# EE-559 – Deep learning

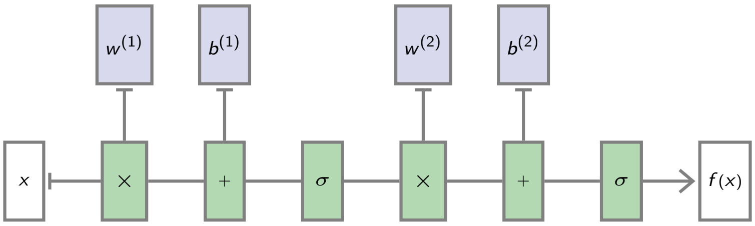
# 4a. DAG networks, autograd, convolution layers

François Fleuret
https://fleuret.org/dlc/

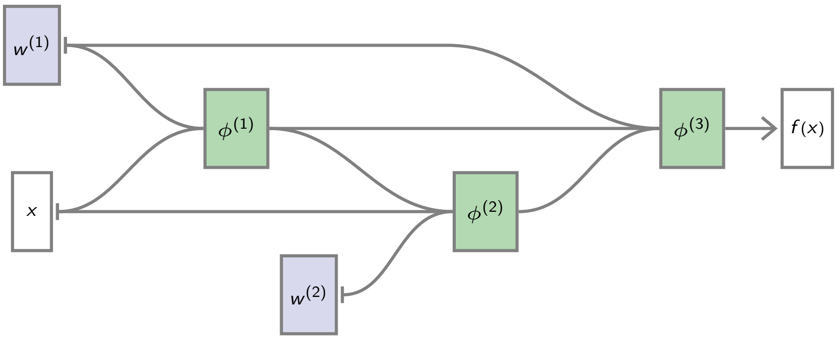
February 15, 2018

# DAG networks

Everything we have seen for an MLP



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



Remember that we use tensorial notation.

If $(a_1, \dots, a_Q) = \phi(b_1, \dots, b_R)$, we have

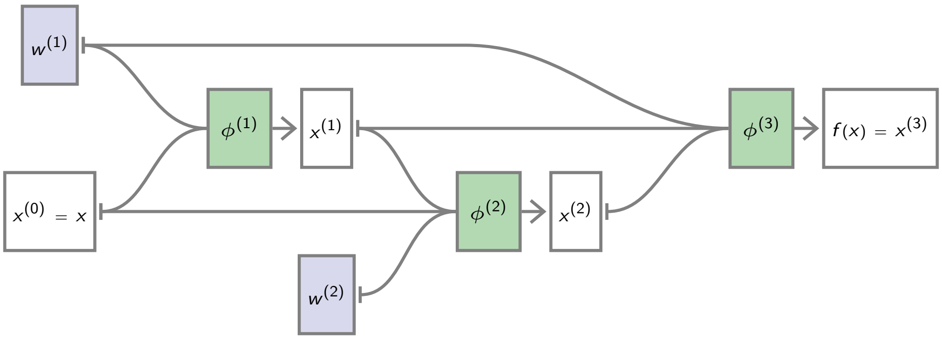
$$\left[\frac{\partial a}{\partial b}\right] = J_\phi = \begin{pmatrix} \frac{\partial a_1}{\partial b_1} & \dots & \frac{\partial a_1}{\partial b_R} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_Q}{\partial b_1} & \dots & \frac{\partial a_Q}{\partial b_R} \end{pmatrix}.$$

There is the usual ambiguity between the mapping and its value. It is almost always the value given the context (*i.e.* for the forward-pass activations).

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

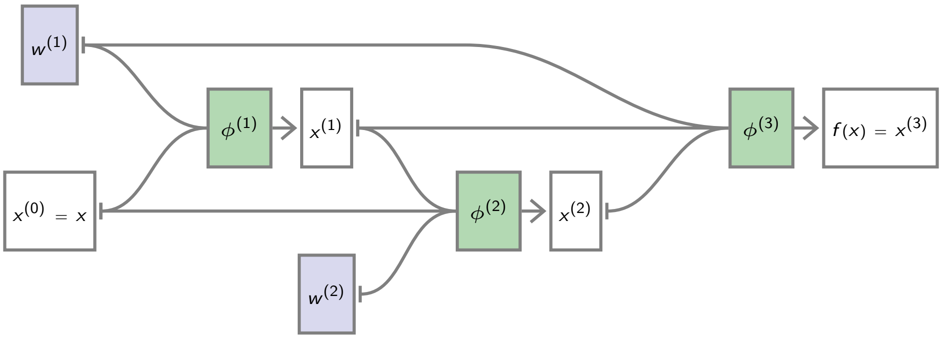
$$J_{\phi|c} = \begin{pmatrix} \frac{\partial a_1}{\partial c_1} & \dots & \frac{\partial a_1}{\partial c_S} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_Q}{\partial c_1} & \dots & \frac{\partial a_Q}{\partial c_S} \end{pmatrix}.$$

