

CAS – Deep learning

2. DAG networks, auto-grad, image processing

François Fleuret

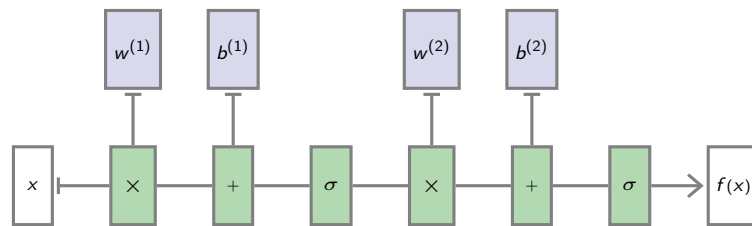
<https://www.idiap.ch/~fleuret/>

Fri Feb 22 13:19:03 UTC 2019

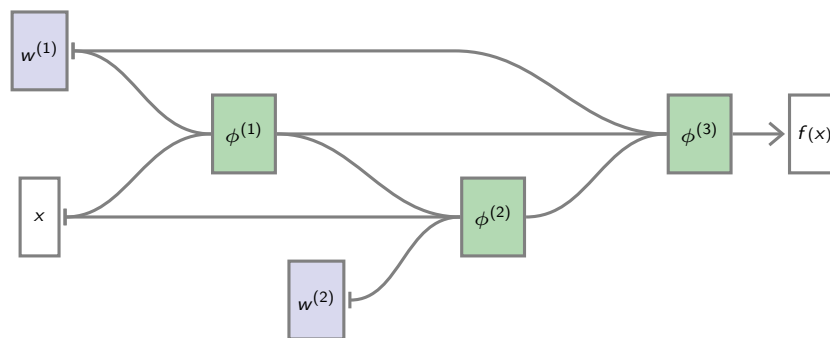


4.1. DAG networks

We can generalize an MLP



to an arbitrary “Directed Acyclic Graph” (DAG) of operators



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

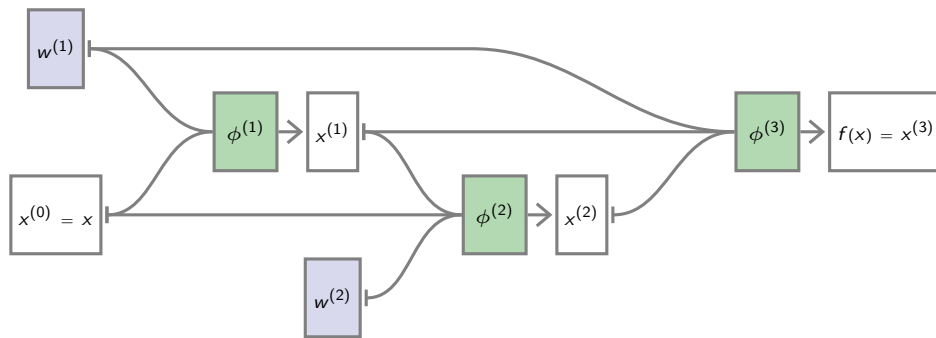
$$\left[\frac{\partial a}{\partial b} \right] = J_\phi = \begin{pmatrix} \frac{\partial a_1}{\partial b_1} & \cdots & \frac{\partial a_1}{\partial b_R} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_Q}{\partial b_1} & \cdots & \frac{\partial a_Q}{\partial b_R} \end{pmatrix}.$$

It does not specify at which point this is computed, but it will always be 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

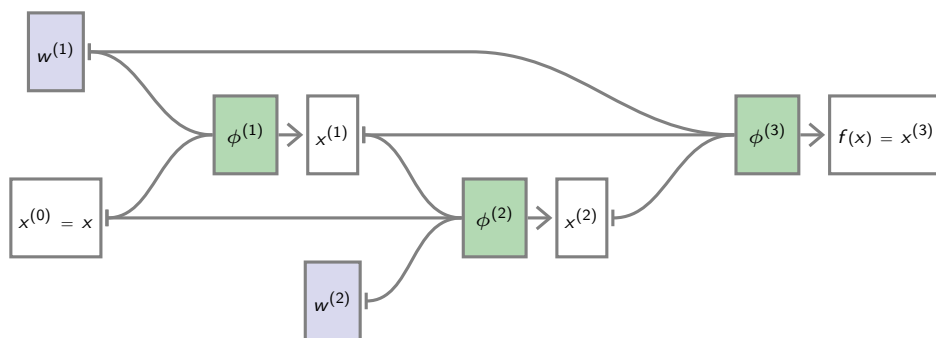
$$\left[\frac{\partial a}{\partial c} \right] = J_{\phi|c} = \begin{pmatrix} \frac{\partial a_1}{\partial c_1} & \cdots & \frac{\partial a_1}{\partial c_S} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_Q}{\partial c_1} & \cdots & \frac{\partial a_Q}{\partial c_S} \end{pmatrix}.$$

Forward pass



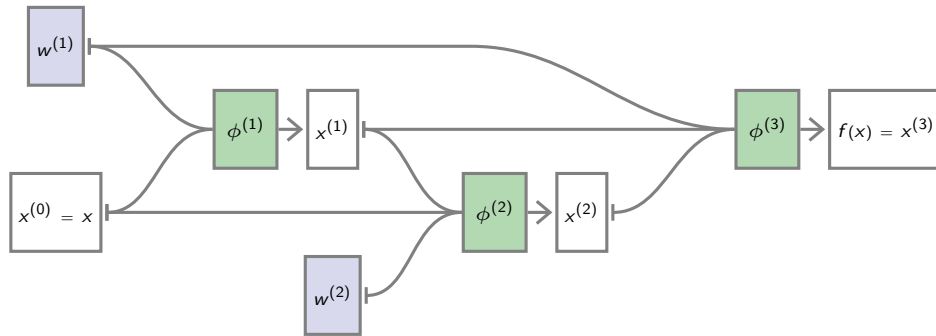
$$\begin{aligned}
 x^{(0)} &= x \\
 x^{(1)} &= \phi^{(1)}(x^{(0)}; w^{(1)}) \\
 x^{(2)} &= \phi^{(2)}(x^{(0)}, x^{(1)}; w^{(2)}) \\
 f(x) = x^{(3)} &= \phi^{(3)}(x^{(1)}, x^{(2)}; w^{(1)})
 \end{aligned}$$

Backward pass, derivatives w.r.t activations



$$\begin{aligned}
 \left[\frac{\partial \ell}{\partial x^{(2)}} \right] &= \left[\frac{\partial x^{(3)}}{\partial x^{(2)}} \right] \left[\frac{\partial \ell}{\partial x^{(3)}} \right] = J_{\phi^{(3)}|x^{(2)}} \left[\frac{\partial \ell}{\partial x^{(3)}} \right] \\
 \left[\frac{\partial \ell}{\partial x^{(1)}} \right] &= \left[\frac{\partial x^{(2)}}{\partial x^{(1)}} \right] \left[\frac{\partial \ell}{\partial x^{(2)}} \right] + \left[\frac{\partial x^{(3)}}{\partial x^{(1)}} \right] \left[\frac{\partial \ell}{\partial x^{(3)}} \right] = J_{\phi^{(2)}|x^{(1)}} \left[\frac{\partial \ell}{\partial x^{(2)}} \right] + J_{\phi^{(3)}|x^{(1)}} \left[\frac{\partial \ell}{\partial x^{(3)}} \right] \\
 \left[\frac{\partial \ell}{\partial x^{(0)}} \right] &= \left[\frac{\partial x^{(1)}}{\partial x^{(0)}} \right] \left[\frac{\partial \ell}{\partial x^{(1)}} \right] + \left[\frac{\partial x^{(2)}}{\partial x^{(0)}} \right] \left[\frac{\partial \ell}{\partial x^{(2)}} \right] = J_{\phi^{(1)}|x^{(0)}} \left[\frac{\partial \ell}{\partial x^{(1)}} \right] + J_{\phi^{(2)}|x^{(0)}} \left[\frac{\partial \ell}{\partial x^{(2)}} \right]
 \end{aligned}$$

Backward pass, derivatives w.r.t parameters



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

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

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

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.

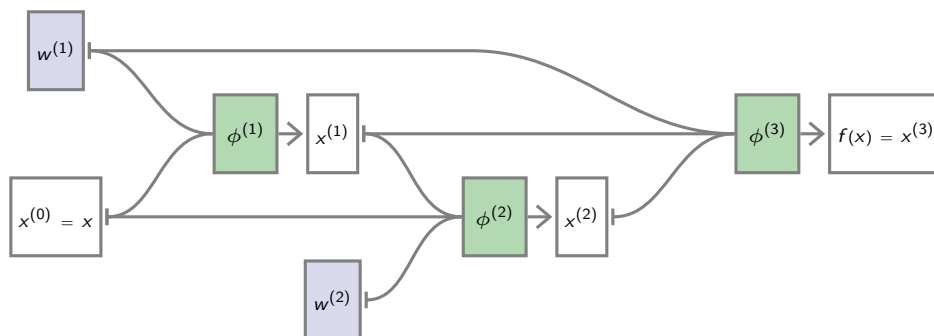
Writing from scratch a large neural network is complex and error-prone.

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.)

In TensorFlow, to run a forward/backward pass on



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

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

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

```
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)
```

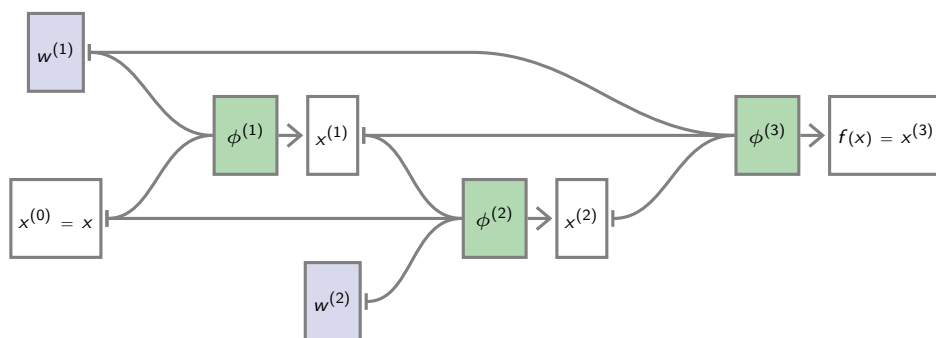
```
gw1, gw2 = tf.gradients(q, [w1, w2])
```

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

Weight sharing

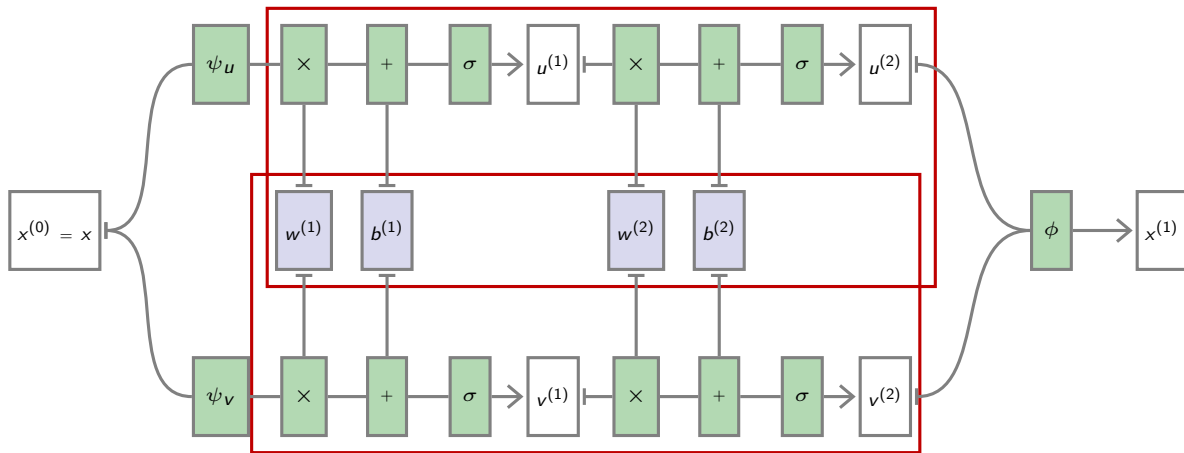
In our generalized DAG formulation, we have in particular implicitly allowed the same parameters to modulate different parts of the processing.

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.



4.2. Autograd

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 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.

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):
  File "<stdin>", line 1, in <module>
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.] )
```

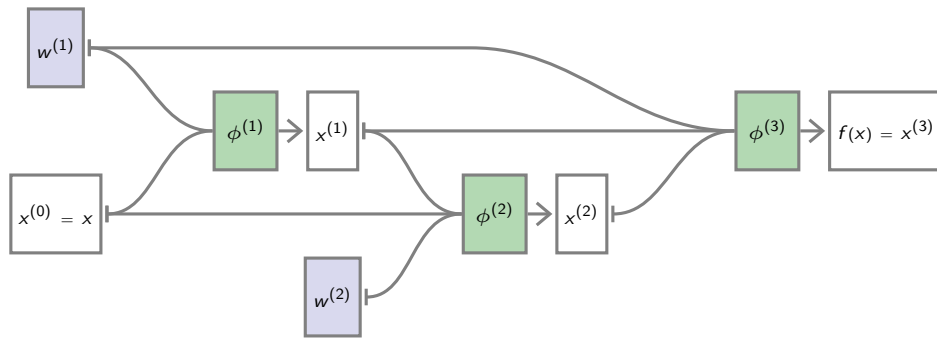
This function is an alternative to `torch.autograd.grad(...)` and standard for training models.



`Tensor.backward()` **accumulates** the gradients in the different 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



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

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

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

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

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

```
q = x3.norm()
```

```
q.backward()
```

The autograd machinery

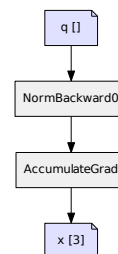
The autograd graph is encoded through the fields `grad_fn` of Tensors, and the fields `next_functions` of Functions.

