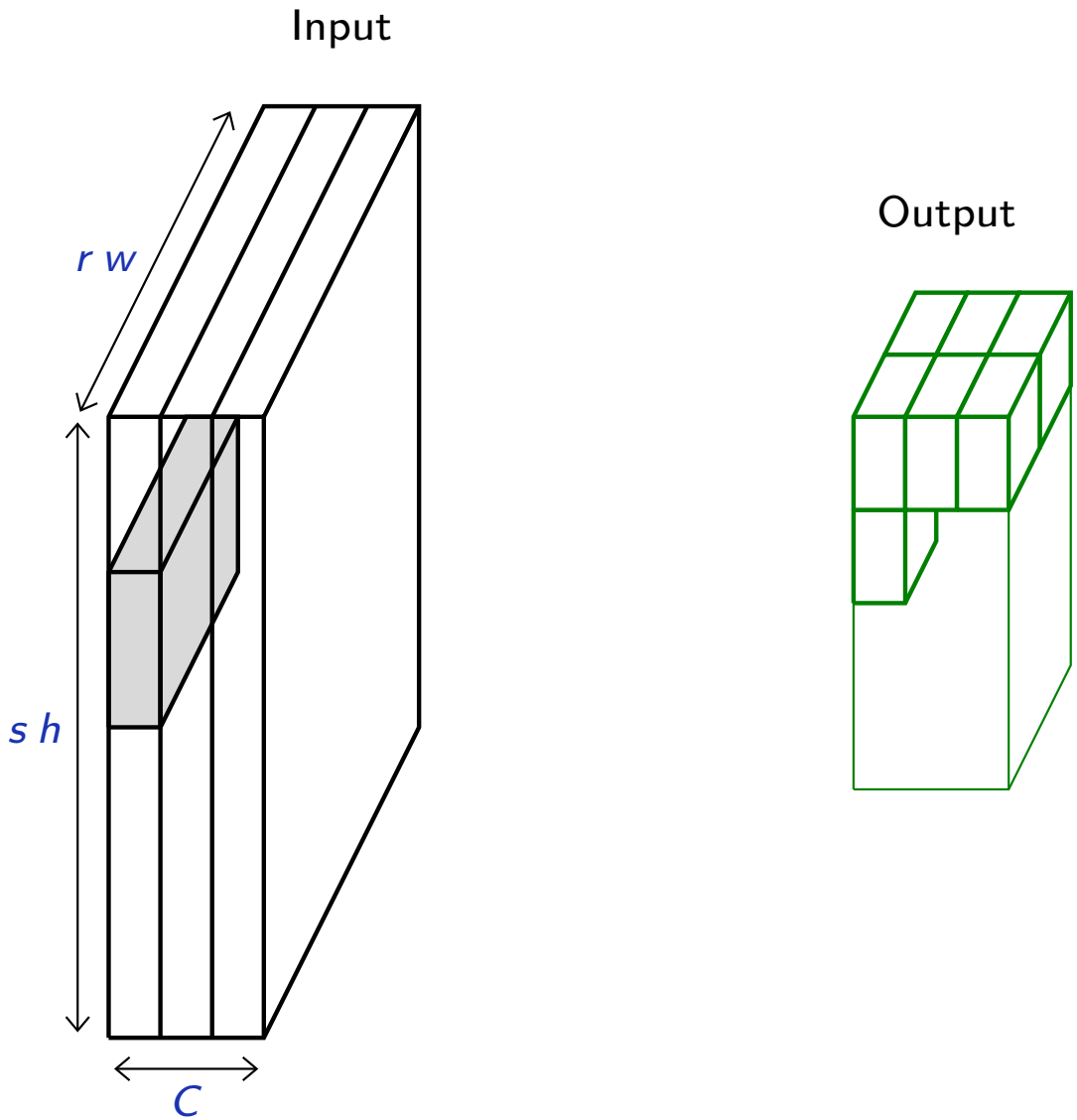
Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