## Forward pass



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

## Backward pass, derivatives w.r.t activations



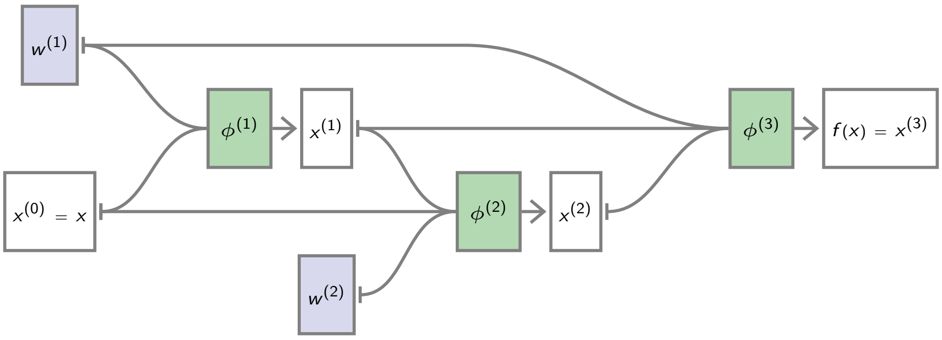
$$
\left[\frac{\partial \ell}{\partial x^{(2)}}\right] = \left[\frac{\partial x^{(3)}}{\partial x^{(2)}}\right]\left[\frac{\partial \ell}{\partial x^{(3)}}\right] = J_{\phi^{(3)}|x^{(2)}}\left[\frac{\partial \ell}{\partial x^{(3)}}\right]
$$

why not with respect to phi 3 ?

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

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

Backward pass, derivatives w.r.t parameters



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

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

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

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

we can build an arbitrary directed acyclic graph with these operators at the nodes, compute the response of the resulting mapping, and compute its gradient with back-prop.

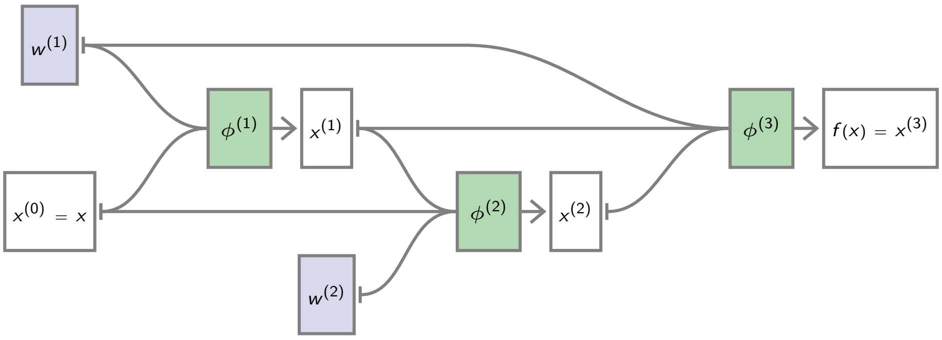
Writing from scratch a large neural network is complex and error-prone.

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** | BSD | Facebook |
| Caffe2 | C++, Python | Apache | Facebook |
| TensorFlow | Python, C++ | Apache | Google |
| MXNet | Python, C++, R, Scala | Apache | Amazon |
| CNTK | Python, C++ | MIT | Microsoft |
| Torch | 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 (Torch, TensorFlow, Caffe, Theano, etc.)

For instance, in TensorFlow, to run a forward/backward pass on



with

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

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

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

we can do

```
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())
    _grads = sess.run(grads)
```

# Autograd

The forward pass is "just" a computation as usual. The graph structure is needed for the backward pass only.

The specification of the graph looks a lot like the forward pass, and the operations of the forward pass fully define those of the backward.

**PyTorch provides `Variable`s, which can be used as `Tensor`s, with the advantage that during any computation, the graph of operations to computes the gradient wrt any quantity is automatically constructed.**

This "autograd" mechanism has two main benefits:

- Simpler syntax: one just need to write the forward pass as a standard computation,
- greater flexibility: Since the graph is not static, the forward pass can be dynamically modulated.

To use autograd, use `torch.autogradVariable` instead of `torch.Tensor`. Most of the `Tensor` operations [have corresponding operations that] accept `Variable`.