```
>>> 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 Functions.

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.

```

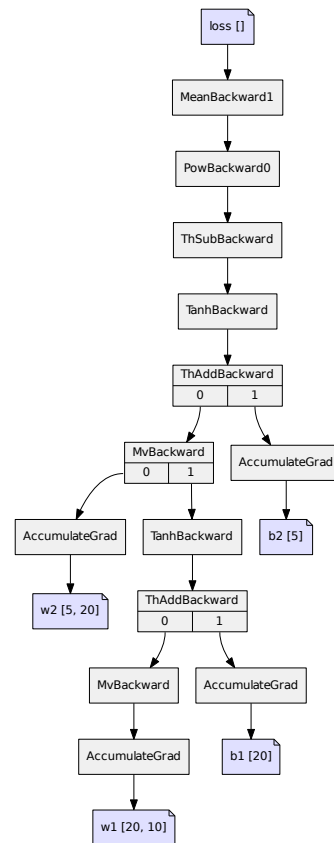
w1 = torch.rand(20, 10).requires_grad_()
b1 = torch.rand(20).requires_grad_()
w2 = torch.rand(5, 20).requires_grad_()
b2 = torch.rand(5).requires_grad_()

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

target = torch.rand(5)

loss = (y - target).pow(2).mean()

```



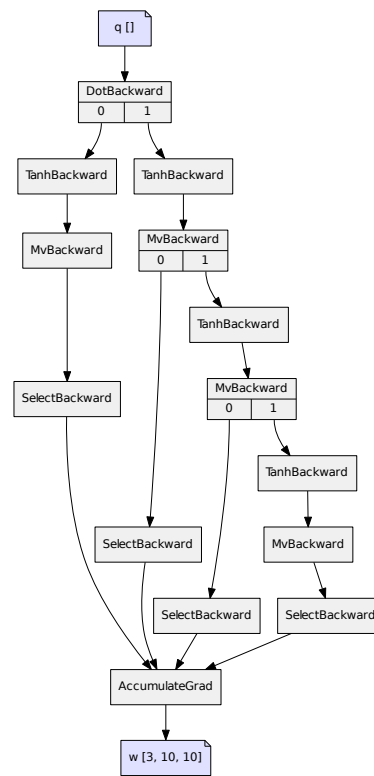
```

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('%0.06f' % a.item(), '%0.06f' % b.item())

prints

0.333333 -0.333333
```

```

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('%.06f' % a.item(), '%.06f' % b.item())

prints

1.000000 -0.000000

```

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

```

>>> x = torch.tensor([ 1., 2., 3. ]).requires_grad_()
>>> phi = x.pow(2).sum()
>>> g1, = torch.autograd.grad(phi, x, create_graph = True)
>>> g1
tensor([2., 4., 6.], grad_fn=<ThMulBackward>)
>>> psi = g1[0].exp() - g1[2].exp()
>>> g2, = torch.autograd.grad(psi, x)
>>> g2
tensor([ 14.7781,  0.0000, -806.8576])

```




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()
>>> l = y.sum()
>>> l.backward()
>>> y = x.sin()
>>> y += 1
>>> l = y.sum()
>>> l.backward()
>>> 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.

4.3. PyTorch modules and batch processing

Elements from `torch.nn.functional` are autograd-compliant functions which compute a result from provided arguments alone. This is usually imported as `F`.

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()`.



Functions and modules from `torch.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 $\mathbb{R}^C \rightarrow \mathbb{R}^D$ 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.

```
torch.nn.functional.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

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

implements a $\mathbb{R}^C \rightarrow \mathbb{R}^D$ 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.empty(523, 10).normal_()
>>> 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

```
torch.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 = torch.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 tensor with `requires_grad` to `True` for target.

```
>>> import torch
>>> f = torch.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 `torch.nn` process samples by batches. This is motivated by the computational speed-up it induces.

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 it is used.

These memory transfers are slower than the computation itself.

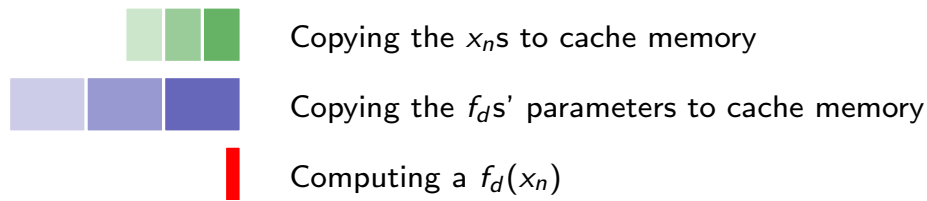
This is the main reason for batch processing: it cuts down to one per module per batch the number of copies of parameters to the cache.

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

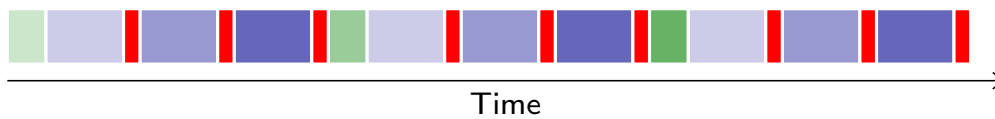
Consider a model composed of three modules

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

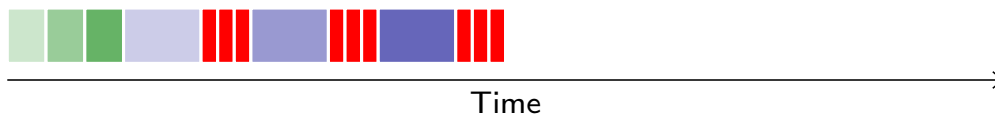
and we want to compute $f(x_1), f(x_2), f(x_3)$.



Processing samples one by one:



Batch processing:



With

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

    for u in range(0, t.size(0)):
        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.empty(2500, 1000).normal_()
w = torch.empty(1500, 1000).normal_()
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

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

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.

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

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

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

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

which gives a tensorial expression for the full batch

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

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

$$\left[\left[\frac{\partial \mathcal{L}}{\partial w^{(l)}} \right] \right] = \left[\left[\frac{\partial \mathcal{L}}{\partial x^{(l)}} \right] \right]^T x^{(l-1)},$$

and

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

4.4. Convolutions

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

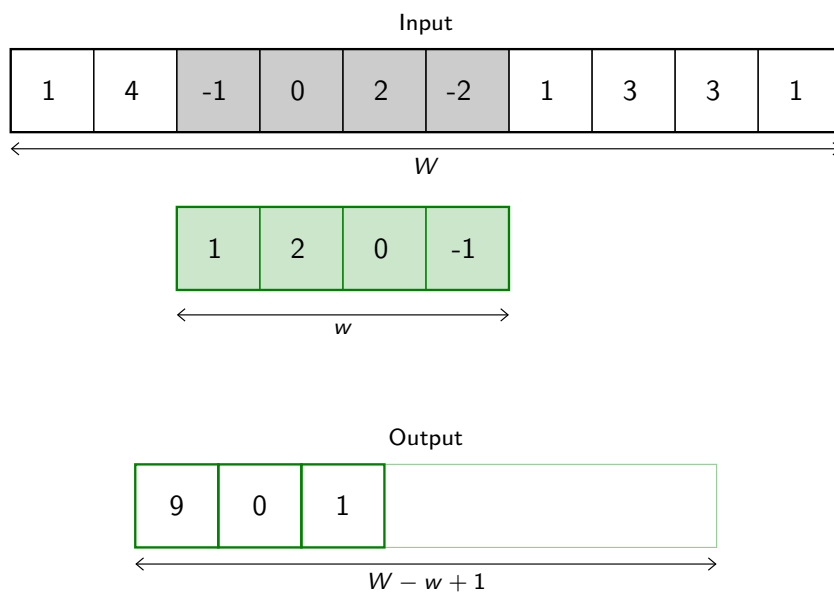
For instance a linear layer taking a 256×256 RGB image as input, and producing an image of same size would require

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

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

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

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



Formally, in 1d, given

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

and a “convolution kernel” (or “filter”) of width w

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

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

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

for instance

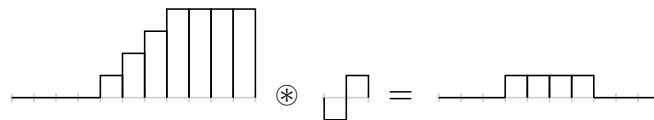
$$(1, 2, 3, 4) \circledast (3, 2) = (3 + 4, 6 + 6, 9 + 8) = (7, 12, 17).$$



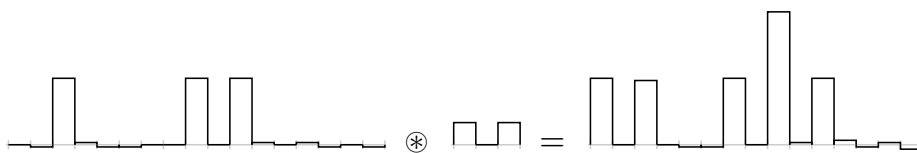
This differs from the usual convolution since the kernel and the signal are both visited in increasing index order.

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.*




Both of these computation examples are indeed “invariant by translation”.

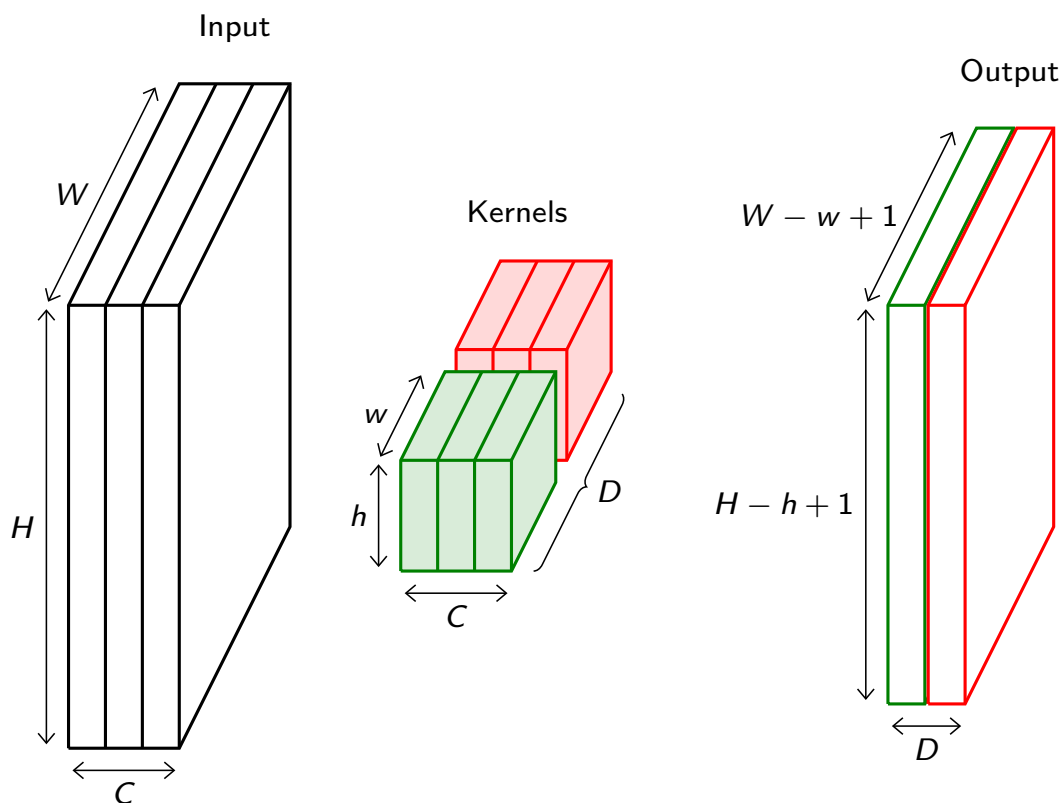
It generalizes naturally to a multi-dimensional input, although specification can become complicated.

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 convolution 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 convolution layer as an **activation map**.

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

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

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

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

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

and the result is of dimension

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

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

Similar functions implement 1d and 3d convolutions.

