EE-559 – Deep learning

4a. DAG networks, autograd, convolution layers

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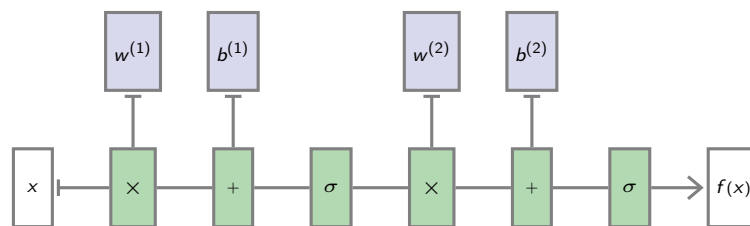
https://fleuret.org/dlc/

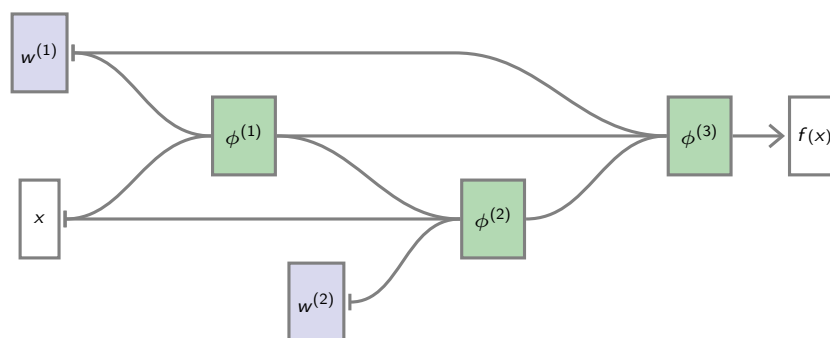[version of: June 24, 2018]

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.

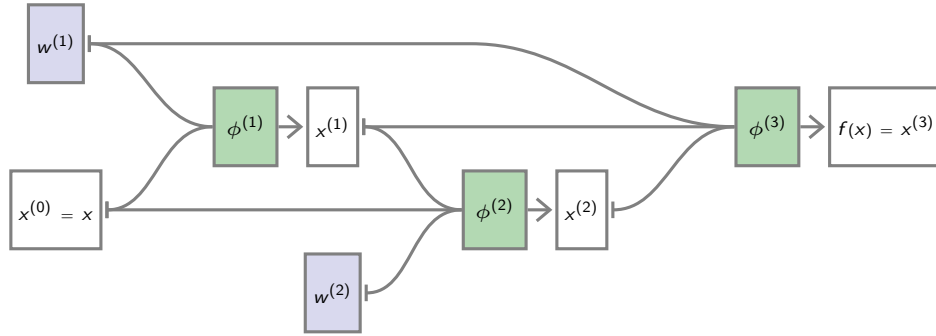If $(a_1, \ldots, a_Q) = \phi(b_1, \ldots, b_R)$, we have

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

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

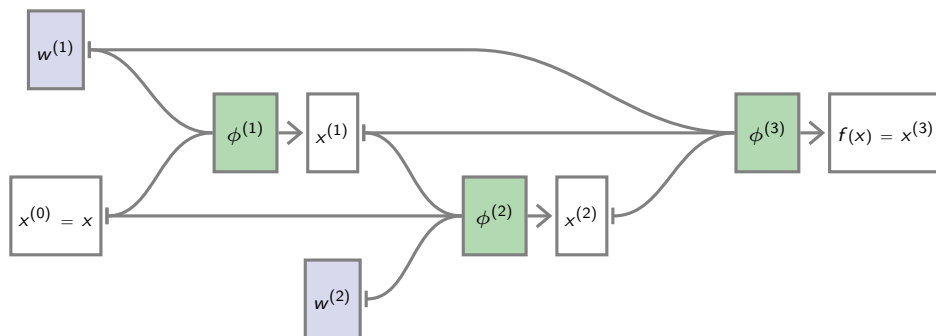Also, if $(a_1, \ldots, a_Q) = \phi(b_1, \ldots, b_R, c_1, \ldots, c_S)$, we use