A `Variable` is a wrapper around a `Tensor`, with the following fields

- `data` is the `Tensor` containing the data itself,
- `grad` is a `Variable` of same dimension to sum the gradient,
- `requires_grad` is a `Boolean` stating if we need the gradient w.r.t this `Variable` (default is `False`).

A `Parameter` is a `Variable` with `requires_grad` to `True` by default, and known to be a parameter by various utility functions.

⚠ A `Variable` can only embed a `Tensor`, so functions returning a scalar (*e.g.* a loss) now return a 1d `Variable` with a single value.

`torch.autograd.grad(outputs, inputs)` computes and returns the sum of gradients of outputs wrt the specified inputs. This is always a `tuple` of `Variable`.

An alternative is to use `torch.autograd.backward(variables)` or `Variable.backward()`, which accumulates the gradients in the `grad` fields of the leaf `Variable`s.

Consider a simple example $(x_1, x_2, x_3) = (1, 2, 2)$, and

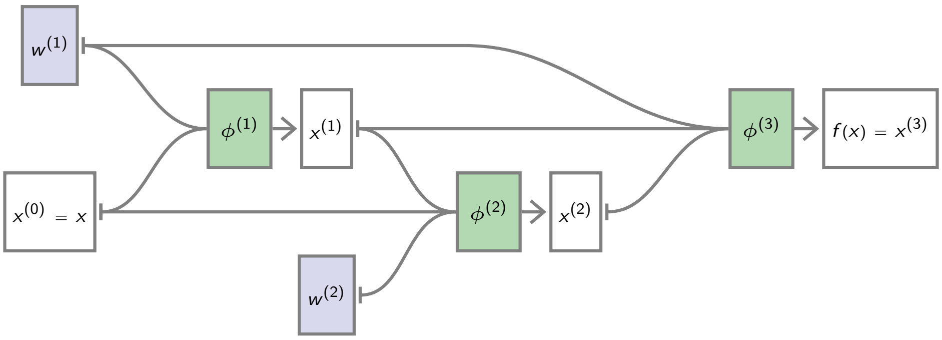
$$\ell = \|x\| = \sqrt{x_1^2 + x_2^2 + x_3^2}.$$

We have $\ell = 3$ and

$$\frac{\partial \ell}{\partial x_i} = \frac{x_i}{\|x\|}.$$

```
>>> from torch import Tensor
>>> from torch.autograd import Variable
>>> x = Variable(Tensor([1, 2, 2]))
>>> l = x.norm()
>>> l
Variable containing:
 3
[torch.FloatTensor of size 1]

>>> l.backward()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/fleuret/misc/anaconda3/lib/python3.5/site-packages/torch/
        autograd/variable.py", line 146, in backward
    self._execution_engine.run_backward((self,), (gradient,),
          retain_variables)
RuntimeError: there are no graph nodes that require computing gradients

>>> x.requires_grad = True
>>> q = x.norm()
>>> q.backward()
>>> x.grad
Variable containing:
 0.3333
 0.6667
 0.6667
[torch.FloatTensor of size 3]
```

For instance, in PyTorch, to run a forward/backward pass on



with

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

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

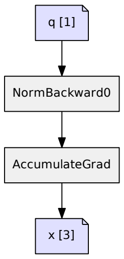
$$\phi^{(3)}\left(x^{(1)}, x^{(2)}; w^{(1)}\right) = w^{(1)}\left(x^{(1)} + x^{(2)}\right)$$

we can do

```
w1 = Parameter(Tensor(5, 5).normal_())
w2 = Parameter(Tensor(5, 5).normal_())
x = Variable(Tensor(5).normal_())

x0 = x
x1 = w1.mv(x0)
x2 = x0 + w2.mv(x1)
x3 = w1.mv(x1 + x2)

q = x3.norm()

q.backward()
```

We can look precisely at the graph built during a computation.

```
x = Parameter(Tensor([1, 2, 2]))
q = x.norm()
```