```
x = mnist_train.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. ] ])

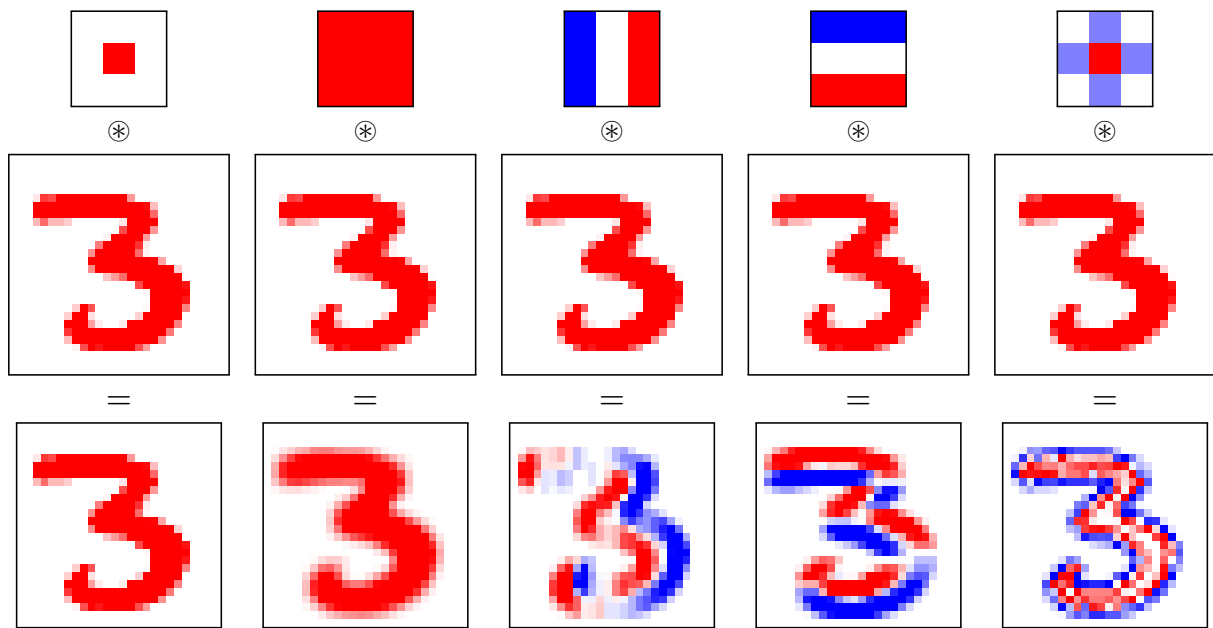
weight[1, 0] = torch.tensor([ [ 1., 1., 1. ],
                              [ 1., 1., 1. ],
                              [ 1., 1., 1. ] ])

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 = torch.nn.functional.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) .

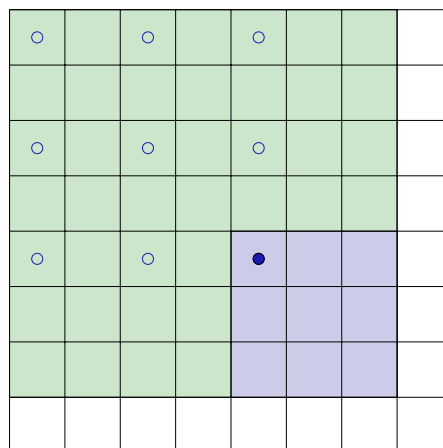
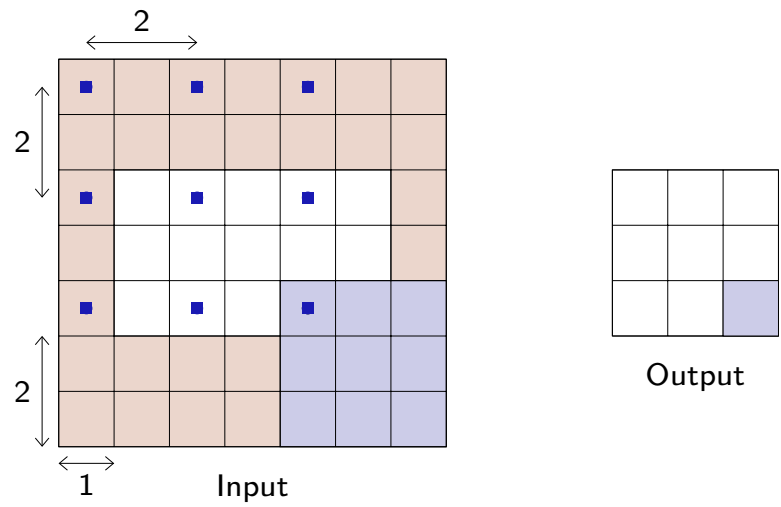
```
>>> f = nn.Conv2d(in_channels = 4, out_channels = 5, kernel_size = (2, 3))
>>> for n, p in f.named_parameters(): print(n, p.size())
...
weight torch.Size([5, 4, 2, 3])
bias torch.Size([5])
>>> x = torch.empty(117, 4, 10, 3).normal_()
>>> y = f(x)
>>> y.size()
torch.Size([117, 5, 9, 1])
```

Padding and stride

Convolutions have two additional standard parameters:

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

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



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

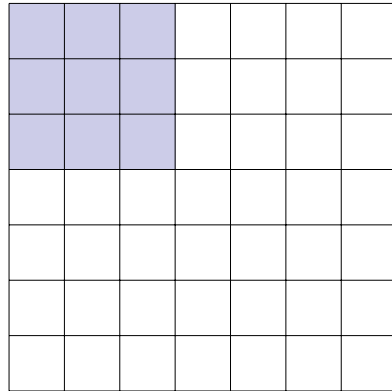
Dilated convolution

Convolution operations admit one more standard parameter that we have not discussed yet: The dilation, which modulates the expansion of the filter support (Yu and Koltun, 2015).

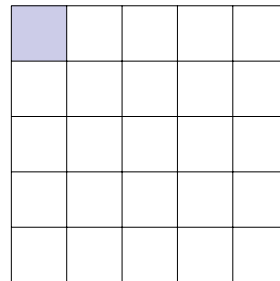
It is 1 for standard convolutions, but can be greater, in which case the resulting operation can be envisioned as a convolution with a regularly sparsified filter.

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

Dilation = 1

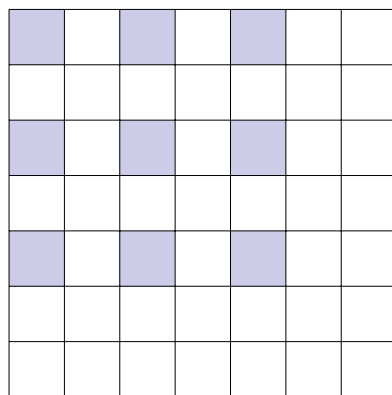


Input

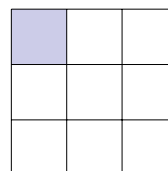


Output

Dilation = 2



Input



Output

A convolution with a 1d 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 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.empty(1, 1, 20, 30).normal_()
>>> 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.

Such networks have the advantage of simplicity:

- non-linear operations are only in the activation function,
- joint operations that combine multiple activations to produce one are only in linear layers.

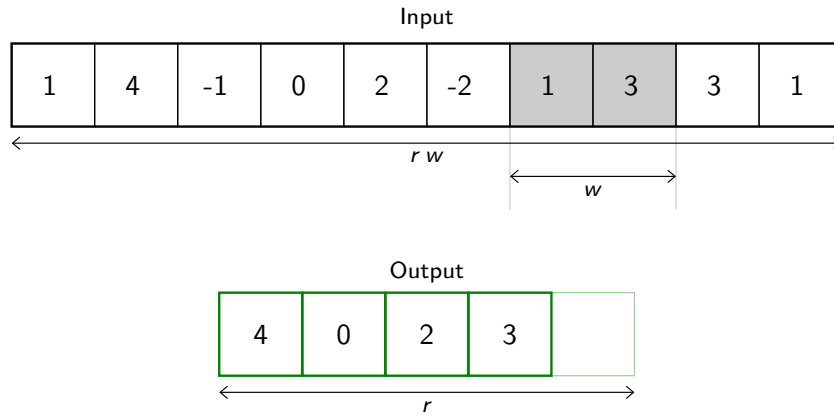
4.5. Pooling

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

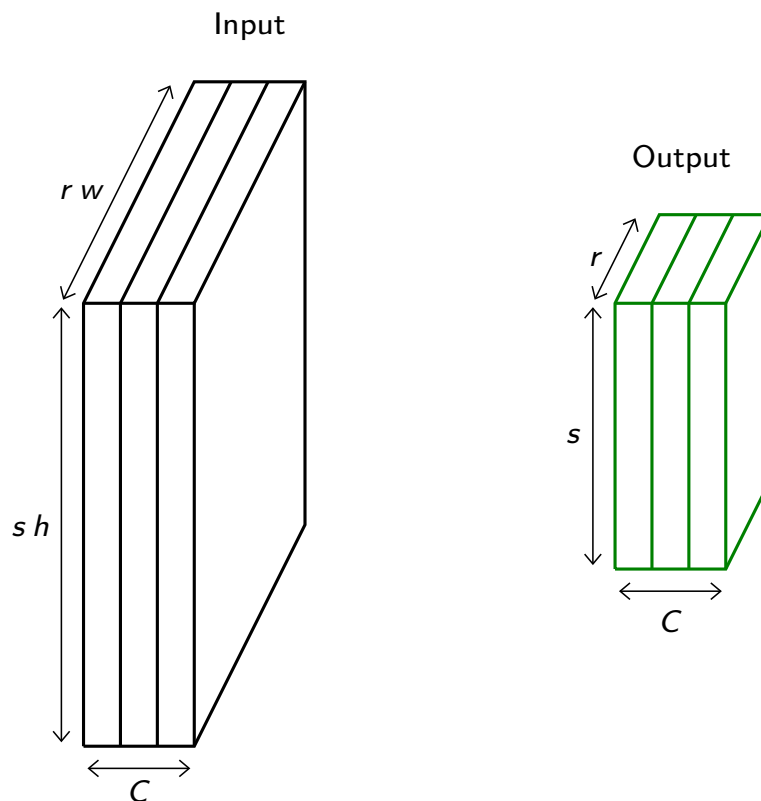
Such an operation aims at grouping several activations into a single “more meaningful” one.

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

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

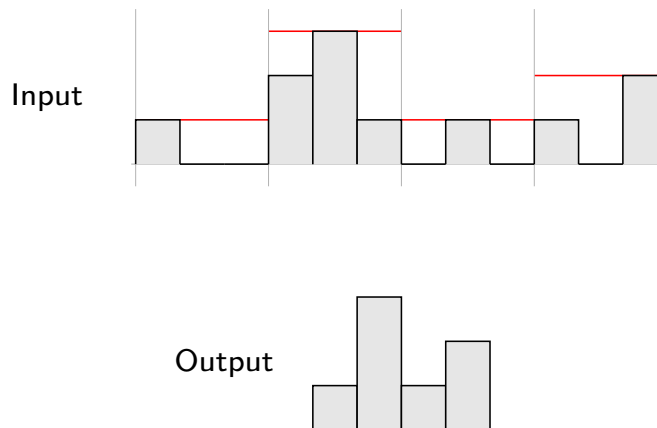


The **average pooling** computes average values per block instead of max values.



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

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



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

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

```
>>> x = torch.empty(2, 2, 6).random_(3)  
>>> x  
tensor([[[ 1.,  2.,  2.,  1.,  2.,  1.],  
         [ 2.,  0.,  0.,  0.,  1.,  0.]],  
        [[ 2.,  0.,  2.,  1.,  1.,  1.],  
         [ 0.,  0.,  0.,  1.,  2.,  1.]])  
>>> F.max_pool2d(x, (1, 2))  
tensor([[[ 2.,  2.,  2.],  
         [ 2.,  0.,  1.]],  
        [[ 2.,  2.,  1.],  
         [ 0.,  1.,  2.]])
```

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

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) .

4.6. Writing a PyTorch module

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
$1 \times 28 \times 28$		
<code>nn.Conv2d(1, 32, kernel_size=5)</code>	$32 \times (5^2 + 1) = 832$	$32 \times 24^2 \times 5^2 = 460,800$
$32 \times 24 \times 24$		
<code>F.max_pool2d(., kernel_size=3)</code>	0	0
$32 \times 8 \times 8$		
<code>F.relu(.)</code>	0	0
$32 \times 8 \times 8$		
<code>nn.Conv2d(32, 64, kernel_size=5)</code>	$64 \times (32 \times 5^2 + 1) = 51,264$	$32 \times 64 \times 4^2 \times 5^2 = 819,200$
$64 \times 4 \times 4$		
<code>F.max_pool2d(., kernel_size=2)</code>	0	0
$64 \times 2 \times 2$		
<code>F.relu(.)</code>	0	0
$64 \times 2 \times 2$		
<code>x.view(-1, 256)</code>	0	0
256		
<code>nn.Linear(256, 200)</code>	$200 \times (256 + 1) = 51,400$	$200 \times 256 = 51,200$
200		
<code>F.relu(.)</code>	0	0
200		
<code>nn.Linear(200, 10)</code>	$10 \times (200 + 1) = 2,010$	$10 \times 200 = 2,000$
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 function 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 practical 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(Net, self).__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.empty(12, 1, 28, 28).normal_()
output = model(input)
print(output.size())
```

prints

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

Also, all Parameters added as class attributes are seen by `Module.parameters()`.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__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 k in model.parameters():
    print(k.size())
```

prints

```
torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([64, 32, 5, 5])
torch.Size([64])
torch.Size([200, 256])
torch.Size([200])
torch.Size([10, 200])
torch.Size([10])
```



Parameters added in dictionaries or arrays are not seen.

```
class Buggy(nn.Module):
    def __init__(self):
        super(Buggy, self).__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

```
torch.Size([123, 456])
torch.Size([32, 1, 5, 5])
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 AnotherNotBuggy(nn.Module):
    def __init__(self):
        super(AnotherNotBuggy, self).__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 = AnotherNotBuggy()
```

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

prints

```
torch.Size([123, 456])
torch.Size([32, 1, 5, 5])
torch.Size([32])
torch.Size([21, 543])
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 Parameters with gradient descent.