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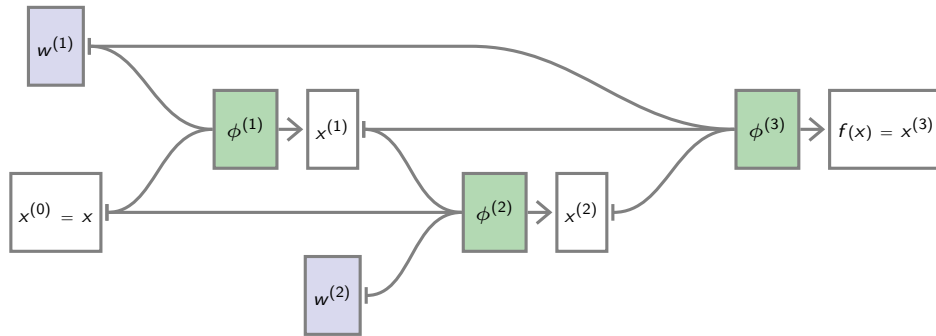
Forward pass

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

Backward pass, derivatives w.r.t activations

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

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

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

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

$$(x_1, \ldots, x_d, w) \mapsto \phi(x_1, \ldots, x_d; w)$$
$$\forall c, \ (x_1, \ldots, x_d, w) \mapsto J_{\phi|x_c}(x_1, \ldots, x_d; w)$$
$$(x_1, \ldots, x_d, w) \mapsto J_{\phi|w}(x_1, \ldots, x_d; w),$$

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
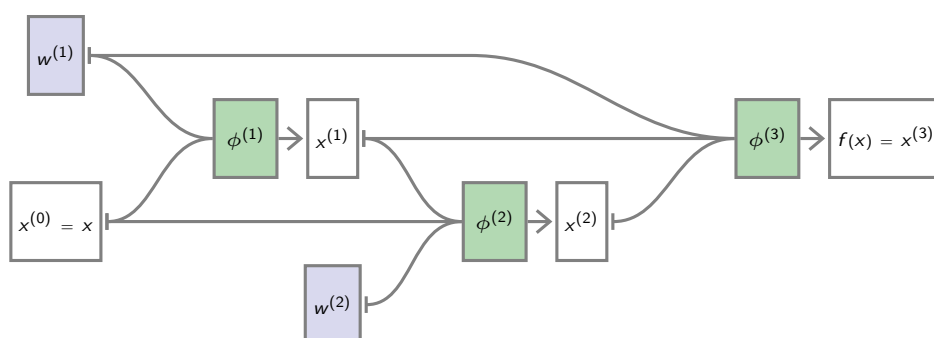
we can do

with

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

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

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

```
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])
```

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 sequence of Python operations,
- greater flexibility: Since the graph is not static, the forward pass can be dynamically modulated.

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

A `Variable` is first a wrapper around a `Tensor`. It has 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]), requires_grad = True)
>>> l = x.norm()
>>> l
Variable containing:
 3
[torch.FloatTensor of size 1]