This graph was generated with

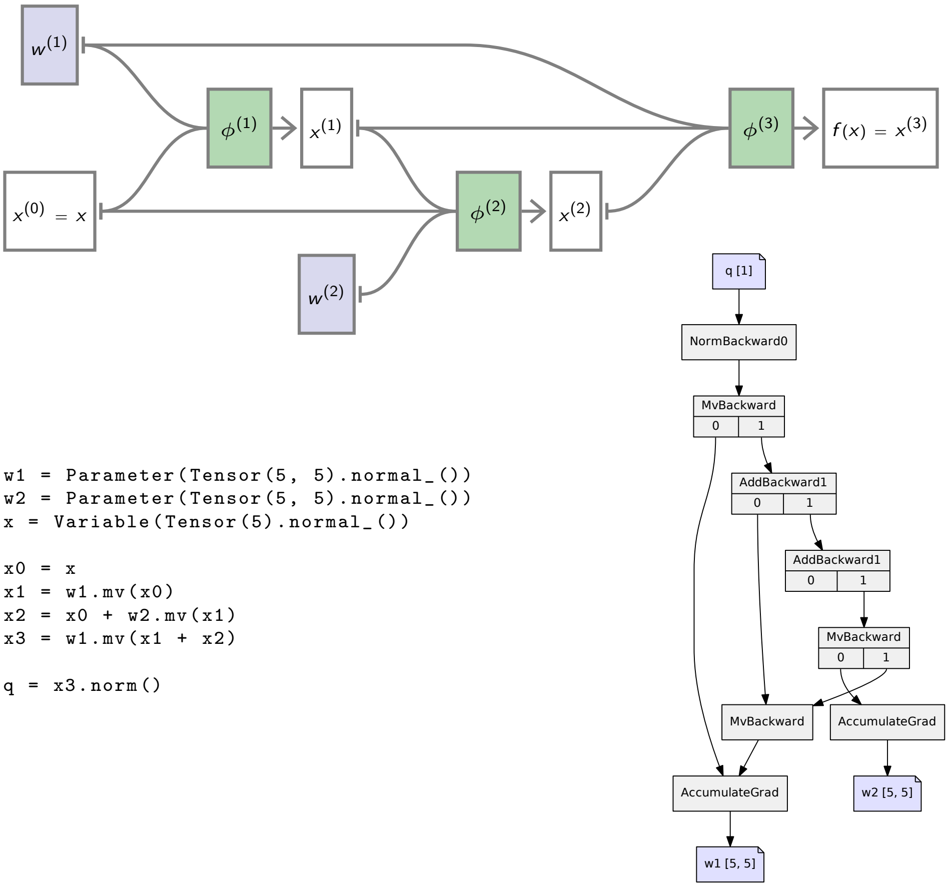
`https://fleuret.org/git/agtree2dot`

and Graphviz.

Each `Variable` has a reference to the `Function` it should send its gradient to through the `grad_fn` field.

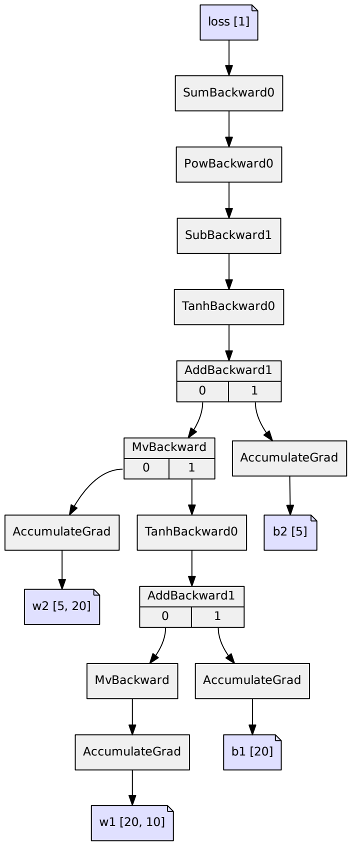
```
>>> x = Variable(Tensor(10).fill_(1.0))
>>> m = Variable(Tensor(2, 10).fill_(2.0))
>>> y = m.mv(x)
>>> z = y + 3
>>> z.grad_fn
<torch.autograd.function.AddConstantBackward object at 0x7f2baedfdb88>
>>> y.grad_fn
<torch.autograd.function.AddmvBackward object at 0x7f2bae3cf228>
```

Each `Function` has references to the functions it has to send its own gradient to, and some have references to `Variable`s for accumulating gradients.

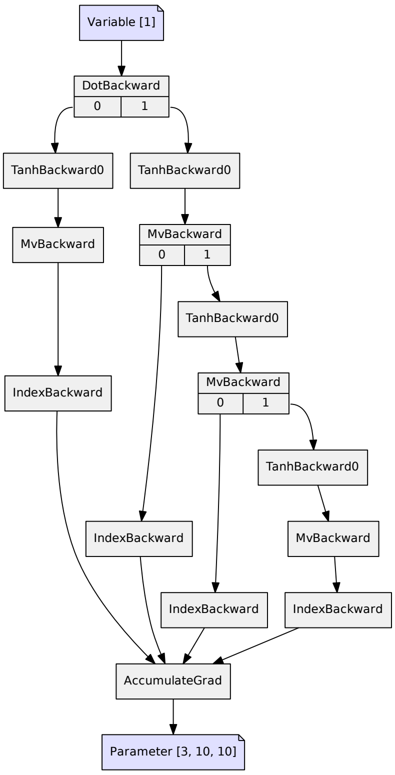


```
w1 = Parameter(Tensor(5, 5).normal_())
w2 = Parameter(Tensor(5, 5).normal_())
x = Variable(Tensor(5).normal_())

x0 = x
x1 = w1.mv(x0)
x2 = x0 + w2.mv(x1)
x3 = w1.mv(x1 + x2)

q = x3.norm()
```

```
w1 = Parameter(Tensor(20, 10))
b1 = Parameter(Tensor(20))
w2 = Parameter(Tensor(5, 20))
b2 = Parameter(Tensor(5))

x = Variable(Tensor(10).normal_())
h = torch.tanh(w1.mv(x) + b1)
y = torch.tanh(w2.mv(h) + b2)

target = Variable(Tensor(5).normal_())

loss = (y - target).pow(2).sum()
```



```
w = Parameter(Tensor(3, 10, 10))

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

u = blah(1, Variable(Tensor(10)))
v = blah(3, Variable(Tensor(10)))
q = u.dot(v)
```