5.1. Cross-entropy loss

We can train a model for classification using a regression loss such as the MSE using a “one-hot vector” encoding: given a training set

$$(x_n, y_n) \in \mathbb{R}^D \times \{1, \dots, C\}, \quad n = 1, \dots, N,$$

we would convert the labels into a tensor $z \in \mathbb{R}^{N \times C}$, with

$$\forall n, z_{n,m} = \begin{cases} 1 & \text{if } m = y_n \\ 0 & \text{otherwise.} \end{cases}$$

For instance, with $N = 5$ and $C = 3$, we would have

$$\begin{pmatrix} 2 \\ 1 \\ 1 \\ 3 \\ 2 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

Training can be achieved by matching the output of the model with these binary values in a MSE sense.

However, MSE is justified with a Gaussian noise around a target value that makes sense geometrically. Beside being conceptually wrong for classification, in practice it penalizes responses “too strongly on the right side”.

As we will see, the criterion of choice for classification is the cross-entropy.

We can generalize the logistic regression to a multi-class setup with f_1, \dots, f_C functionals that we interpret as “logit values”

$$P(Y = y | X = x, W = w) = \frac{1}{Z} \exp f_y(x; w) = \frac{\exp f_y(x; w)}{\sum_k \exp f_k(x; w)},$$

from which

$$\begin{aligned} \log \mu_W(w | \mathcal{D} = \mathbf{d}) &= \log \frac{\mu_{\mathcal{D}}(\mathbf{d} | W = w) \mu_W(w)}{\mu_{\mathcal{D}}(\mathbf{d})} \\ &= \log \mu_{\mathcal{D}}(\mathbf{d} | W = w) + \log \mu_W(w) - \log Z \\ &= \sum_n \log \mu_{\mathcal{D}}(x_n, y_n | W = w) + \log \mu_W(w) - \log Z \\ &= \sum_n \log P(Y = y_n | X = x_n, W = w) + \log \mu_W(w) - \log Z' \\ &= \underbrace{\sum_n \log \left(\frac{\exp f_{y_n}(x; w)}{\sum_k \exp f_k(x; w)} \right)}_{\text{Depends on the outputs}} + \underbrace{\log \mu_W(w)}_{\text{Depends on } w} - \log Z'. \end{aligned}$$

If we ignore the penalty on w , it makes sense to minimize the average

$$\mathcal{L}(w) = -\frac{1}{N} \sum_{n=1}^N \log \underbrace{\left(\frac{\exp f_{y_n}(x_n; w)}{\sum_k \exp f_k(x_n; w)} \right)}_{\hat{P}_w(Y=y_n|X=x_n)}.$$

Given two distributions p and q , their **cross-entropy** is defined as

$$\mathbb{H}(p, q) = -\sum_k p(k) \log q(k),$$

with the convention that $0 \log 0 = 0$. So we can re-write

$$\begin{aligned} -\log \left(\frac{\exp f_{y_n}(x_n; w)}{\sum_k \exp f_k(x_n; w)} \right) &= -\log \hat{P}_w(Y = y_n | X = x_n) \\ &= -\sum_k \delta_{y_n}(k) \log \hat{P}_w(Y = k | X = x_n) \\ &= \mathbb{H}(\delta_{y_n}, \hat{P}_w(Y = \cdot | X = x_n)). \end{aligned}$$

So \mathcal{L} above is the average of the cross-entropy between the deterministic “true” posterior δ_{y_n} and the estimated $\hat{P}_w(Y = \cdot | X = x_n)$.

This is precisely the value of `torch.nn.CrossEntropyLoss`.

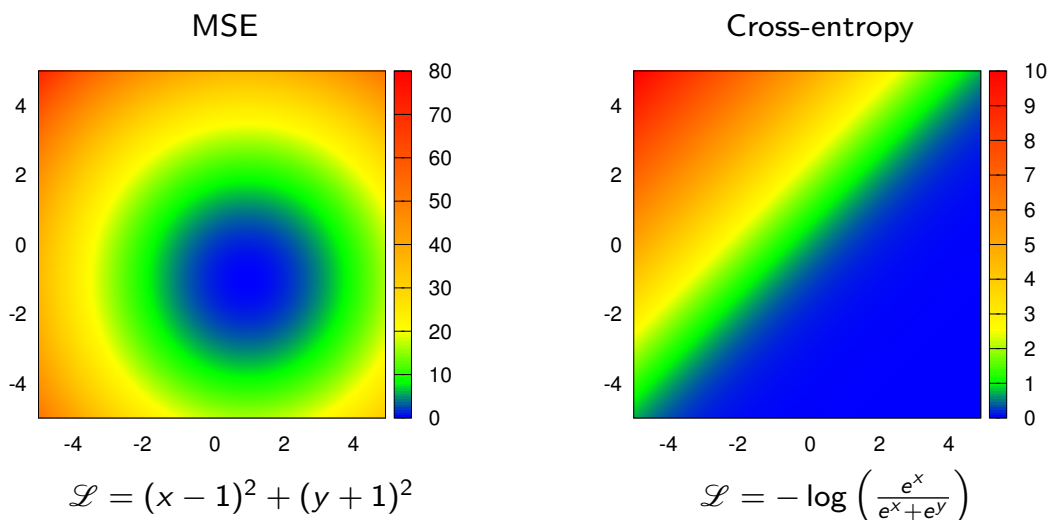
```
>>> f = torch.tensor([[ -1., -3., 4.], [-3., 3., -1.]])
>>> target = torch.tensor([0, 1])
>>> criterion = torch.nn.CrossEntropyLoss()
>>> criterion(f, target)
tensor(2.5141)
```

and indeed

$$-\frac{1}{2} \left(\log \frac{e^{-1}}{e^{-1} + e^{-3} + e^4} + \log \frac{e^3}{e^{-3} + e^3 + e^{-1}} \right) \simeq 2.5141.$$

The range of values is 0 for perfectly classified samples, $\log(C)$ if the posterior is uniform, and up to $+\infty$ if the posterior distribution is “worst” than uniform.

Let's consider the loss for a single sample in a two-class problem, with a predictor with two output values. The x axis here is the activation of the correct output unit, and the y axis is the activation of the other one.



MSE incorrectly penalizes outputs which are perfectly valid for prediction, contrary to cross-entropy.

The cross-entropy loss can be seen as the composition of a “log soft-max” to normalize the score into logs of probabilities

$$(\alpha_1, \dots, \alpha_C) \mapsto \left(\log \frac{\exp \alpha_1}{\sum_k \exp \alpha_k}, \dots, \log \frac{\exp \alpha_C}{\sum_k \exp \alpha_k} \right),$$

which can be done with the `torch.nn.LogSoftmax` module, and a read-out of the normalized score of the correct class

$$\mathcal{L}(w) = -\frac{1}{N} \sum_{n=1}^N f_{y_n}(x_n; w),$$

which is implemented by the `torch.nn.NLLLoss` criterion.

```
>>> f = torch.tensor([[ -1., -3., 4.], [-3., 3., -1.]])
>>> target = torch.tensor([0, 1])
>>> model = nn.LogSoftmax(dim = 1)
>>> criterion = torch.nn.NLLLoss()
>>> criterion(model(f), target)
tensor(2.5141)
```

Hence, if a network should compute log-probabilities, it may have a `torch.nn.LogSoftmax` final layer, and be trained with `torch.nn.NLLLoss`.

The mapping

$$(\alpha_1, \dots, \alpha_C) \mapsto \left(\frac{\exp \alpha_1}{\sum_k \exp \alpha_k}, \dots, \frac{\exp \alpha_C}{\sum_k \exp \alpha_k} \right)$$

is called soft-max since it computes a “soft arg-max Boolean label.”

```
>>> y = torch.tensor([[ -10., -10., 10., -5. ],
...                  [ 3., 0., 0., 0. ],
...                  [ 1., 2., 3., 4. ]])
>>> f = torch.nn.Softmax(1)
>>> f(y)
tensor([[ 2.0612e-09,  2.0612e-09,  1.0000e+00,  3.0590e-07],
        [ 8.7005e-01,  4.3317e-02,  4.3317e-02,  4.3317e-02],
        [ 3.2059e-02,  8.7144e-02,  2.3688e-01,  6.4391e-01]])
```

PyTorch provides many other criteria, among which

- `torch.nn.MSELoss`
- `torch.nn.CrossEntropyLoss`
- `torch.nn.NLLLoss`
- `torch.nn.L1Loss`
- `torch.nn.NLLLoss2d`
- `torch.nn.MultiMarginLoss`

5.2. Stochastic gradient descent

To minimize a loss of the form

$$\mathcal{L}(w) = \sum_{n=1}^N \underbrace{\ell(f(x_n; w), y_n)}_{\ell_n(w)}$$

the standard gradient-descent algorithm update has the form

$$w_{t+1} = w_t - \eta \nabla \mathcal{L}(w_t).$$

A straight-forward implementation would be

```
for e in range(nb_epochs):
    output = model(train_input)
    loss = criterion(output, train_target)

    model.zero_grad()
    loss.backward()
    with torch.no_grad():
        for p in model.parameters(): p -= eta * p.grad
```

However, the memory footprint is proportional to the full set size. This can be mitigated by summing the gradient through “mini-batches”:

```
for e in range(nb_epochs):
    model.zero_grad()

    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input[b:b+batch_size])
        loss = criterion(output, train_target[b:b+batch_size])
        loss.backward()

    with torch.no_grad():
        for p in model.parameters(): p -= eta * p.grad
```

While it makes sense in principle to compute the gradient exactly, in practice:

- It takes time to compute (more exactly **all our time!**).
- It is an empirical estimation of an hidden quantity, and any partial sum is also an unbiased estimate, although of greater variance.
- It is computed incrementally

$$\nabla \mathcal{L}(w_t) = \sum_{n=1}^N \nabla \ell_n(w_t),$$

and when we compute ℓ_n , we have already computed $\ell_1, \dots, \ell_{n-1}$, and we could have a better estimate of w^* than w_t .

To illustrate how partial sums are good estimates, consider an ideal case where the training set is the same set of $M \ll N$ samples replicated K times. Then

$$\begin{aligned} \mathcal{L}(w) &= \sum_{n=1}^N \ell(f(x_n; w), y_n) \\ &= \sum_{k=1}^K \sum_{m=1}^M \ell(f(x_m; w), y_m) \\ &= K \sum_{m=1}^M \ell(f(x_m; w), y_m). \end{aligned}$$

So instead of summing over all the samples and moving by η , we can visit only $M = N/K$ samples and move by $K\eta$, which would cut the computation by K .

Although this is an ideal case, there is redundancy in practice that results in similar behaviors.

The **stochastic gradient descent** consists of updating the parameters w_t after every sample

$$w_{t+1} = w_t - \eta \nabla \ell_{n(t)}(w_t).$$

However this does not benefit from the speed-up of batch-processing.

The **mini-batch stochastic gradient descent** is the standard procedure for deep learning. It consists of visiting the samples in “mini-batches”, each of a few tens of samples, and updating the parameters each time.

$$w_{t+1} = w_t - \eta \sum_{b=1}^B \nabla \ell_{n(t,b)}(w_t).$$

The order $n(t, b)$ to visit the samples can either be sequential, or uniform sampling, usually without replacement.

The stochastic behavior of this procedure helps evade local minima.

So our exact gradient descent with mini-batches

```
for e in range(nb_epochs):
    model.zero_grad()

    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input[b:b+batch_size])
        loss = criterion(output, train_target[b:b+batch_size])
        loss.backward()

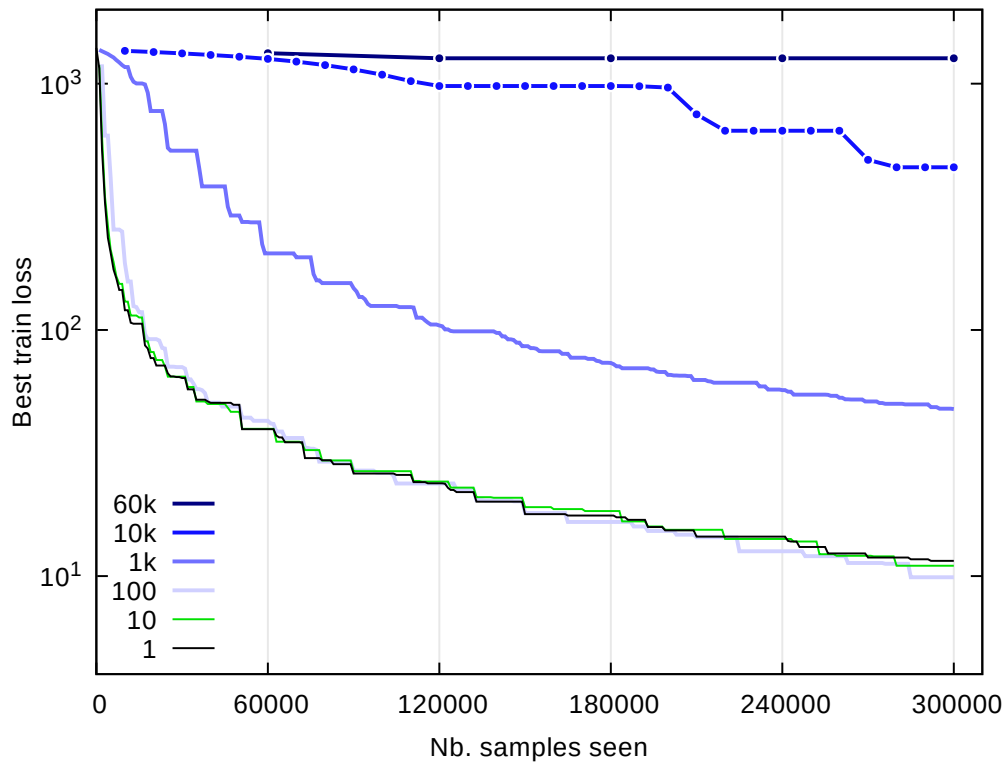
    with torch.no_grad():
        for p in model.parameters(): p -= eta * p.grad
```