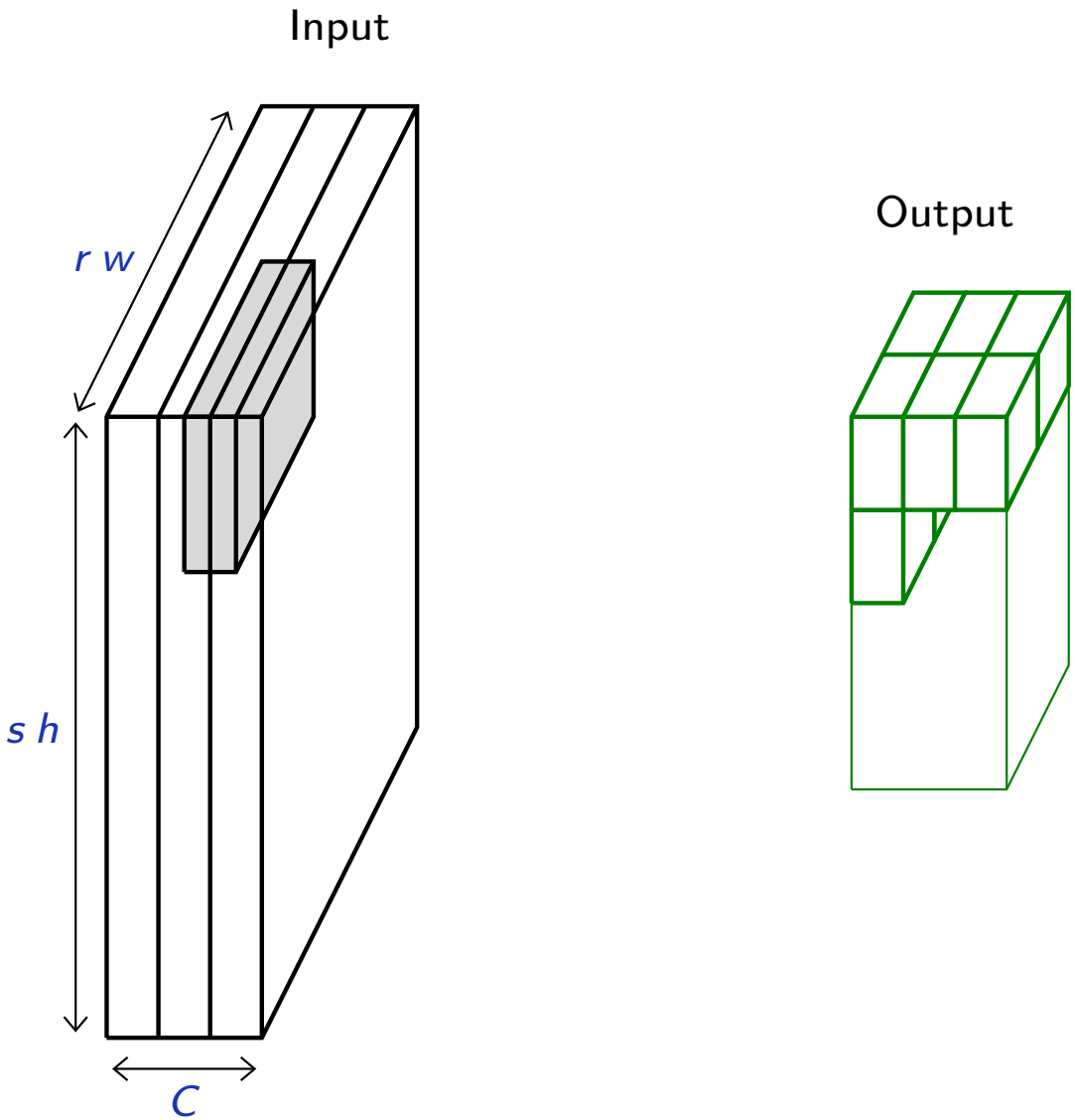
Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



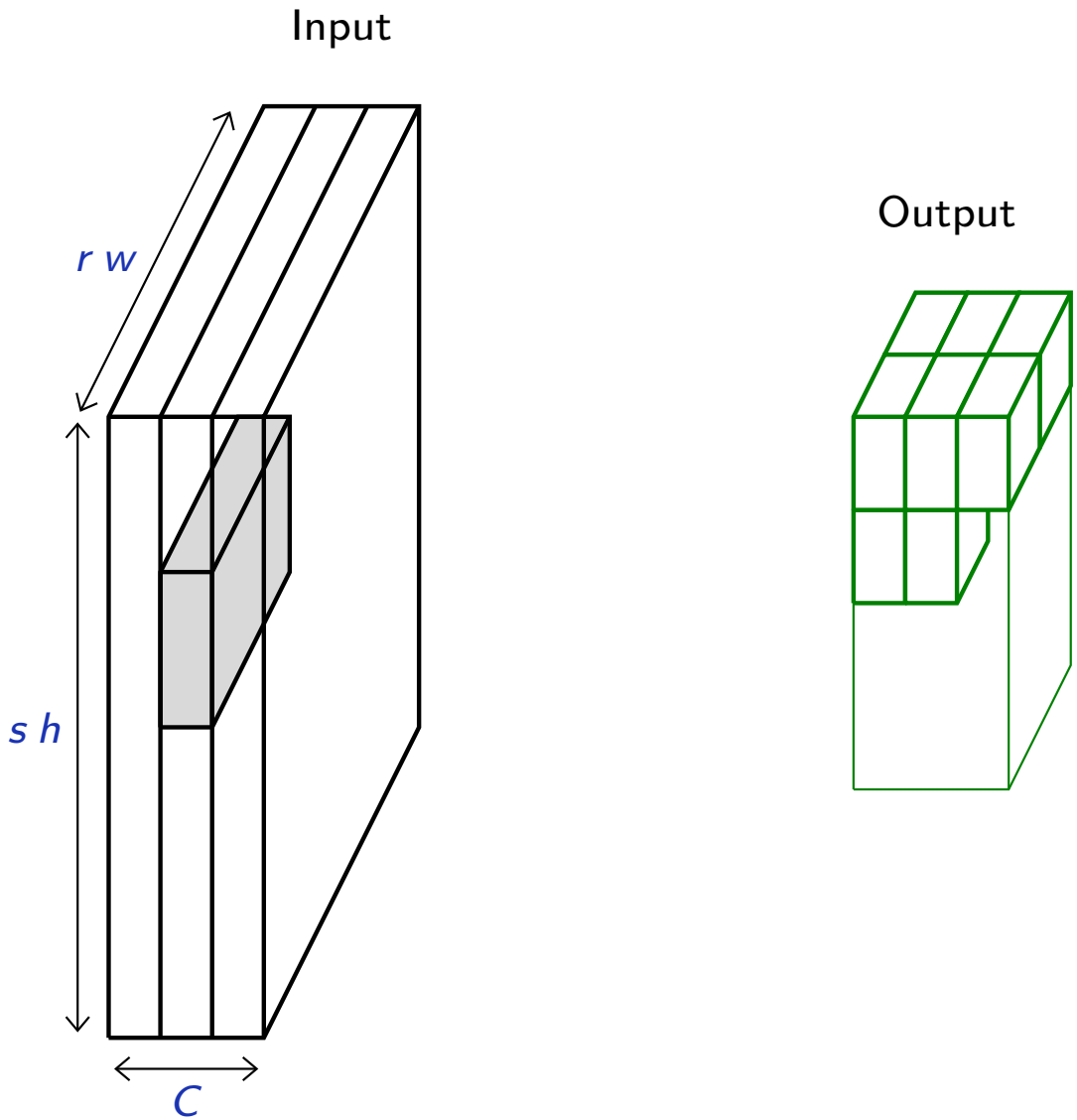
Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