⚠ `Variable.backward()` **accumulates** the gradients in the different `Variable`s, so one may have to zero them before.

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.

⚠ 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.

Autograd can generate the computational graph for computing **higher-order derivatives**.

This is done by passing `create_graph=True` to `torch.autograd.grad(...)`

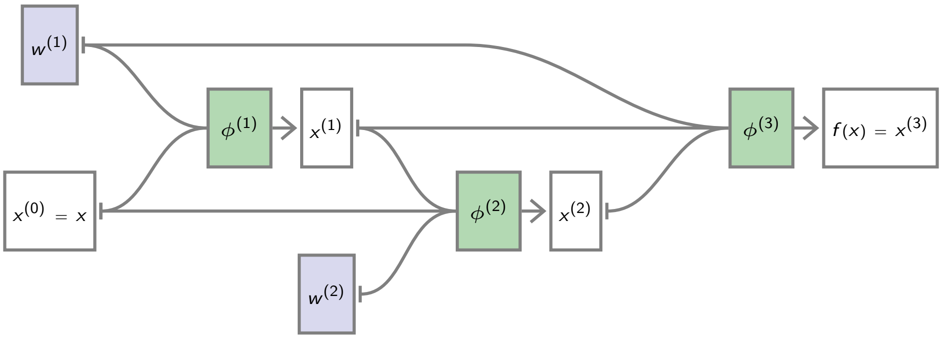
```
>>> x = Variable(Tensor([ 1, 2, 3 ]), requires_grad = True)
>>> s1 = x.pow(2).sum()
>>> g1, = torch.autograd.grad(s1, x, create_graph = True)
>>> g1
Variable containing:
 2
 4
 6
[torch.FloatTensor of size 3]
```

```
>>> s2 = g1[0].exp() - g1[2].exp()
>>> g2, = torch.autograd.grad(s2, x)
>>> g2
Variable containing:
  14.7781
   0.0000
-806.8576
[torch.FloatTensor of size 3]
```

# Weight sharing

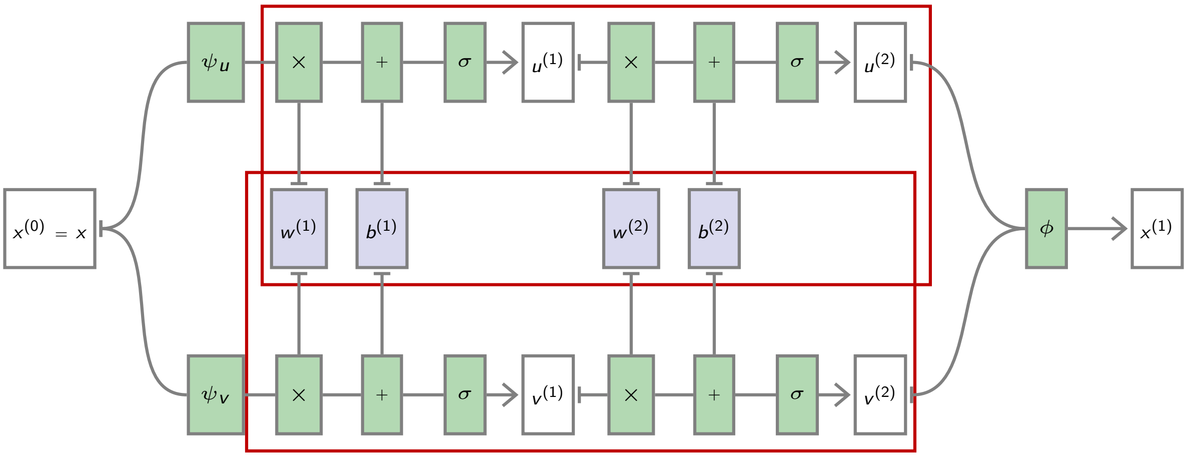
In our generalized DAG formulation, we have in particular implicitly allowed the same parameters to modulate different parts of the processing.