can be modified into the mini-batch stochastic gradient descent as follows:

```
for e in range(nb_epochs):
    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input[b:b+batch_size])
        loss = criterion(output, train_target[b:b+batch_size])

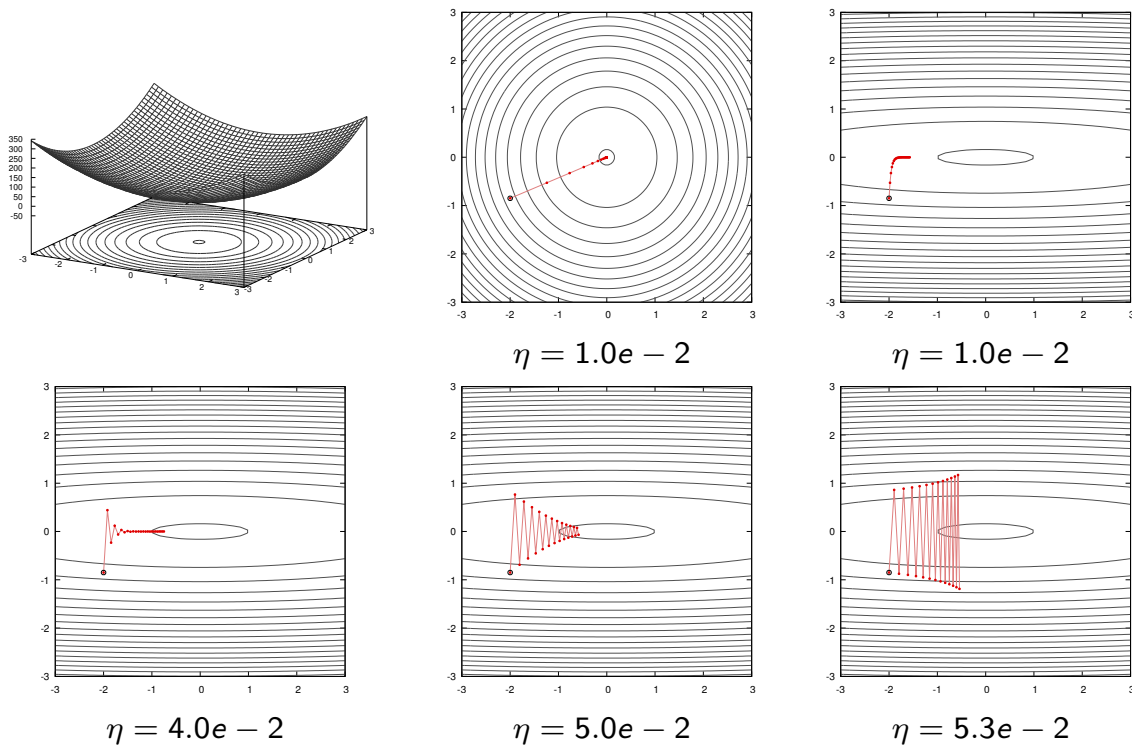
    model.zero_grad()
    loss.backward()
    with torch.no_grad():
        for p in model.parameters(): p -= eta * p.grad
```

Mini-batch size and loss reduction (MNIST)



Limitation of the gradient descent

The gradient descent method makes a strong assumption about the magnitude of the “local curvature” to fix the step size, and about its isotropy so that the same step size makes sense in all directions.



Some optimization methods leverage higher-order moments, in particular second order to use a more accurate local model of the functional to optimize.

However for a fixed computational budget, the complexity of these methods reduces the total number of iterations, and the eventual optimization is worst.

Deep-learning generally relies on a smarter use of the gradient, using statistics over its past values to make a “smarter step” with the current one.

Momentum and moment estimation

The “vanilla” mini-batch stochastic gradient descent (SGD) consists of

$$w_{t+1} = w_t - \eta g_t,$$

where

$$g_t = \sum_{b=1}^B \nabla \ell_{n(t,b)}(w_t)$$

is the gradient summed over a mini-batch.

The first improvement is the use of a “momentum” to add inertia in the choice of the step direction

$$u_t = \gamma u_{t-1} + \eta g_t$$

$$w_{t+1} = w_t - u_t.$$

(Rumelhart et al., 1986)

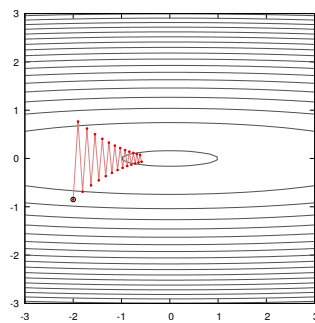
With $\gamma = 0$, this is the same as vanilla SGD.

With $\gamma > 0$, this update has three nice properties:

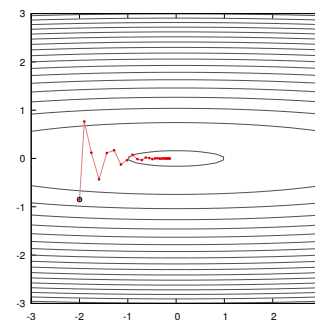
- it can “go through” local barriers,
- it accelerates if the gradient does not change much:

$$(u = \gamma u + \eta g) \Rightarrow \left(u = \frac{\eta}{1 - \gamma} g \right),$$

- it dampens oscillations in narrow valleys.



$\eta = 5.0e - 2, \gamma = 0$



$\eta = 5.0e - 2, \gamma = 0.5$

Another class of methods exploits the statistics over the previous steps to compensate for the anisotropy of the mapping.

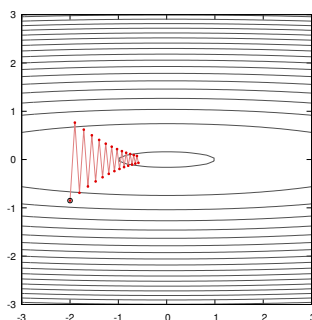
The Adam algorithm uses moving averages of each coordinate and its square to rescale each coordinate separately.

The update rule is, **on each coordinate separately**

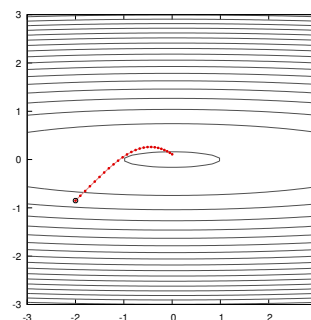
$$\begin{aligned}
 m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
 \hat{m}_t &= \frac{m_t}{1 - \beta_1} \\
 v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\
 \hat{v}_t &= \frac{v_t}{1 - \beta_2} \\
 w_{t+1} &= w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
 \end{aligned}$$

(Kingma and Ba, 2014)

This can be seen as a combination of momentum, with \hat{m}_t , and a per-coordinate re-scaling with \hat{v}_t .



$$\eta = 5.0e - 2$$



Adam,
 $\beta_1 = 0.9, \beta_2 = 0.999,$
 $\epsilon = 1e - 8, \eta = 1.0e - 1$

These two core strategies have been used in multiple incarnations:

- Nesterov's accelerated gradient,
- Adagrad,
- Adadelta,
- RMSprop,
- AdaMax,
- Nadam ...

5.3. PyTorch optimizers

The PyTorch module `torch.optim` provides many optimizers.

An optimizer has an internal state to keep quantities such as moving averages, and operates on an iterator over Parameters.

- Values specific to the optimizer can be specified to its constructor, and
- its `step` method updates the internal state according to the `grad` attributes of the Parameters, and updates the latter according to the internal state.

We implemented the standard SGD as follows

```
for e in range(nb_epochs):
    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input[b:b+batch_size])
        loss = criterion(output, train_target[b:b+batch_size])
        model.zero_grad()
        loss.backward()
        with torch.no_grad():
            for p in model.parameters(): p -= eta * p.grad
```

which can be re-written with the `torch.optim` package as

```
optimizer = torch.optim.SGD(model.parameters(), lr = eta)
```

```
for e in range(nb_epochs):
    for b in range(0, train_input.size(0), batch_size):
        output = model(train_input[b:b+batch_size])
        loss = criterion(output, train_target[b:b+batch_size])
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

We have at our disposal many variants of the SGD:

- `torch.optim.SGD` (momentum, and Nesterov's algorithm),
- `torch.optim.Adam`
- `torch.optim.Adadelta`
- `torch.optim.Adagrad`
- `torch.optim.RMSprop`
- `torch.optim.LBFGS`
- ...

An optimizer can also operate on several iterators, each corresponding to a group of Parameters that should be handled similarly. For instance, different layers may have different learning rates or momentums.

So to use Adam, with its default setting, we just have to replace in our example

```
optimizer = optim.SGD(model.parameters(), lr = eta)
```

with

```
optimizer = optim.Adam(model.parameters(), lr = eta)
```



The learning rate may have to be different if the functional was not properly scaled.

An example putting all this together

We now have the tools to build and train a deep network:

- fully connected layers,
- convolutional layers,
- pooling layers,
- ReLU.

And we have the tools to optimize it:

- Loss,
- back-propagation,
- stochastic gradient descent.

The only piece missing is the policy to initialize the parameters.

PyTorch initializes parameters with default rules when modules are created. They normalize weights according to the layer sizes (Glorot and Bengio, 2010) and behave usually very well. We will come back to this.

```

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__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))
        x = F.relu(F.max_pool2d(self.conv2(x), kernel_size = 2))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

```

```

train_set = torchvision.datasets.MNIST('./data/mnist/',
                                     train = True, download = True)
train_input = train_set.train_data.view(-1, 1, 28, 28).float()
train_target = train_set.train_labels

lr, nb_epochs, batch_size = 1e-1, 10, 100

model = Net()

optimizer = torch.optim.SGD(model.parameters(), lr = lr)
criterion = nn.CrossEntropyLoss()

model.to(device)
criterion.to(device)
train_input, train_target = train_input.to(device), train_target.to(device)

mu, std = train_input.mean(), train_input.std()
train_input.sub_(mu).div_(std)