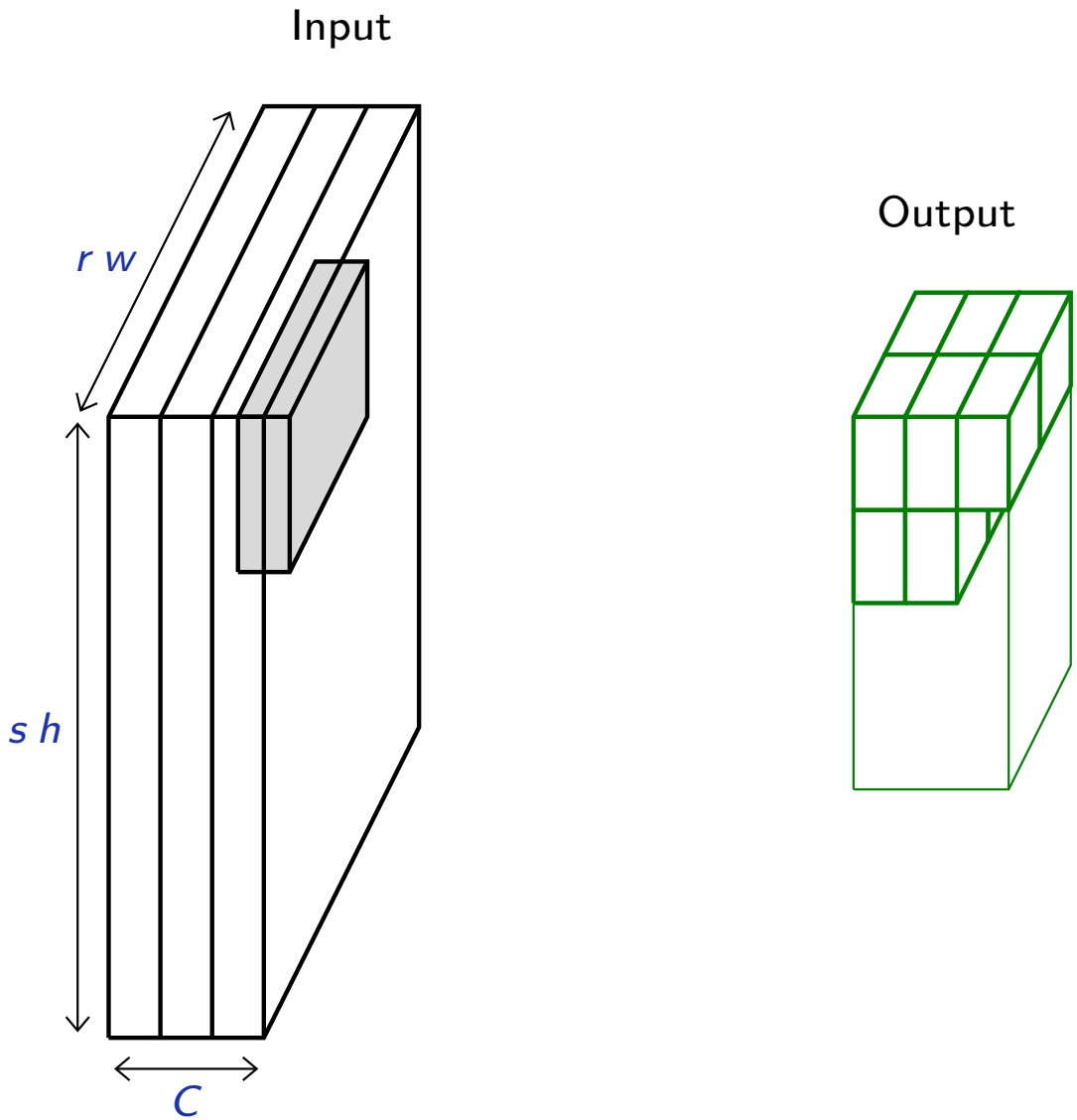
Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