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.



# Convolutional layers

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

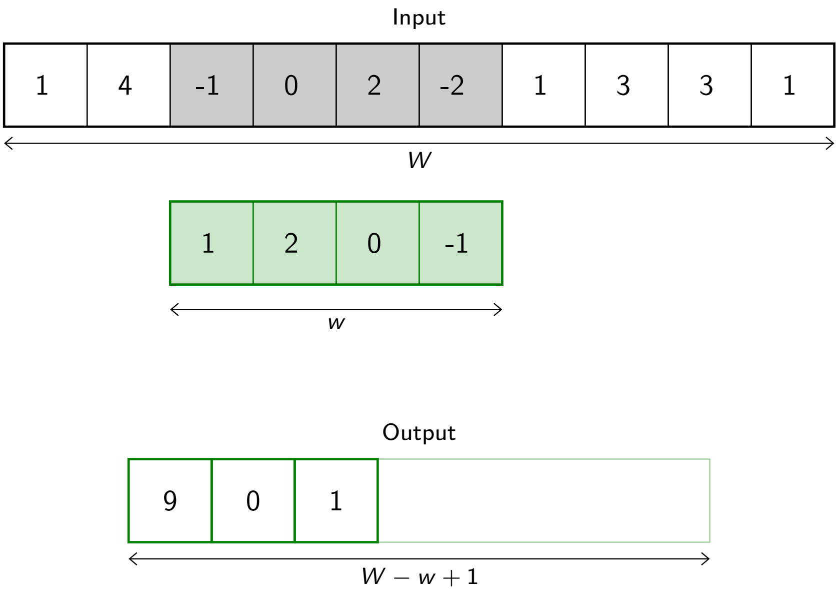
For instance a linear layer taking a $256 \times 256$ RGB image as input, and producing an image of same size would require

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

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 representation meaningful at a certain location can / should be used everywhere.**

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



Formally, in 1d, given

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

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

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

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

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

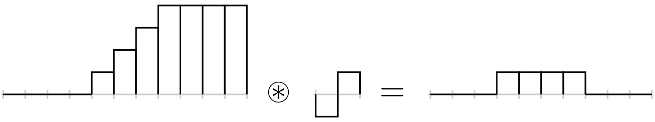
for instance

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

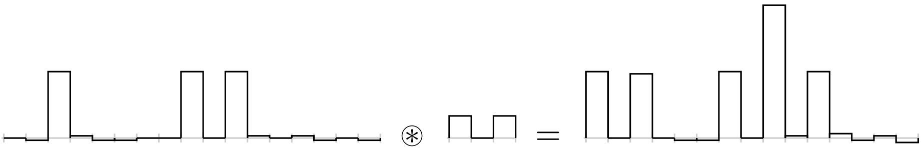
⚠ This differs from the usual convolution since the kernel and the signal are both visited in increasing index order.

Convolution can implement a differential operator

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



or a crude "template matcher"



Both of these computation examples are indeed "invariant by translation".