for e in range(nb_epochs):
    for input, target in zip(train_input.split(batch_size),
                             train_target.split(batch_size)):
        output = model(input)
        loss = criterion(output, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

```


5.6. Architecture choice and training protocol

Choosing the network structure is a difficult exercise. **There is no silver bullet.**

- Re-use something “well known, that works”, or at least start from there,
- split feature extraction / inference (although this is debatable),
- modulate the capacity until it overfits a small subset, but does not overfit / underfit the full set,
- capacity increases with more layers, more channels, larger receptive fields, or more units,
- regularization to reduce the capacity or induce sparsity,
- identify common paths for siamese-like,
- identify what path(s) or sub-parts need more/less capacity,
- use prior knowledge about the “scale of meaningful context” to size filters / combinations of filters (e.g. knowing the size of objects in a scene, the max duration of a sound snippet that matters),
- grid-search all the variations that come to mind (and hopefully have farms of GPUs to do so).

We will re-visit this list with additional regularization / normalization methods.

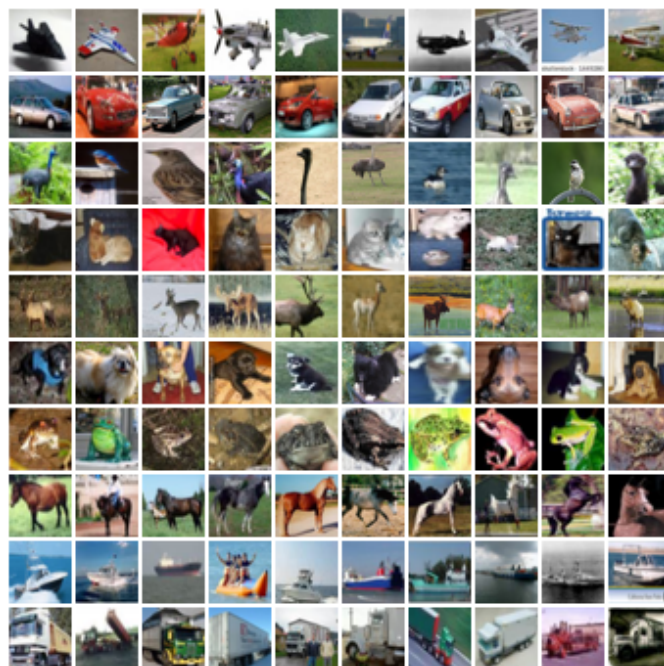
Regarding the learning rate, for training to succeed it has to

- reduce the loss quickly \Rightarrow large learning rate,
- not be trapped in a bad minimum \Rightarrow large learning rate,
- not bounce around in narrow valleys \Rightarrow small learning rate, and
- not oscillate around a minimum \Rightarrow small learning rate.

These constraints lead to a general policy of using a **larger step size first, and a smaller one in the end.**

The practical strategy is to look at the losses and error rates across epochs and pick a learning rate and learning rate adaptation. For instance by reducing it at discrete pre-defined steps, or with a geometric decay.

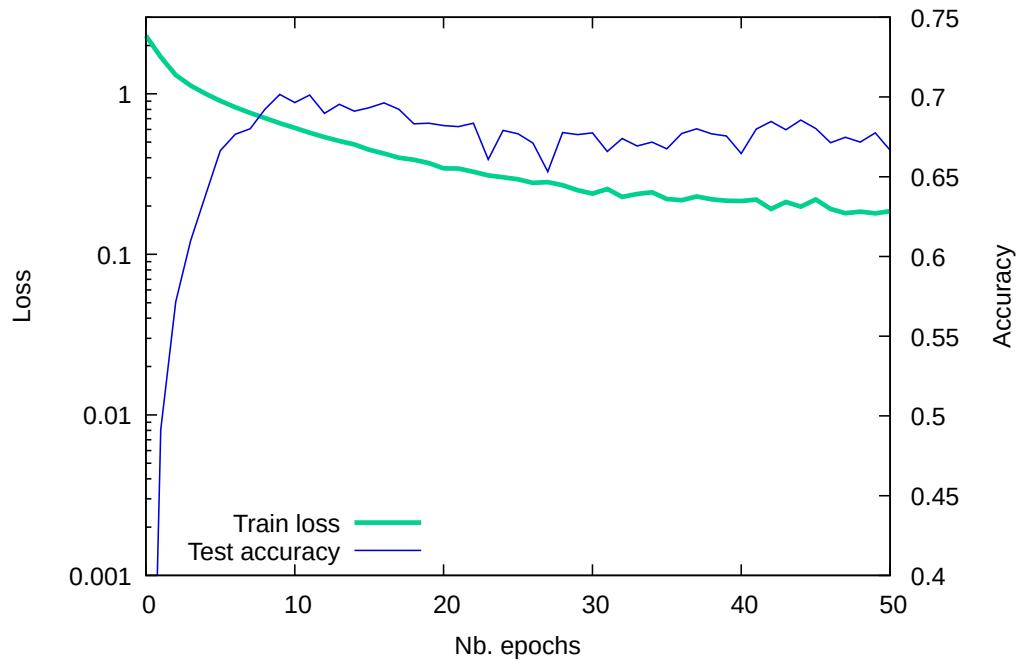
CIFAR10 data-set



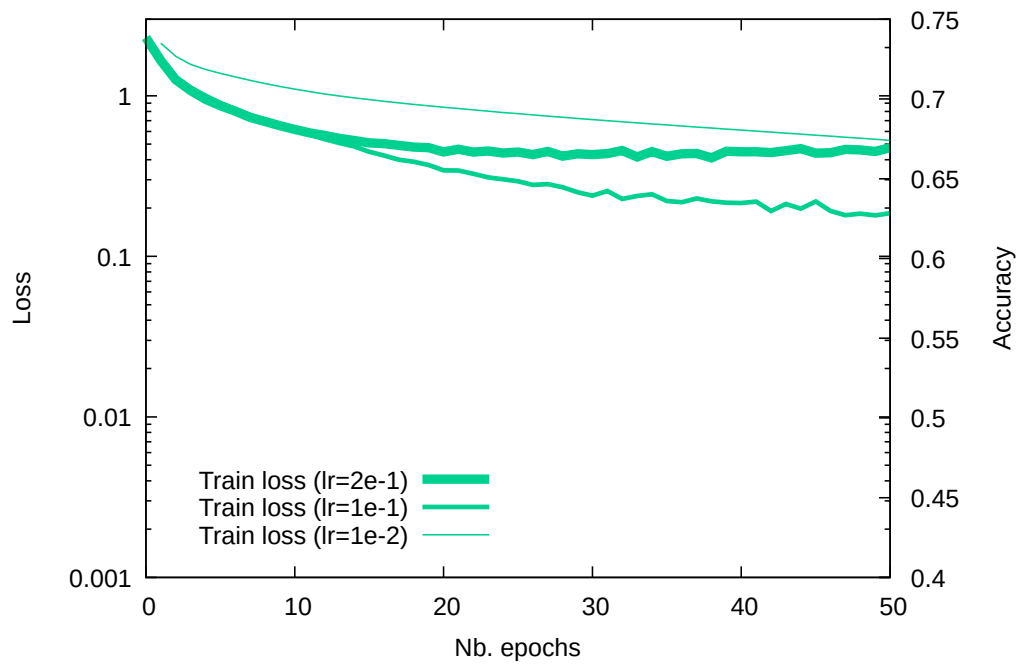
32×32 color images, 50,000 train samples, 10,000 test samples.

(Krizhevsky, 2009, chap. 3)

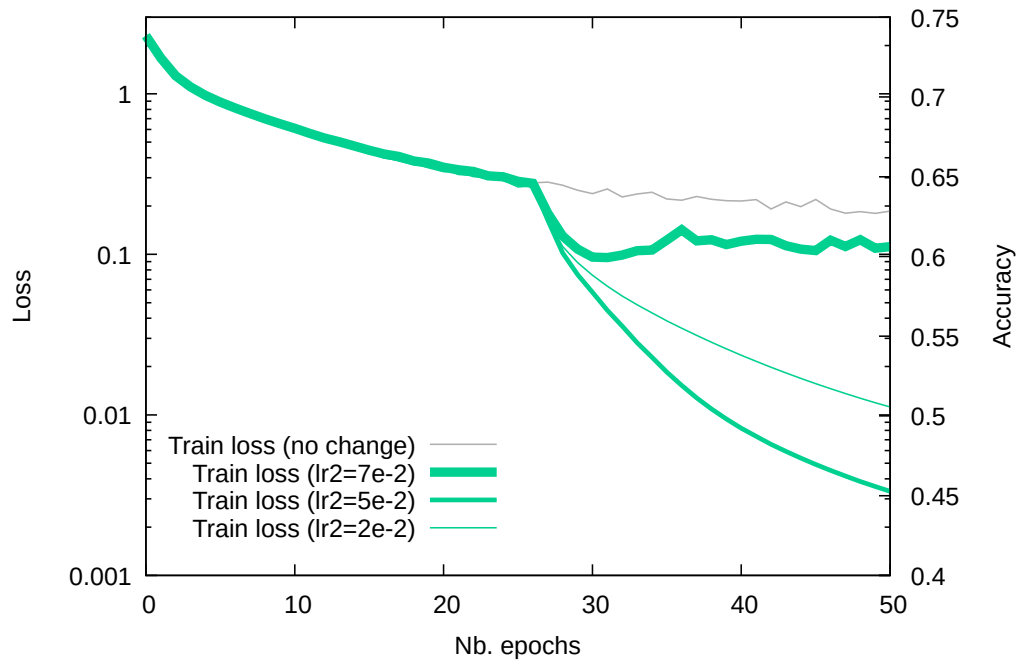
Small convnet on CIFAR10, cross-entropy, batch size 100, $\eta = 1e - 1$.



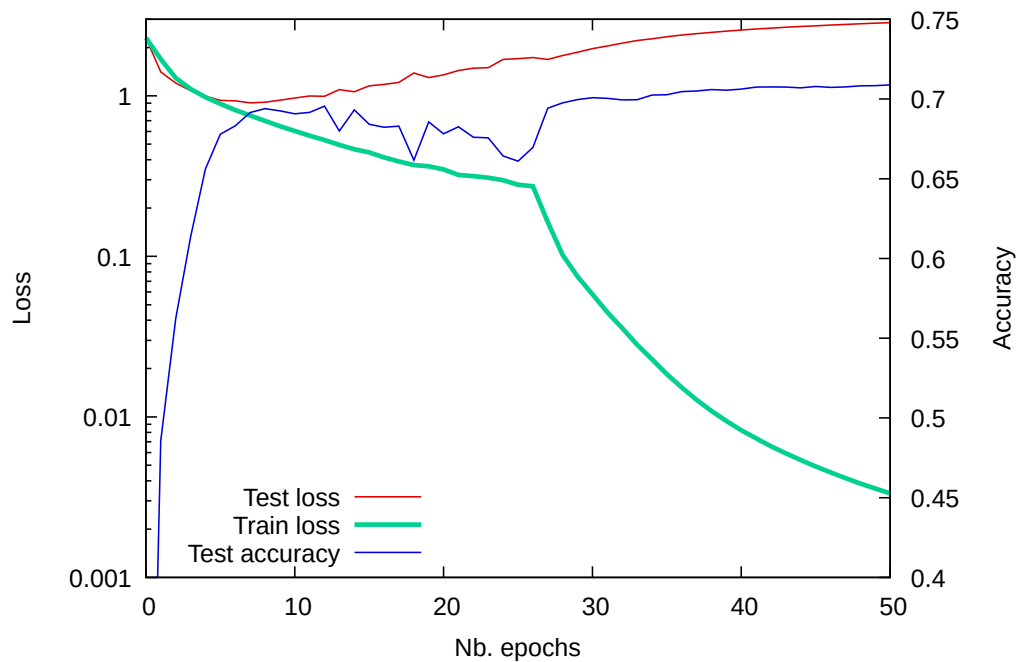
Small convnet on CIFAR10, cross-entropy, batch size 100



Using $\eta = 1e - 1$ for 25 epochs, then reducing it.



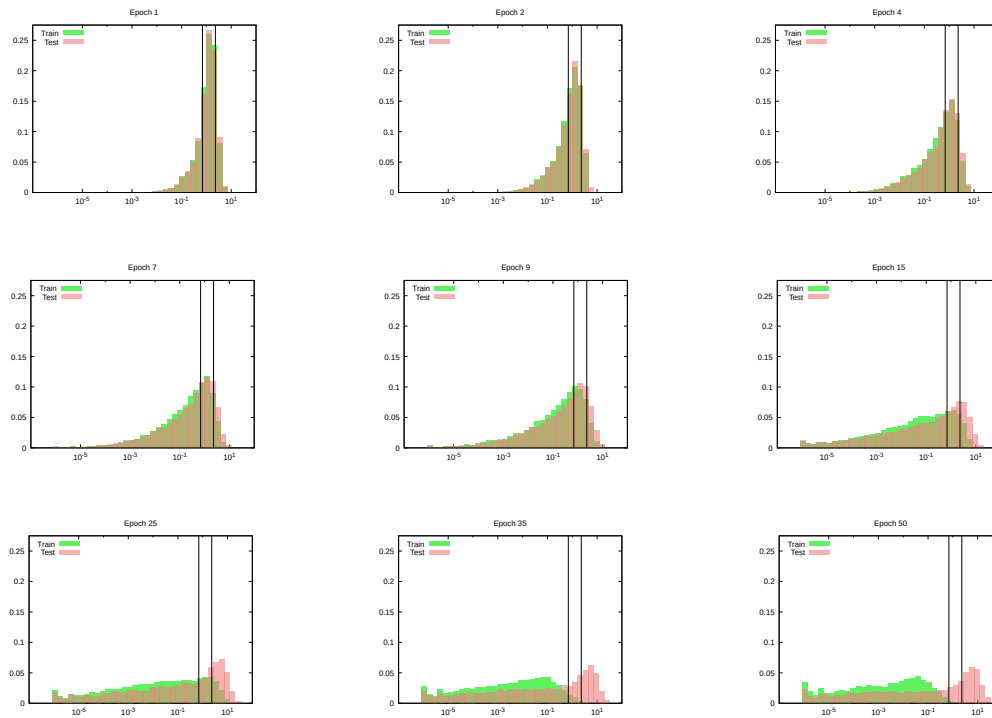
While the test error still goes down, the test loss may increase, as it gets even worse on misclassified examples, and decreases less on the ones getting fixed.



We can plot the train and test distributions of the per-sample loss

$$\ell = -\log \left(\frac{\exp(f_Y(X; w))}{\sum_k \exp(f_k(X; w))} \right)$$

through epochs to visualize the over-fitting.



7.2. Networks for image classification

Image classification, standard convnets

The most standard networks for image classification are the LeNet family (LeCun et al., 1998), and its modern extensions, among which AlexNet (Krizhevsky et al., 2012) and VGGNet (Simonyan and Zisserman, 2014).

They share a common structure of several convolutional layers seen as a feature extractor, followed by fully connected layers seen as a classifier.

The performance of AlexNet was a wake-up call for the computer vision community, as it vastly out-performed other methods in spite of its simplicity.

Recent advances rely on moving from standard convolutional layers to local complex architectures to reduce the model size.

`torchvision.models` provides a collection of reference networks for computer vision, e.g.:

```
import torchvision
alexnet = torchvision.models.alexnet()
```

The trained models can be obtained by passing `pretrained = True` to the constructor(s). This may involve an heavy download given there size.



The networks from PyTorch listed in the coming slides may differ slightly from the reference papers which introduced them historically.

LeNet5 (LeCun et al., 1989). 10 classes, input $1 \times 28 \times 28$.

```
(features): Sequential (
  (0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (1): ReLU (inplace)
  (2): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (4): ReLU (inplace)
  (5): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
)

(classifier): Sequential (
  (0): Linear (256 -> 120)
  (1): ReLU (inplace)
  (2): Linear (120 -> 84)
  (3): ReLU (inplace)
  (4): Linear (84 -> 10)
)
```

Alexnet (Krizhevsky et al., 2012). 1,000 classes, input $3 \times 224 \times 224$.

```
(features): Sequential (  
  (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))  
  (1): ReLU (inplace)  
  (2): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))  
  (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))  
  (4): ReLU (inplace)  
  (5): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))  
  (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (7): ReLU (inplace)  
  (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (9): ReLU (inplace)  
  (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (11): ReLU (inplace)  
  (12): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))  
)  
  
(classifier): Sequential (  
  (0): Dropout (p = 0.5)  
  (1): Linear (9216 -> 4096)  
  (2): ReLU (inplace)  
  (3): Dropout (p = 0.5)  
  (4): Linear (4096 -> 4096)  
  (5): ReLU (inplace)  
  (6): Linear (4096 -> 1000)  
)
```

Krizhevsky et al. used **data augmentation** during training to reduce over-fitting.

They generated 2,048 samples from every original training example through two classes of transformations:

- crop a 224×224 image at a random position in the original 256×256 , and randomly reflect it horizontally,
- apply a color transformation using a PCA model of the color distribution.

During test the prediction is averaged over five random crops and their horizontal reflections.

VGGNet19 (Simonyan and Zisserman, 2014). 1,000 classes, input $3 \times 224 \times 224$. 16 convolutional layers + 3 fully connected layers.