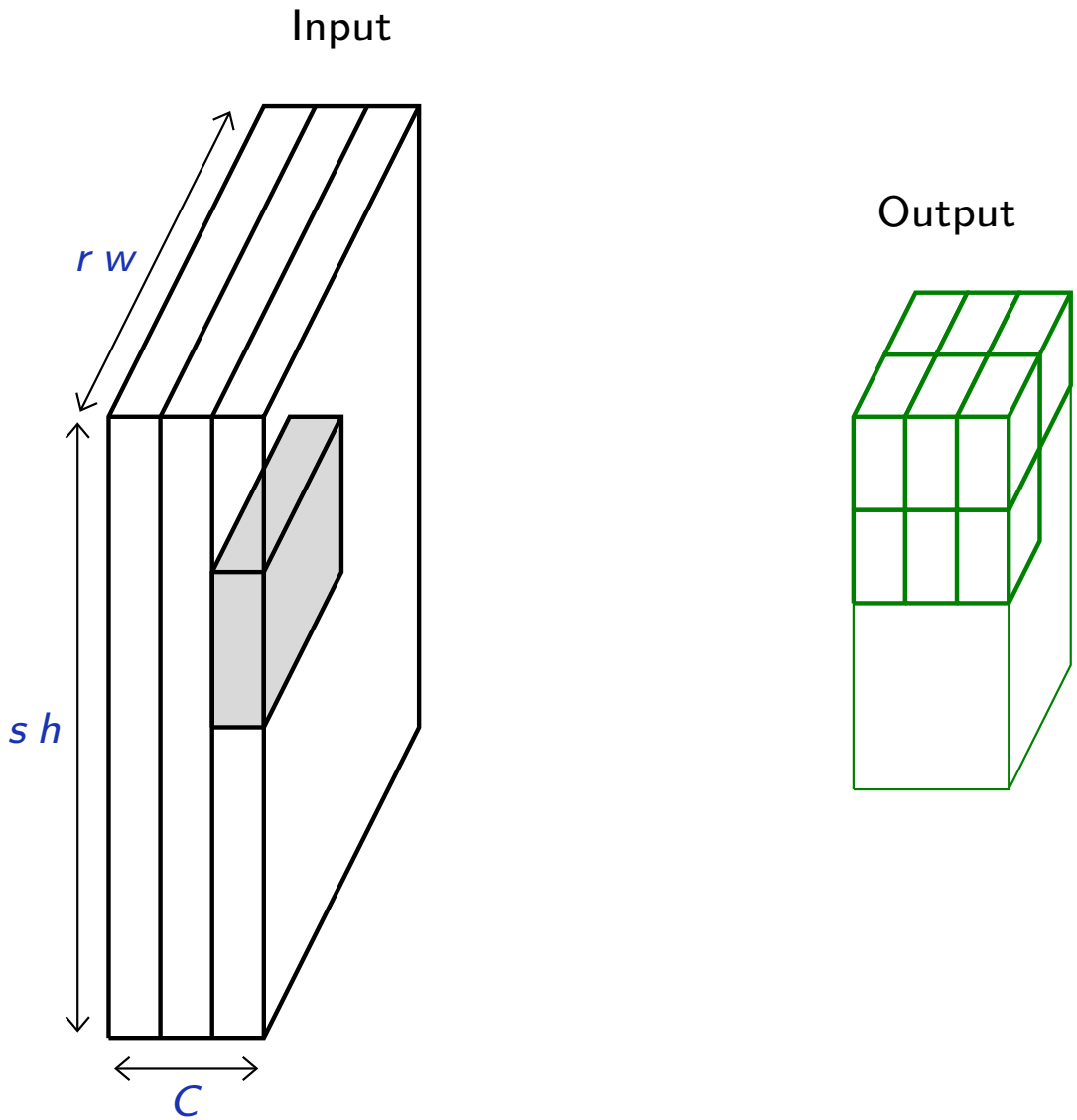
Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



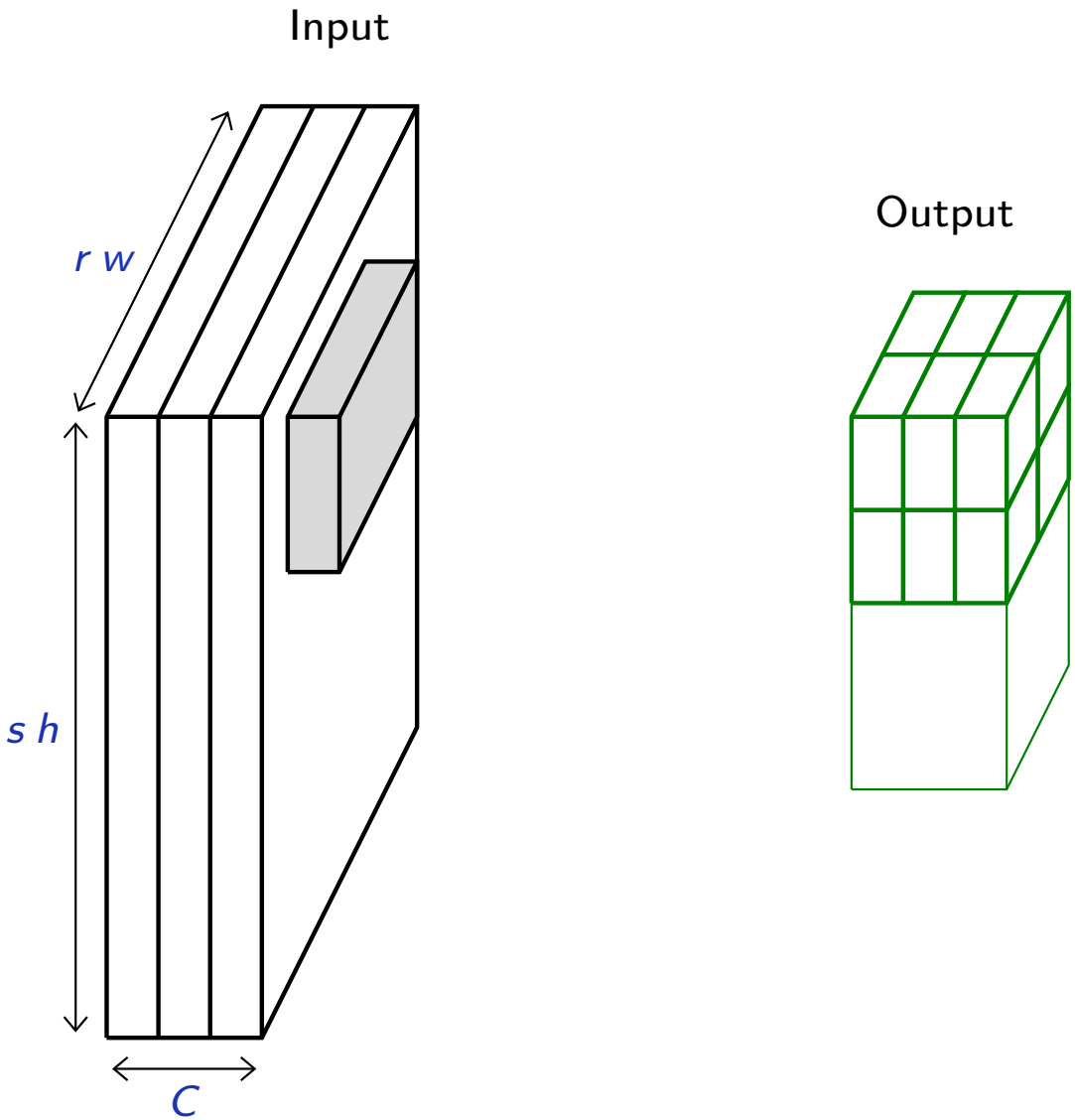
Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