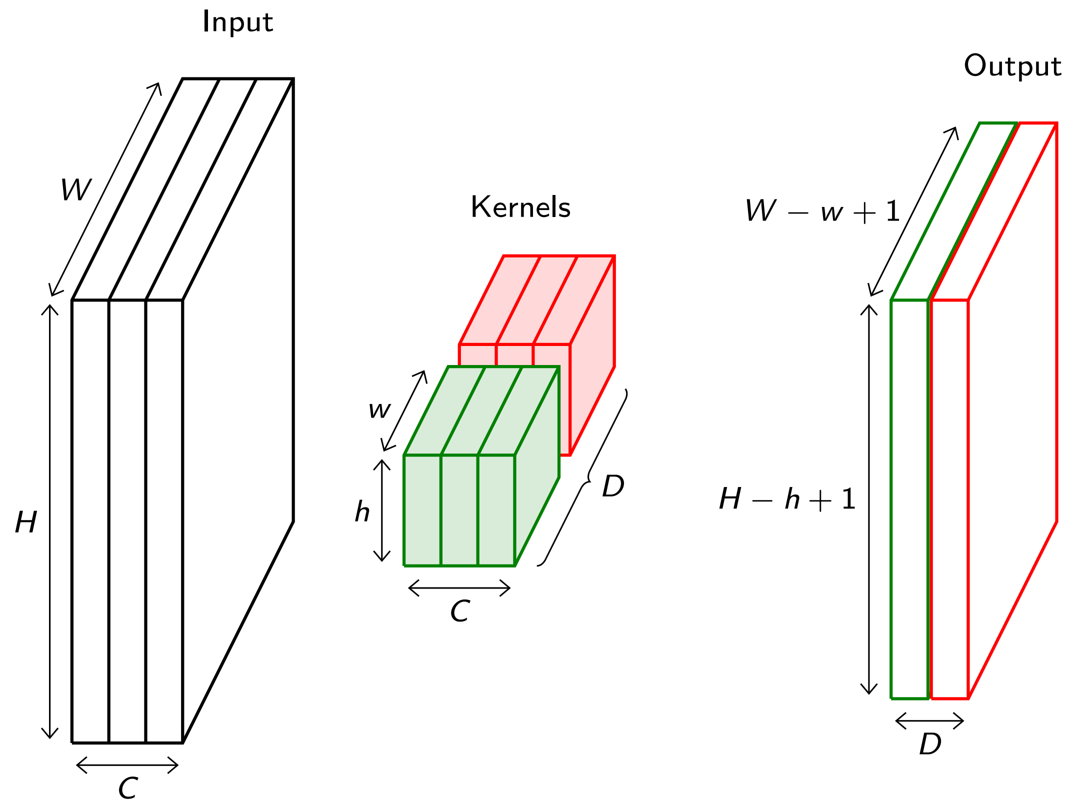
It generalizes naturally to a multi-dimensional input, although specification can become complicated.

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.

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

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

⚠ 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.



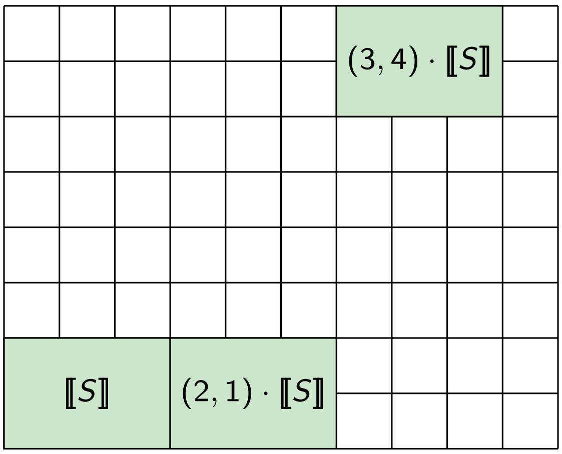
More formally, if for $(n_1, \dots, n_D) \in \mathbb{N}^D$, we use the notation

$$[\![n_1, \dots, n_D]\!] = \{1, \dots, n_1\} \times \dots \times \{1, \dots, n_D\},$$

with the left-product by a tuple of integers defined as follows

$$(k_1, \dots, k_D) \cdot [\![n_1, \dots, n_D]\!] = \{1 + (k_1 - 1)n_1, \dots, k_1 n_1\} \times \dots$$

For instance, with $S = (3, 2)$



Let $S = (C, H, W)$ be the input tensor size, and $s = (c, h, w)$ be the kernel size.

Given an input

$$x \in \mathbb{R}^{C \times H \times W} = \mathbb{R}^{\llbracket S \rrbracket}$$

and a kernel of size

$$u \in \mathbb{R}^{c \times h \times w} = \mathbb{R}^{\llbracket s \rrbracket}$$

we have

$$\forall a \in \llbracket S - s + 1 \rrbracket, \ (x \circledast u)(a) = \sum_{b \in \llbracket s \rrbracket} x_{a+b-1} \, u_b.$$

⚠ We can define a 3d convolution, but if the channel ordering is meaningless, moving across channels makes no sense.

Note that 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.

In particular the convolution of a $C \times H \times W$ tensor with a $C \times 1 \times 1$ kernel can be interpreted as applying the same linear classifier at every point separately.

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

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

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

# Pooling

In many cases, a feed-forward network computes a low-dimension signal (*e.g.* a few scores) from a very high-dimension signal (*e.g.* an image).

As for convolution, it makes sense to reduce the signal's size in a way that preserves its structure, just "down-scaling it".

This operation is called **pooling**, and aims at grouping several activations into a single "more meaningful" one.

Given a pooling area size $h \times w$, and a 3d input tensor

$$x \in \mathbb{R}^{C \times (rh) \times (sw)},$$

there are two main types of pooling, both producing a tensor

$$y \in \mathbb{R}^{C \times r \times s}.$$

Let $u = (1, w, h)$ and $M = (C, r, s)$.