```
(features): Sequential (  
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (1): ReLU (inplace)  
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (3): ReLU (inplace)  
  (4): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))  
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (6): ReLU (inplace)  
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (8): ReLU (inplace)  
  (9): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))  
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (11): ReLU (inplace)  
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (13): ReLU (inplace)  
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (15): ReLU (inplace)  
  (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (17): ReLU (inplace)  
  (18): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))  
  (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (20): ReLU (inplace)  
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (22): ReLU (inplace)  
  (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (24): ReLU (inplace)  
  (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
  (26): ReLU (inplace)  
  (27): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))  
/.../
```

VGGNet19 (cont.)

```
(classifier): Sequential (  
  (0): Linear (25088 -> 4096)  
  (1): ReLU (inplace)  
  (2): Dropout (p = 0.5)  
  (3): Linear (4096 -> 4096)  
  (4): ReLU (inplace)  
  (5): Dropout (p = 0.5)  
  (6): Linear (4096 -> 1000)  
)
```

We can illustrate the convenience of these pre-trained models on a simple image-classification problem.



To be sure this picture did not appear in the training data, it was not taken from the web.

```
import PIL, torch, torchvision

# Imagenet class names
class_names = eval(open('imagenet1000_clsidx_to_human.txt', 'r').read())

# Load and normalize the image
to_tensor = torchvision.transforms.ToTensor()
img = to_tensor(PIL.Image.open('example_images/blacklab.jpg'))
img = img.view(1, img.size(0), img.size(1), img.size(2))
img = 0.5 + 0.5 * (img - img.mean()) / img.std()

# Load and evaluate the network
alexnet = torchvision.models.alexnet(pretrained = True)
alexnet.eval()

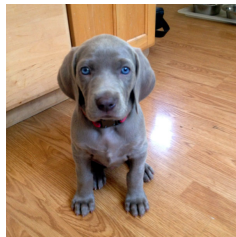
output = alexnet(img)

# Prints the classes
scores, indexes = output.view(-1).sort(descending = True)

for k in range(15):
    print('%02f' % scores[k].item(), class_names[indexes[k].item()])
```



- 12.26 Weimaraner
- 10.95 Chesapeake Bay retriever
- 10.87 Labrador retriever
- 10.10 Staffordshire bullterrier, Staffordshire bull terrier
- 9.55 flat-coated retriever
- 9.40 Italian greyhound
- 9.31 American Staffordshire terrier, Staffordshire terrier, American pit bull terrier, pit bull terrier
- 9.12 Great Dane
- 8.94 German short-haired pointer
- 8.53 Doberman, Doberman pinscher
- 8.35 Rottweiler
- 8.25 kelpie
- 8.24 barrow, garden cart, lawn cart, wheelbarrow
- 8.12 bucket, pail
- 8.07 soccer ball



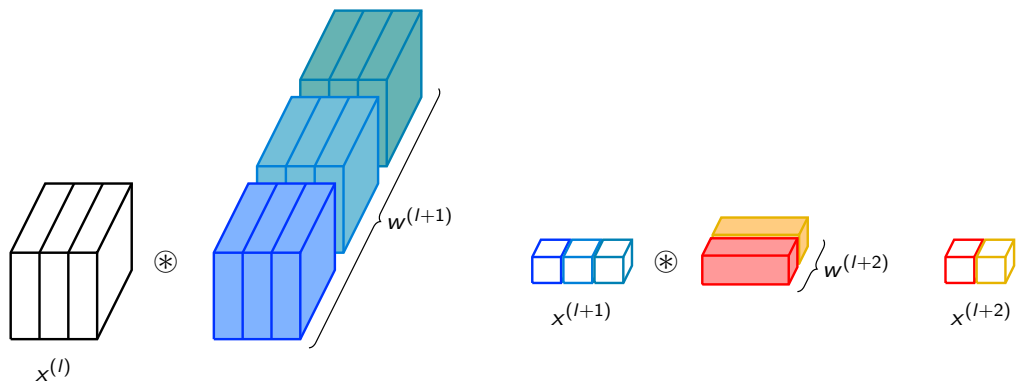
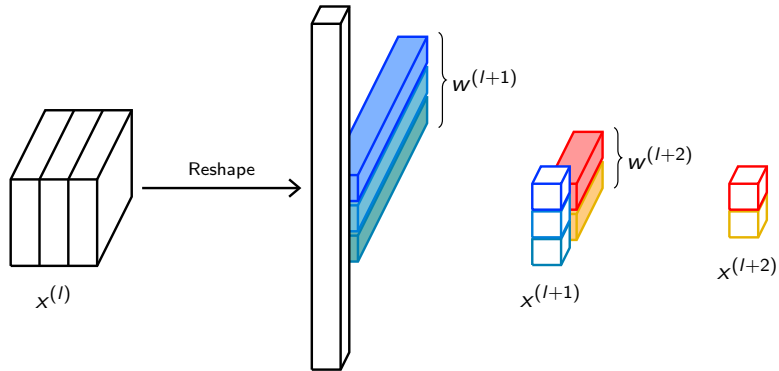
Weimaraner



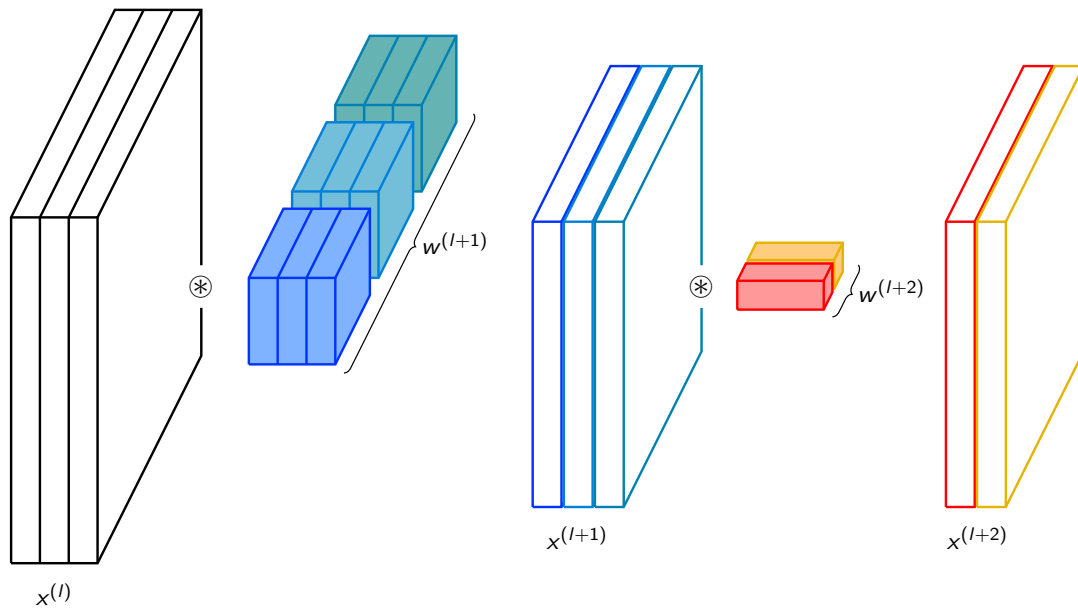
Chesapeake Bay retriever

Fully convolutional networks

In many applications, standard convolutional networks are made **fully convolutional** by converting their fully connected layers to convolutional ones.



This “convolutionization” does not change anything if the input size is such that the output has a single spatial cell, but it **fully re-uses computation to get a prediction at multiple locations** when the input is larger.



We can write a routine that transforms a series of layers from a standard convnets to make it fully convolutional:

```
def convolutionize(layers, input_size):
    result_layers = []
    x = torch.zeros((1, ) + input_size)

    for m in layers:
        if isinstance(m, torch.nn.Linear):
            n = torch.nn.Conv2d(in_channels = x.size(1),
                                out_channels = m.weight.size(0),
                                kernel_size = (x.size(2), x.size(3)))

            with torch.no_grad():
                n.weight.view(-1).copy_(m.weight.view(-1))
                n.bias.view(-1).copy_(m.bias.view(-1))
            m = n

        result_layers.append(m)
        x = m(x)

    return result_layers
```



This function makes the [strong and disputable] assumption that only `nn.Linear` has to be converted.

To apply this to AlexNet

```
model = torchvision.models.alexnet(pretrained = True)
print(model)

layers = list(model.features) + list(model.classifier)

model = nn.Sequential(*convolutionize(layers, (3, 224, 224)))
print(model)
```

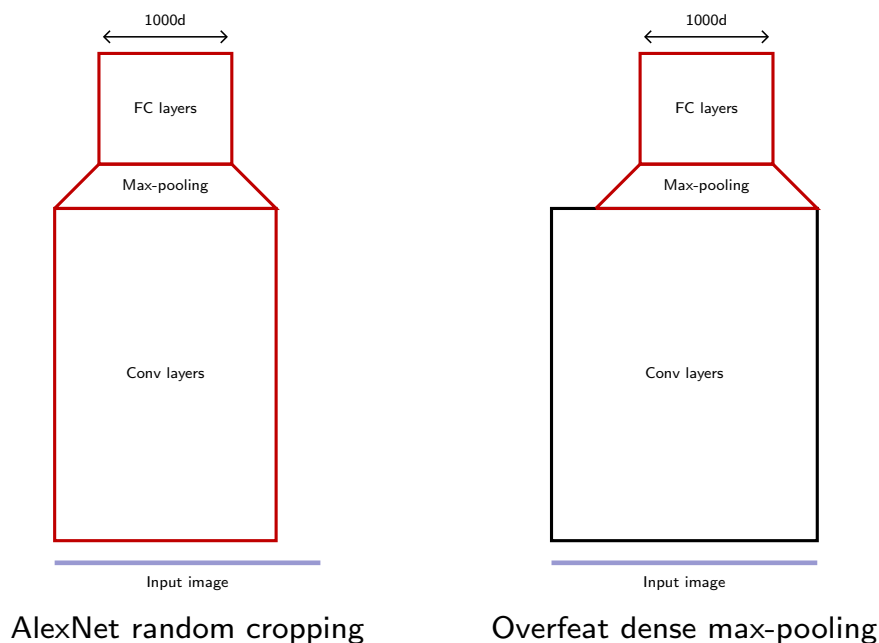
```
AlexNet (
  (features): Sequential (
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU (inplace)
    (2): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU (inplace)
    (5): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU (inplace)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU (inplace)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU (inplace)
    (12): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
  )
  (classifier): Sequential (
    (0): Dropout (p = 0.5)
    (1): Linear (9216 -> 4096)
    (2): ReLU (inplace)
    (3): Dropout (p = 0.5)
    (4): Linear (4096 -> 4096)
    (5): ReLU (inplace)
    (6): Linear (4096 -> 1000)
  )
)
```