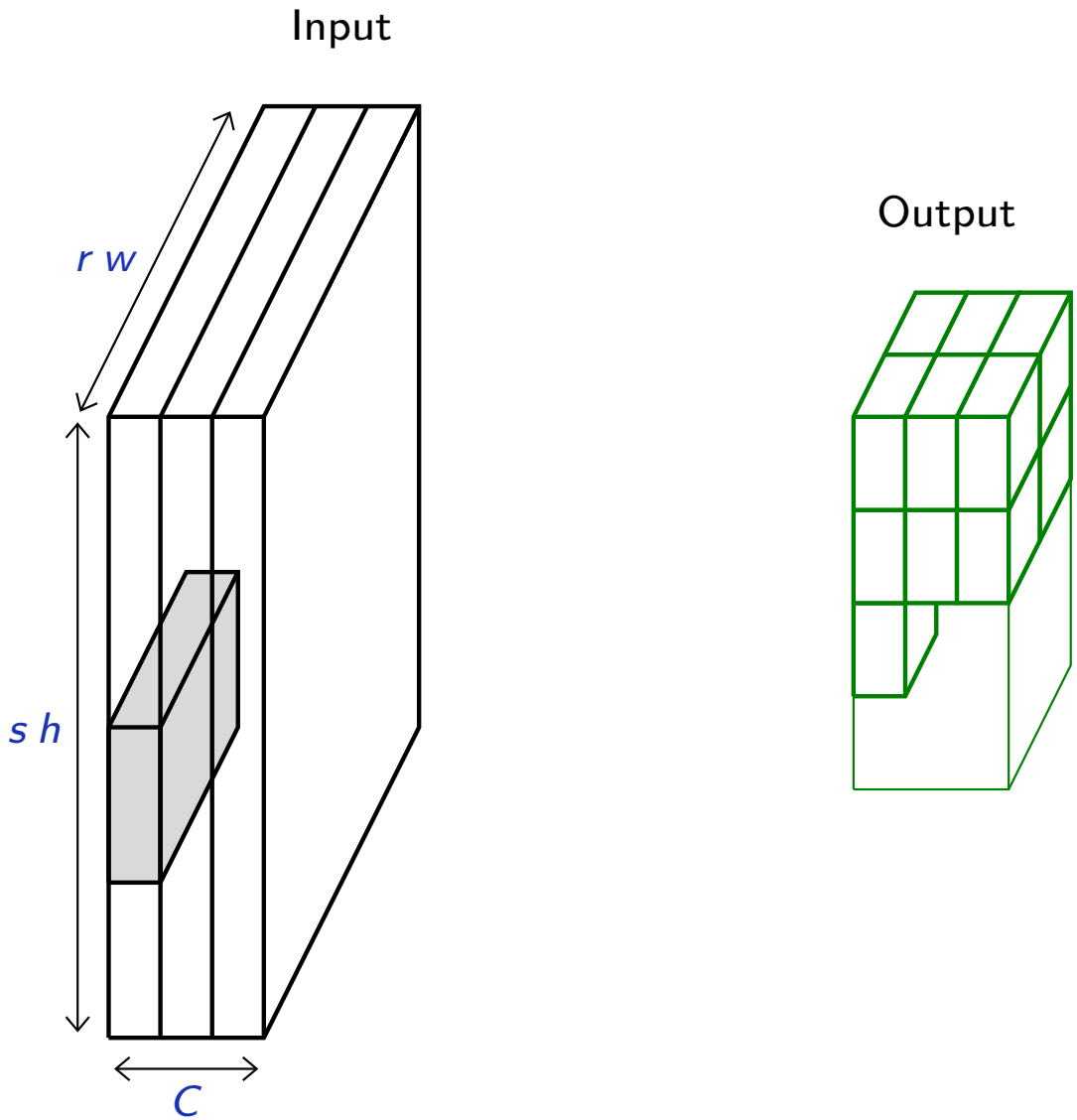
Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