>>> g = torch.autograd.grad(l, x)
>>> g
(Variable containing:
 0.3333
 0.6667
 0.6667
[torch.FloatTensor of size 3]
,)
```

Alternatively, `Variable.backward()` accumulates the gradient in the variable's `grad` fields.

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

>>> l.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
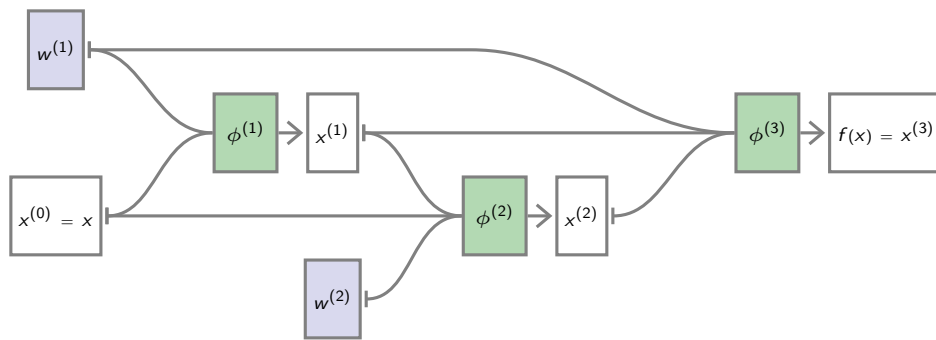
with

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

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

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

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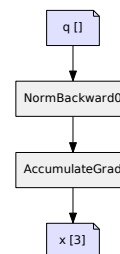
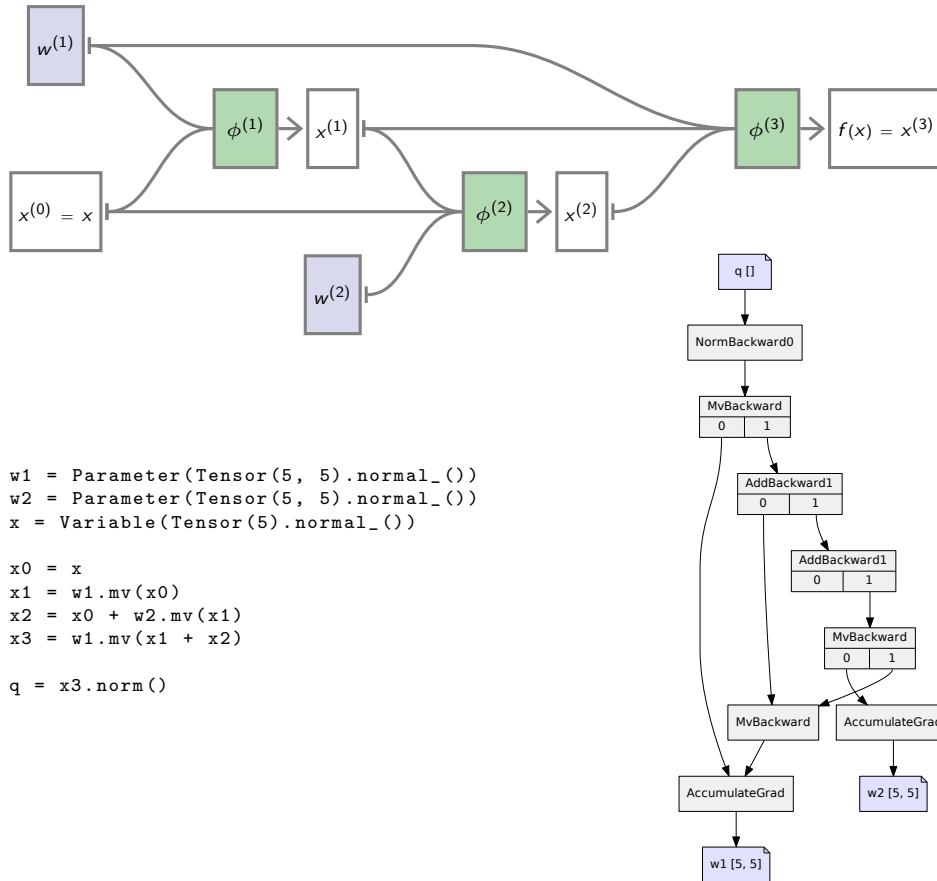We can look precisely at the graph built during a computation.

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


This graph was generated with

$$\texttt{https://fleuret.org/git/agtree2dot}$$

and Graphviz.

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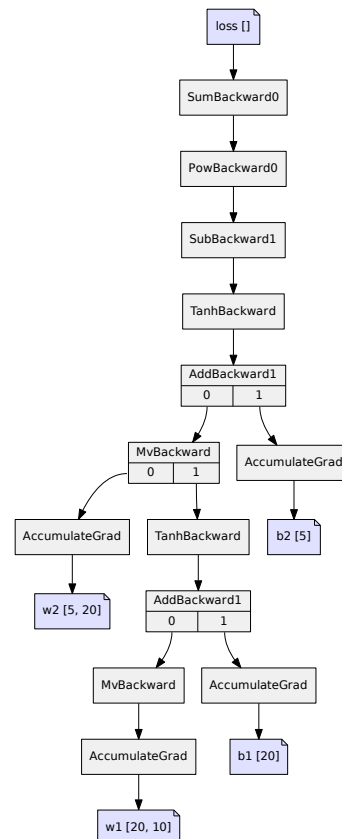
```
w1 = Parameter(Tensor(20, 10))
b1 = Parameter(Tensor(20))
w2 = Parameter(Tensor(5, 20))
b2 = Parameter(Tensor(5))