```

Sequential (
  (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
  (1): ReLU (inplace)
  (2): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
  (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (4): ReLU (inplace)
  (5): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
  (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (7): ReLU (inplace)
  (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (9): ReLU (inplace)
  (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU (inplace)
  (12): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
  (13): Dropout (p = 0.5)
  (14): Conv2d(256, 4096, kernel_size=(6, 6), stride=(1, 1))
  (15): ReLU (inplace)
  (16): Dropout (p = 0.5)
  (17): Conv2d(4096, 4096, kernel_size=(1, 1), stride=(1, 1))
  (18): ReLU (inplace)
  (19): Conv2d(4096, 1000, kernel_size=(1, 1), stride=(1, 1))
)

```

In their “overfeat” approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.



Doing so, they could afford parsing the scene at 6 scales to improve invariance.

This “convolutionization” has a practical consequence, as we can now re-use classification networks for **dense prediction** without re-training.

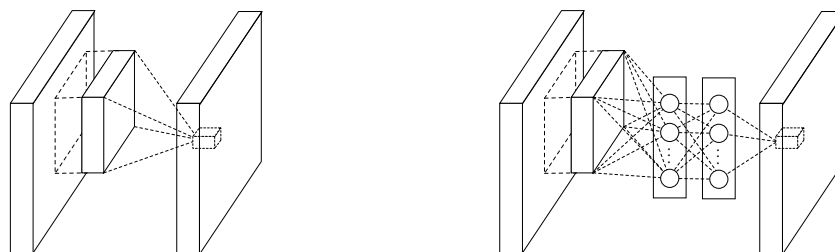
Also, and maybe more importantly, it blurs the conceptual boundary between “features” and “classifier” and leads to an intuitive understanding of convnet activations as gradually transitioning from appearance to semantic.

In the case of a large output prediction map, a final prediction can be obtained by averaging the final output map channel-wise.

If the last layer is linear, the averaging can be done first, as in the residual networks (He et al., 2015).

Image classification, network in network

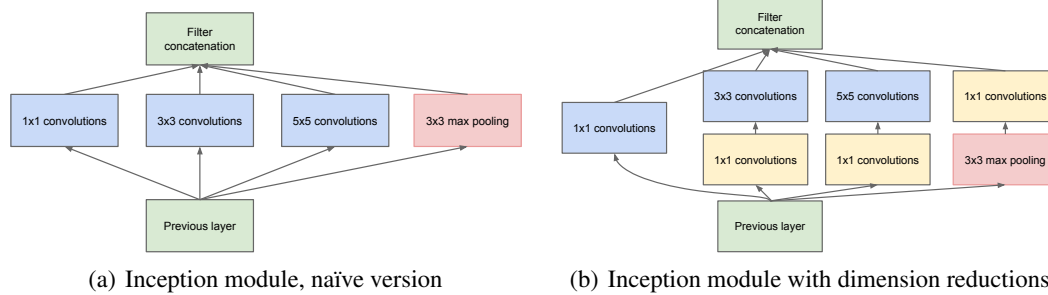
Lin et al. (2013) re-interpreted a convolution filter as a one-layer perceptron, and extended it with an “MLP convolution” (aka “network in network”) to improve the capacity vs. parameter ratio.



(Lin et al., 2013)

As for the fully convolutional networks, such local MLPs can be implemented with 1×1 convolutions.

The same notion was generalized by Szegedy et al. (2015) for their GoogLeNet, through the use of module combining convolutions at multiple scales to let the optimal ones be picked during training.

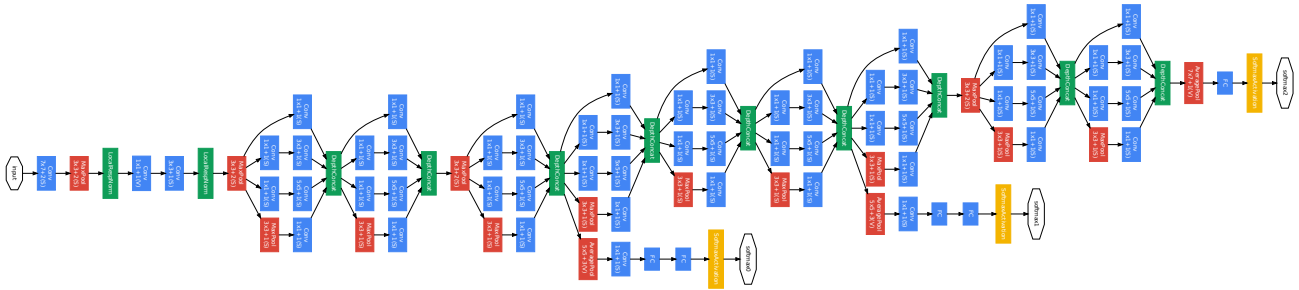


(Szegedy et al., 2015)

Szegedy et al. (2015) also introduce the idea of **auxiliary classifiers** to help the propagation of the gradient in the early layers.

This is motivated by the reasonable performance of shallow networks that indicates early layers already encode informative and invariant features.

The resulting GoogLeNet has 12 times less parameters than AlexNet and is more accurate on ILSVRC14 (Szegedy et al., 2015).



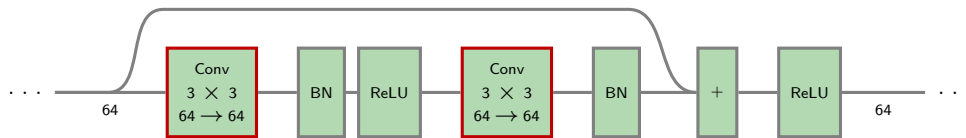
(Szegedy et al., 2015)

It was later extended with techniques we are going to see in the next slides: batch-normalization (Ioffe and Szegedy, 2015) and pass-through à la resnet (Szegedy et al., 2016).

Image classification, residual networks

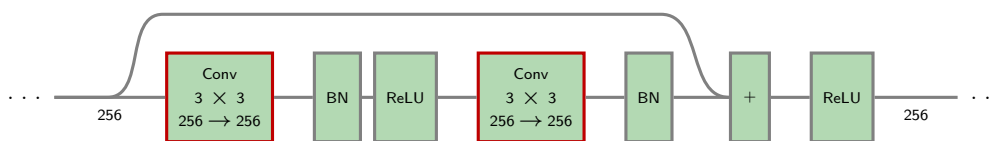
We already saw the structure of the residual networks and how well they perform on CIFAR10 (He et al., 2015).

The default residual block proposed by He et al. is of the form



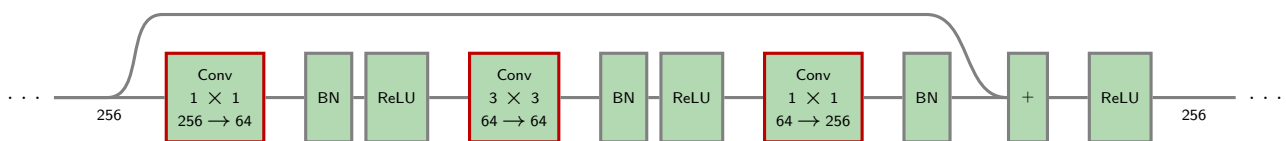
and as such requires $2 \times (3 \times 3 \times 64 + 1) \times 64 \simeq 73k$ parameters.

To apply the same architecture to ImageNet, more channels are required, e.g.



However, such a block requires $2 \times (3 \times 3 \times 256 + 1) \times 256 \simeq 1.2m$ parameters.

They mitigated that requirement with what they call a **bottleneck** block:



$256 \times 64 + (3 \times 3 \times 64 + 1) \times 64 + 64 \times 256 \simeq 70k$ parameters.

The encoding pushed between blocks is high-dimensional, but the “contextual reasoning” in convolutional layers is done on a simpler feature representation.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3\times 3, 64 \\ 3\times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times 3, 64 \\ 3\times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1, 64 \\ 3\times 3, 64 \\ 1\times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1, 64 \\ 3\times 3, 64 \\ 1\times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1, 64 \\ 3\times 3, 64 \\ 1\times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3\times 3, 128 \\ 3\times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times 3, 128 \\ 3\times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times 1, 128 \\ 3\times 3, 128 \\ 1\times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times 1, 128 \\ 3\times 3, 128 \\ 1\times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times 1, 128 \\ 3\times 3, 128 \\ 1\times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3\times 3, 256 \\ 3\times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times 3, 256 \\ 3\times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times 1, 256 \\ 3\times 3, 256 \\ 1\times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times 1, 256 \\ 3\times 3, 256 \\ 1\times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1\times 1, 256 \\ 3\times 3, 256 \\ 1\times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3\times 3, 512 \\ 3\times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times 3, 512 \\ 3\times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1, 512 \\ 3\times 3, 512 \\ 1\times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1, 512 \\ 3\times 3, 512 \\ 1\times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1, 512 \\ 3\times 3, 512 \\ 1\times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10 ⁹	3.6×10 ⁹	3.8×10 ⁹	7.6×10 ⁹	11.3×10 ⁹

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

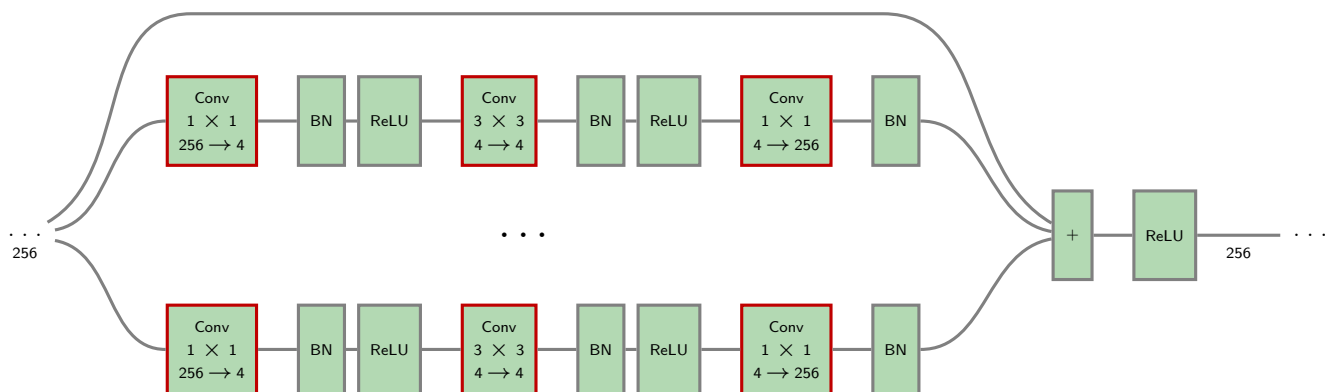
(He et al., 2015)

method	top-5 err. (test)
VGG [41] (ILSVRC'14)	7.32
GoogLeNet [44] (ILSVRC'14)	6.66
VGG [41] (v5)	6.8
PReLU-net [13]	4.94
BN-inception [16]	4.82
ResNet (ILSVRC'15)	3.57

Table 5. Error rates (%) of **ensembles**. The top-5 error is on the test set of ImageNet and reported by the test server.

(He et al., 2015)

This was extended to the ResNeXt architecture by Xie et al. (2016), with blocks with similar number of parameters, but split into 32 “aggregated” pathways.



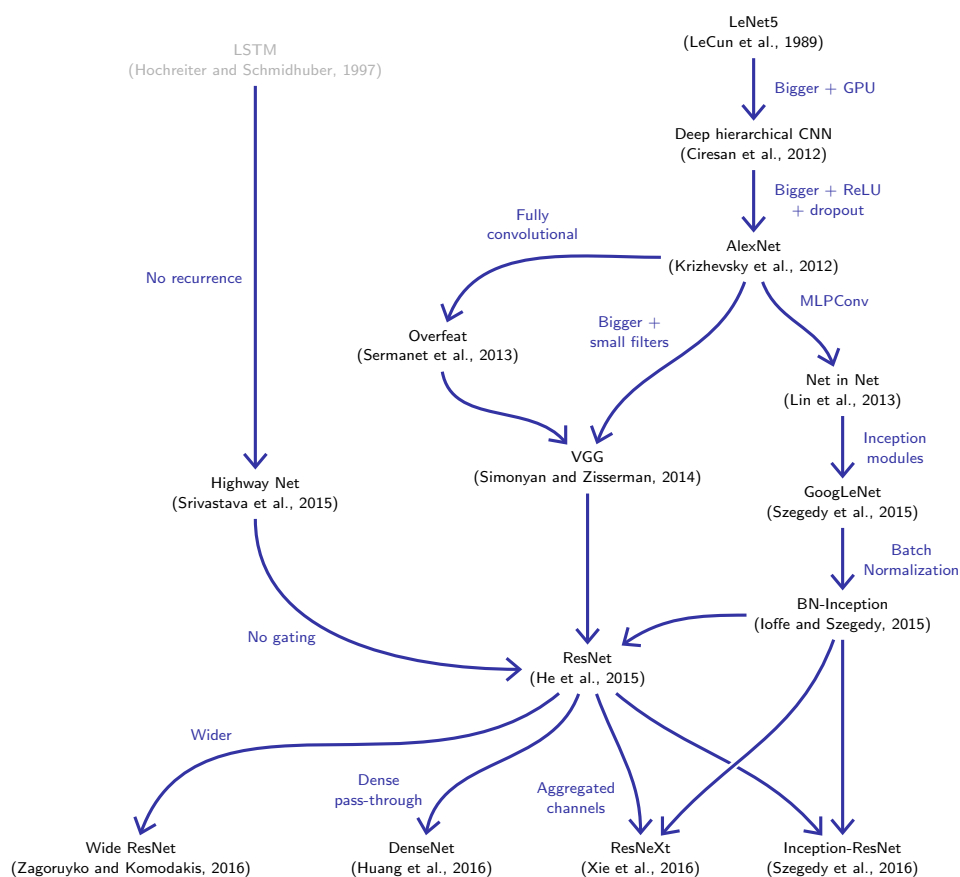
When equalizing the number of parameters, this architecture performs better than a standard resnet.

Image classification, summary

To summarize roughly the evolution of convnets for image classification:

- standard ones are extensions of LeNet5,
- everybody loves ReLU,
- state-of-the-art networks have 100s of channels and 10s of layers,
- they can (should?) be fully convolutional,
- pass-through connections allow deeper “residual” nets,
- bottleneck local structures reduce the number of parameters,
- aggregated pathways reduce the number of parameters.

Image classification networks



References

- D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. CoRR, abs/1202.2745, 2012.
- X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In International Conference on Artificial Intelligence and Statistics (AISTATS), 2010.
- K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. CoRR, abs/1512.03385, 2015.
- S. Hochreiter and J. Schmidhuber. Long short-term memory. Neural Computation, 9(8): 1735–1780, 1997.
- G. Huang, Z. Liu, K. Weinberger, and L. van der Maaten. Densely connected convolutional networks. CoRR, abs/1608.06993, 2016.
- S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International Conference on Machine Learning (ICML), 2015.
- D. Kingma and J. Ba. Adam: A method for stochastic optimization. CoRR, abs/1412.6980, 2014.
- A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, Department of Computer Science, University of Toronto, 2009.
- A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In Neural Information Processing Systems (NIPS), 2012.
- Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. Neural Computation, 1(4):541–551, 1989.
- Y. leCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278–2324, 1998.
- M. Lin, Q. Chen, and S. Yan. Network in network. CoRR, abs/1312.4400, 2013.
- A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In Proceedings of the NIPS Autodiff workshop, 2017.
- D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. Nature, 323(9):533–536, 1986.
- P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. CoRR, abs/1312.6229, 2013.
- K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. CoRR, abs/1409.1556, 2014.
- R. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. CoRR, abs/1505.00387, 2015.
- C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In Conference on Computer Vision and Pattern Recognition (CVPR), 2015.
- C. Szegedy, S. Ioffe, and V. Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. CoRR, abs/1602.07261, 2016.
- S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. CoRR, abs/1611.05431.pdf, 2016.

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

S. Zagoruyko and N. Komodakis. Wide residual networks. CoRR, abs/1605.07146, 2016.