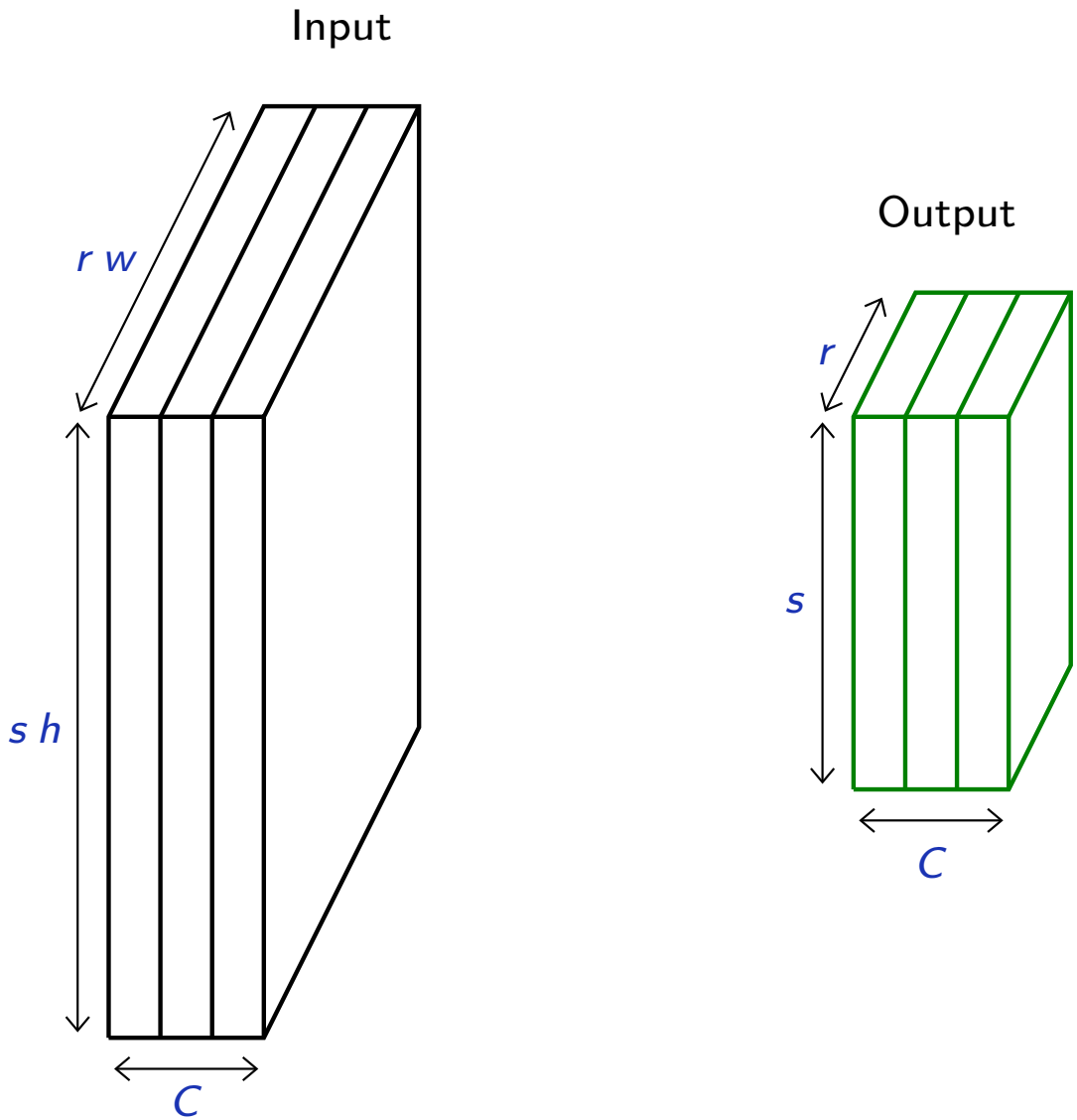
Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.



Pooling with a $w \times h$ kernel. Contrary to convolution, pooling is applied independently on each channel. There are as many channels as output.

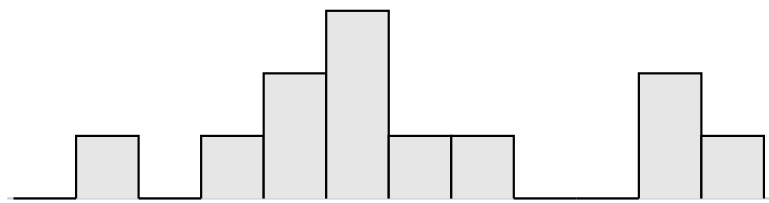
Pooling provides invariance to any permutation inside one of the cell.

More practically, it provides a pseudo-invariance to deformations that result into local translations.

Pooling provides invariance to any permutation inside one of the cell.

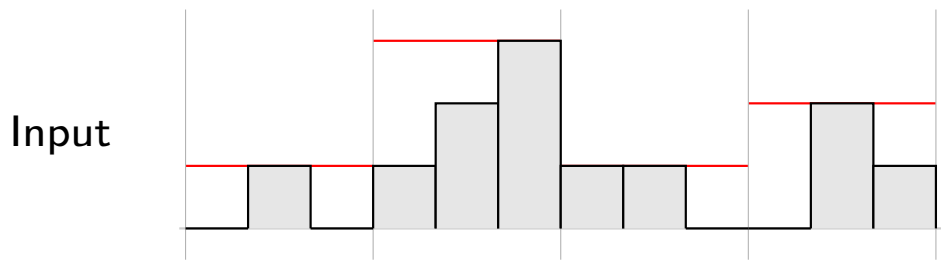
More practically, it provides a pseudo-invariance to deformations that result into local translations.