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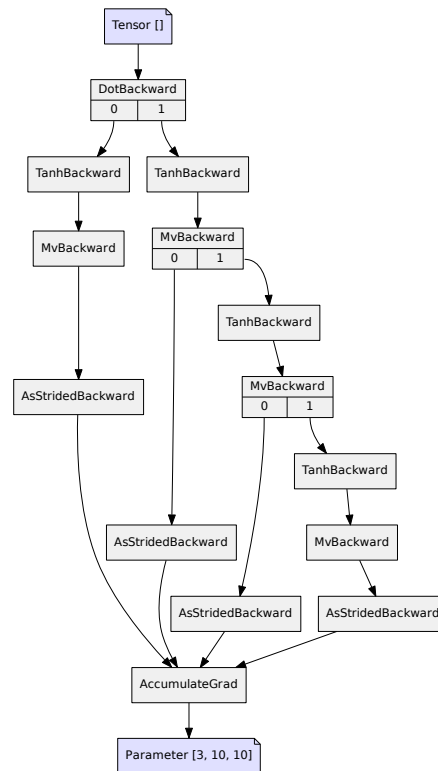
```
w = Parameter(Tensor(3, 10, 10))

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.

```
>>> x = Variable(Tensor([3, 4]), requires_grad = True)
>>> a = x.norm()
>>> a.backward()
>>> b = x.dot(Variable(Tensor([-10, 10])))
>>> b.backward()
>>> x.grad
Variable containing:
 -9.4000
 10.8000
[torch.FloatTensor of size 2]