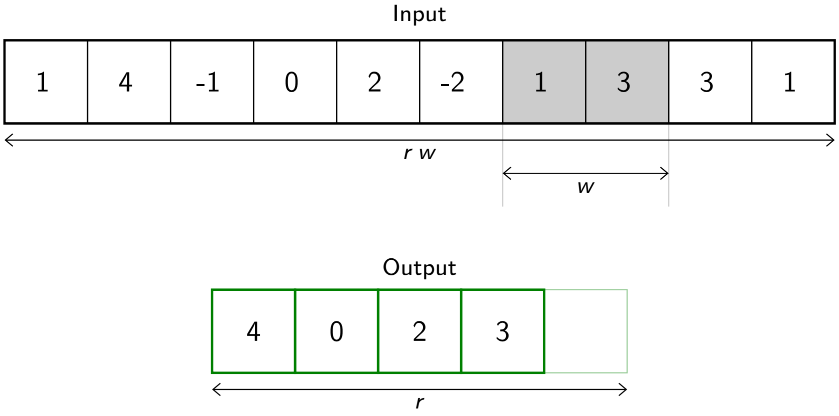
- **Average pooling**

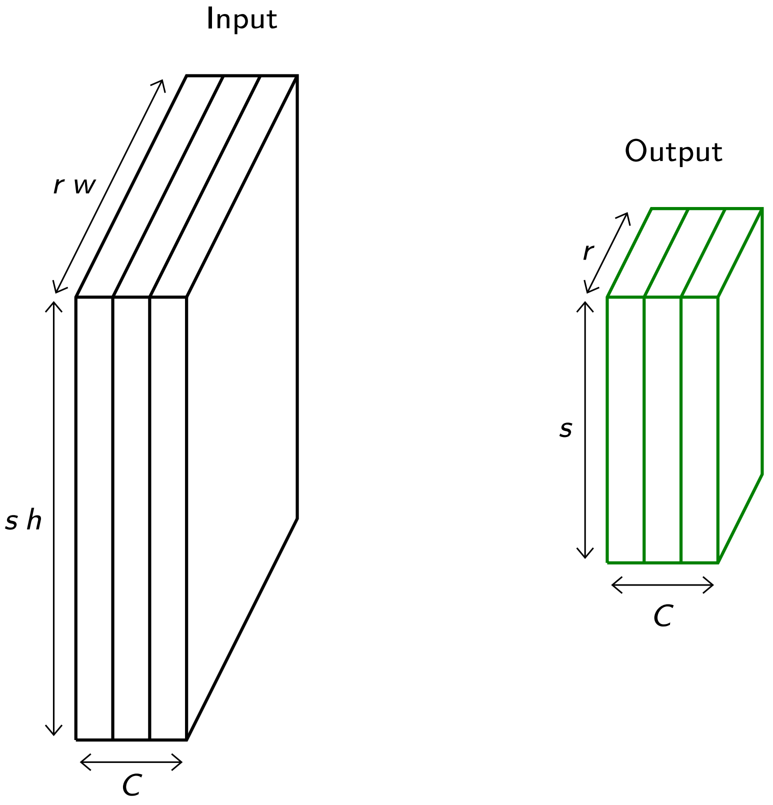
$$\forall a \in \llbracket M \rrbracket,\ y_a = \frac{1}{C\,w\,h} \sum_{b \in a \cdot \llbracket u \rrbracket} x_b.$$

- **Max-pooling**

$$\forall a \in \llbracket M \rrbracket,\ y_a = \max_{b \in a \cdot \llbracket u \rrbracket} x_b.$$

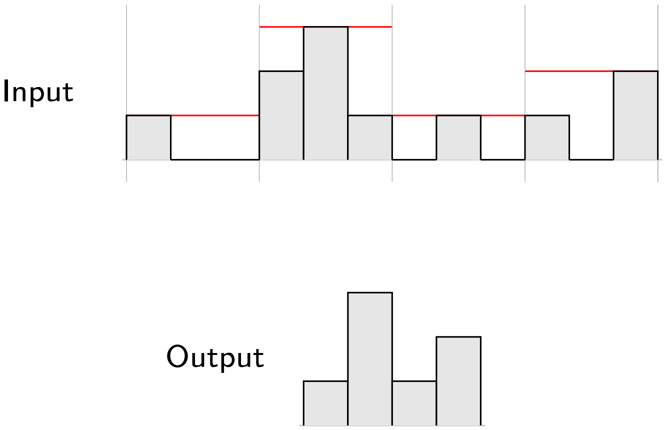
1d example of max-pooling with a kernel of size 2:





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.



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.

```
>>> mnist = datasets.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 = d.view(d.size(0), -1)
>>> x.size()
torch.Size([60000, 784])
```