Input



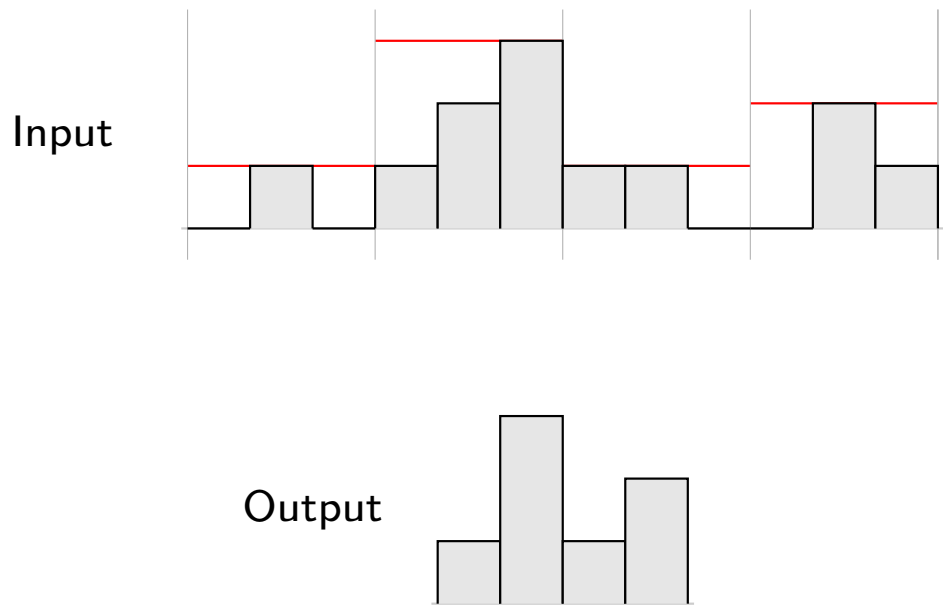
Pooling provides invariance to any permutation inside one of the cell.

More practically, it provides a pseudo-invariance to deformations that result into local translations.



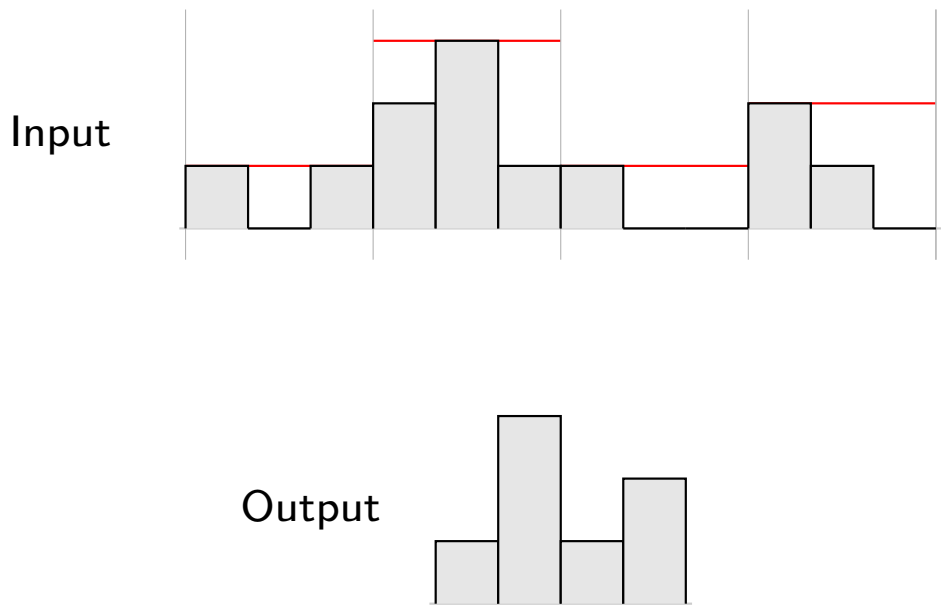
Pooling provides invariance to any permutation inside one of the cell.

More practically, it provides a pseudo-invariance to deformations that result into local translations.



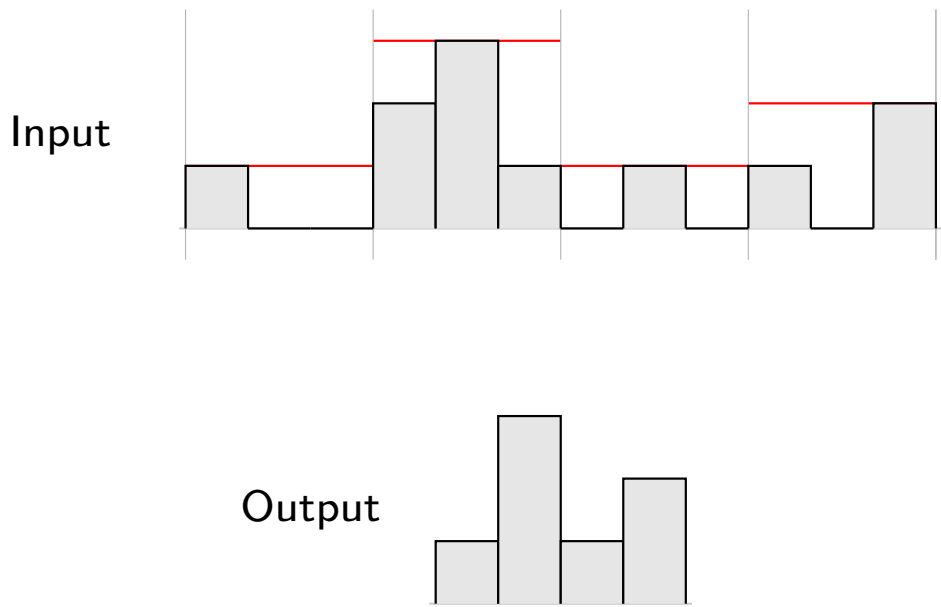
Pooling provides invariance to any permutation inside one of the cell.

More practically, it provides a pseudo-invariance to deformations that result into local translations.



Pooling provides invariance to any permutation inside one of the cell.

More practically, it provides a pseudo-invariance to deformations that result into local translations.



```
F.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 produces (if the padding is 0) a $N \times C \times \lfloor H/h \rfloor \times \lfloor W/w \rfloor$ output.

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

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

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

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

As for convolution, pooling operations can be modulated through their stride and padding.

While for convolution the default stride is 1, for pooling it is equal to the kernel size, but this not obligatory.

Default padding is zero.