>>> x = Variable(Tensor([3, 4]), requires_grad = True)
>>> q = x.norm() + x.dot(Variable(Tensor([-10, 10])))
>>> q.backward()
>>> x.grad
Variable containing:
 -9.4000
 10.8000
[torch.FloatTensor of size 2]
```

⚠️ 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.

Finally, since the gradient itself is a `Variable`, 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 $\phi^{(3)}$.
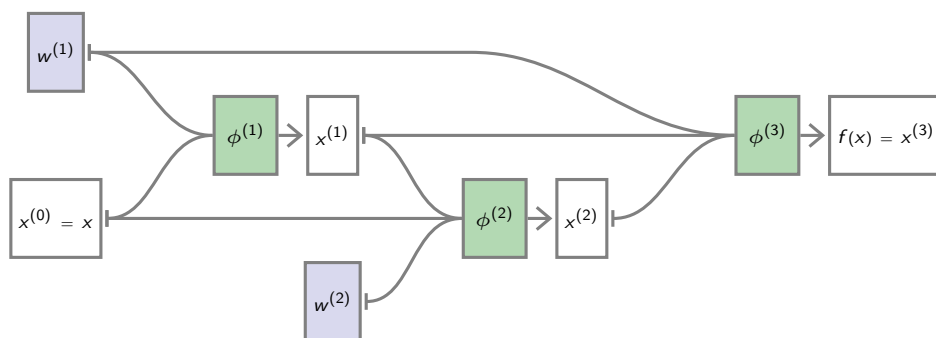
This is called **weight sharing**.

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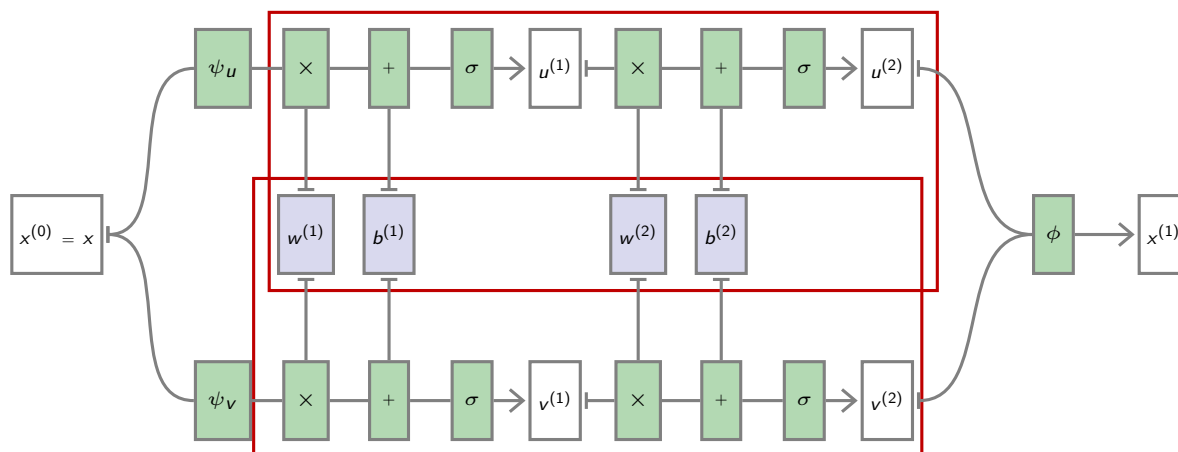
Convolutional layers

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 256×256 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.

Input

1	4	-1	0	2	-2	1	3	3	1

$$W$$

1	2	0	-1

$$w$$

Output

9	0	1				

$$W - w + 1$$
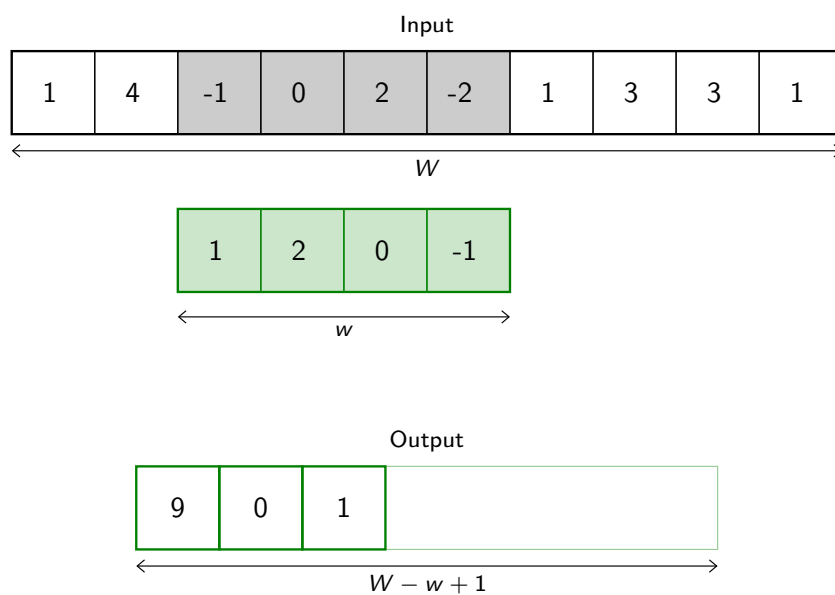
Formally, in 1d, given

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

and a "convolutional 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

$$(x \circledast u)_i = \sum_{j=1}^{w} x_{i-1+j} \, u_j$$

$$= (x_i, \ldots, x_{i+w-1}) \cdot u$$
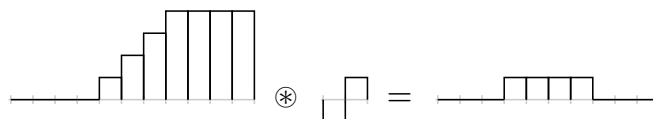
for instance

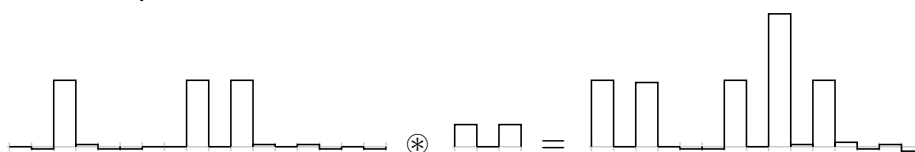$$(1, 2, 3, 4) \circledast (3, 2) = (3 + 4, 6 + 6, 9 + 8) = (7, 12, 17).$$

⚠ 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. 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 convolutional layer, D such convolutions are combined to generate a $D \times (H - h + 1) \times (W - w + 1)$ output.
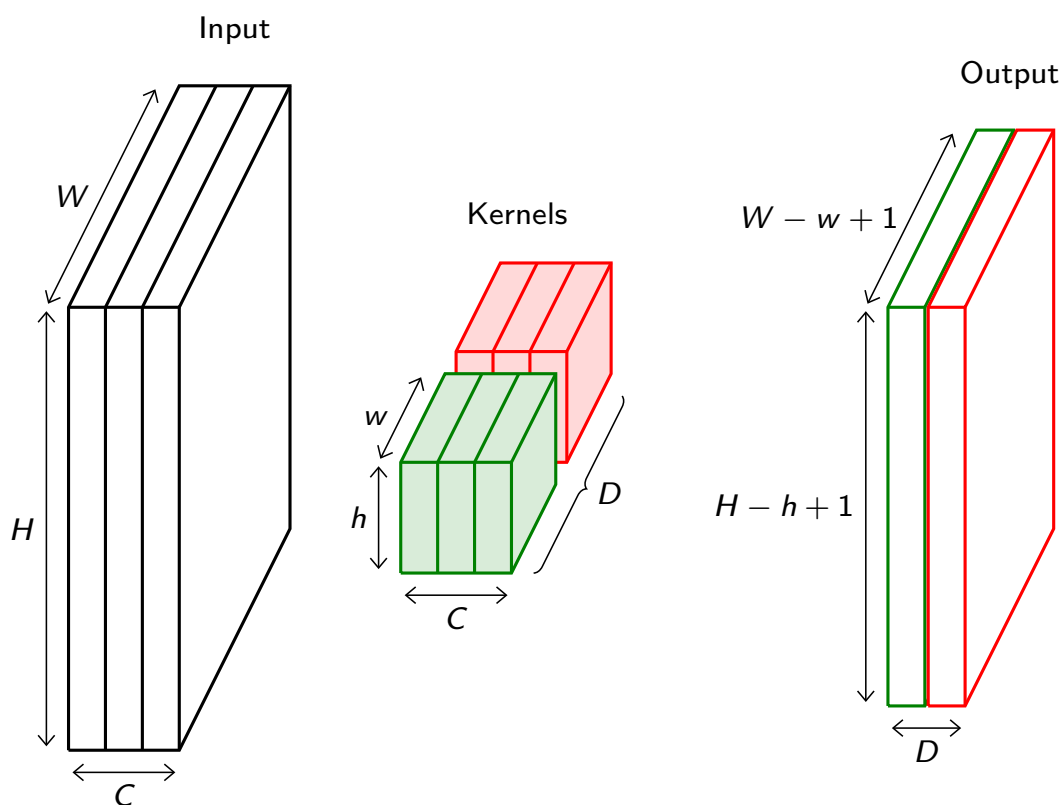