```
class torch.nn.MaxPool2d(kernel_size, stride=None,
                         padding=0, dilation=1,
                         return_indices=False, ceil_mode=False)
```

Wraps the max-pooling operation into a `Module`.

As for convolutions, the kernel size is either a pair (h, w) or a single value k interpreted as (k, k).

Deep learning

4.6. Writing a PyTorch module

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

We now have all the bricks needed to build our first convolutional network from scratch. The last technical point is the tensor shape between layers.

Both the convolutional and pooling layers take as input batches of samples, each one being itself a 3d tensor $C \times H \times W$.

The output has the same structure, and tensors have to be explicitly reshaped before being forwarded to a fully connected layer.

```
>>> from torchvision.datasets import MNIST
>>> mnist = MNIST('./data/mnist/', train = True, download = True)
>>> d = mnist.train_data
>>> d.size()
torch.Size([60000, 28, 28])
>>> x = d.view(d.size(0), 1, d.size(1), d.size(2))
>>> x.size()
torch.Size([60000, 1, 28, 28])
>>> x = x.view(x.size(0), -1)
>>> x.size()
torch.Size([60000, 784])
```

A classical LeNet-like model could be:

Input sizes / operations	Nb. parameters	Nb. products
$1 \times 28 \times 28$		
`nn.Conv2d(1, 32, kernel_size=5)`	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460{,}800$
$32 \times 24 \times 24$		
`F.max_pool2d(., kernel_size=3)`	0	0
$32 \times 8 \times 8$		
`F.relu(.)`	0	0
$32 \times 8 \times 8$		
`nn.Conv2d(32, 64, kernel_size=5)`	$64 \times (32 \times 5^2 + 1) = 51{,}264$	$32 \times 64 \times 4^2 \times 5^2 = 819{,}200$
$64 \times 4 \times 4$		
`F.max_pool2d(., kernel_size=2)`	0	0
$64 \times 2 \times 2$		
`F.relu(.)`	0	0
$64 \times 2 \times 2$		
`x.view(-1, 256)`	0	0
256		
`nn.Linear(256, 200)`	$200 \times (256 + 1) = 51{,}400$	$200 \times 256 = 51{,}200$
200		
`F.relu(.)`	0	0
200		
`nn.Linear(200, 10)`	$10 \times (200 + 1) = 2{,}010$	$10 \times 200 = 2{,}000$
10		

Total 105,506 parameters and 1,333,200 products for the forward pass.

Creating a module

PyTorch offers a sequential container module `torch.nn.Sequential` to build simple architectures.

For instance a MLP with a 10 dimension input, 2 dimension output, ReLU activation and two hidden layers of dimensions 100 and 50 can be written as:

```
model = nn.Sequential(
    nn.Linear(10, 100), nn.ReLU(),
    nn.Linear(100, 50), nn.ReLU(),
    nn.Linear(50, 2)
)
```

However for any model of reasonable complexity, the best is to write a sub-class of `torch.nn.Module`.

To create a `Module`, one has to inherit from the base class and implement the constructor `__init__(self, ...)` and the forward pass `forward(self, x)`.

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), kernel_size=3, stride=3))
        x = F.relu(F.max_pool2d(self.conv2(x), kernel_size=2, stride=2))
        x = x.view(-1, 256)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Inheriting from `torch.nn.Module` provides many mechanisms implemented in the superclass.

First, the `(...)` operator is redefined to call the `forward(...)` method and run additional operations. The forward pass should be executed through this operator and not by calling `forward` explicitly.

Using the class `Net` we just defined

```
model = Net()
input = torch.randn(12, 1, 28, 28)
output = model(input)
print(output.size())
```

prints

```
torch.Size([12, 10])
```

Also, the `Parameter`s added as class attributes, or from modules added as class attributes, are seen by `Module.parameters()`.

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(256, 200)
        self.fc2 = nn.Linear(200, 10)
/.../

model = Net()

for n, k in model.named_parameters():
    print(n, k.size())
```

prints

```
conv1.weight torch.Size([32, 1, 5, 5])
conv1.bias torch.Size([32])
conv2.weight torch.Size([64, 32, 5, 5])
conv2.bias torch.Size([64])
fc1.weight torch.Size([200, 256])
fc1.bias torch.Size([200])
fc2.weight torch.Size([10, 200])
fc2.bias torch.Size([10])
```

Parameters added in dictionaries or arrays are not seen.

```
class Buggy(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(torch.zeros(123, 456))
        self.other_stuff = [ nn.Linear(543, 21) ]

model = Buggy()

for k in model.parameters():
    print(k.size())
```

prints

```
param torch.Size([123, 456])
conv.weight torch.Size([32, 1, 5, 5])
conv.bias torch.Size([32])
```

A simple option is to add modules in a `torch.nn.ModuleList`, which is a list of modules properly dealt with by PyTorch's machinery.

```
class NotBuggy(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(torch.zeros(123, 456))
        self.other_stuff = nn.ModuleList()
        self.other_stuff.append(nn.Linear(543, 21))

model = NotBuggy()

for n, k in model.named_parameters():
    print(n, k.size())
```

prints

```
param torch.Size([123, 456])
conv.weight torch.Size([32, 1, 5, 5])
conv.bias torch.Size([32])
other_stuff.0.weight torch.Size([21, 543])
other_stuff.0.bias torch.Size([21])
```

As long as you use autograd-compliant operations, the backward pass is implemented automatically.

This is crucial to allow the optimization of the `Parameter`s with gradient descent.

The end