Note that a convolution **preserves the signal support structure**.

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

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

We usually refer to one of the channels generated by a 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.
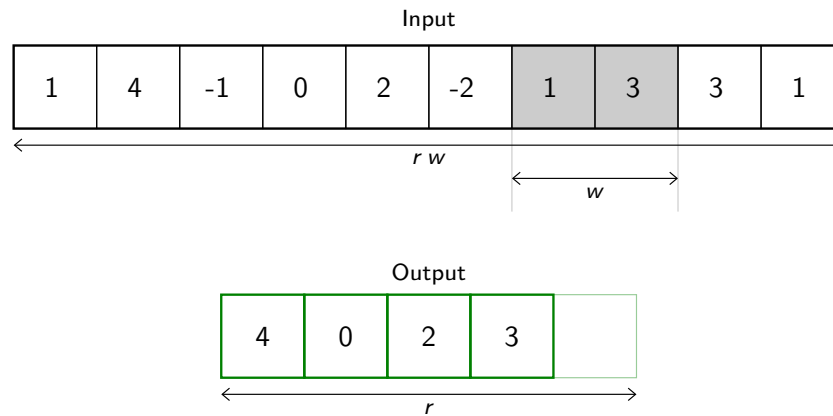
Pooling

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.

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:

Input

1	4	-1	0	2	-2	1	3	3	1

Output

4	0	2	3	

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

Input

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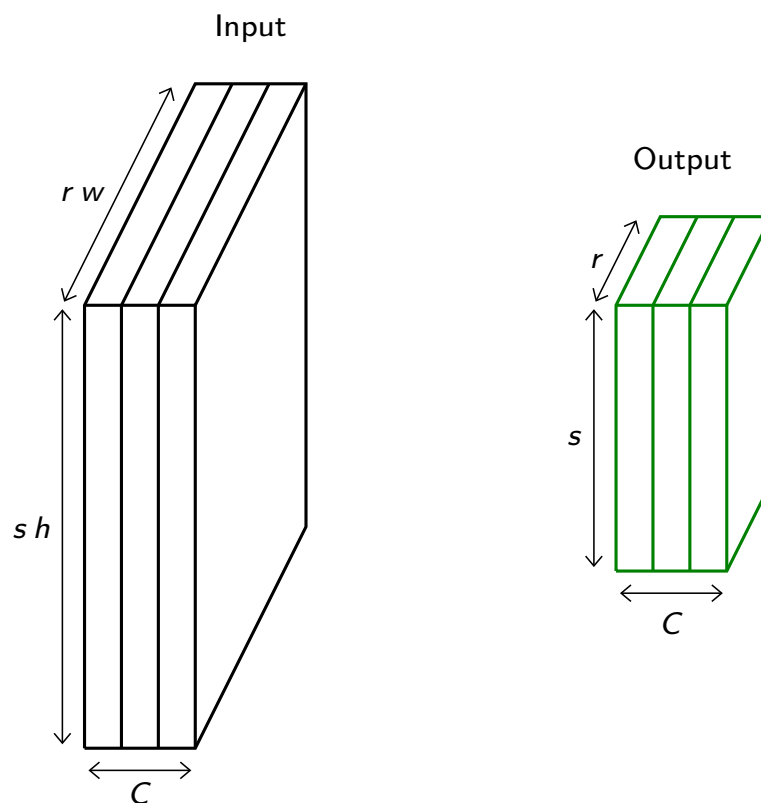
Pooling provides invariance to any permutation inside one of the cell.

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

Input

Output