Tensors, autograd, deep and generative models

François Fleuret

Lecture 1 – Tensors, autograd, convolutions, SGD.

Lecture 2 – Benefits of depth, technics for deeper architectures.

Lecture 3 – Under the hood, generative models.

MATH+/BMS – Introduction to Deep Learning

1. Tensors, autograd, convolutions, SGD

François Fleuret

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

Mon Aug 19 06:08:27 UTC 2019

idiap RESEARCH INSTITUTE

EPFL

From neural networks to deep learning

Deep learning is built on a natural generalization of a neural network: **a graph of tensor operators**, taking advantage of

- the chain rule (aka "back-propagation"),
- stochastic gradient decent,
- convolutions,
- parallel operations on GPUs.

This does not differ much from networks from the 90s

1949 – Donald Hebb proposes the Hebbian Learning principle.

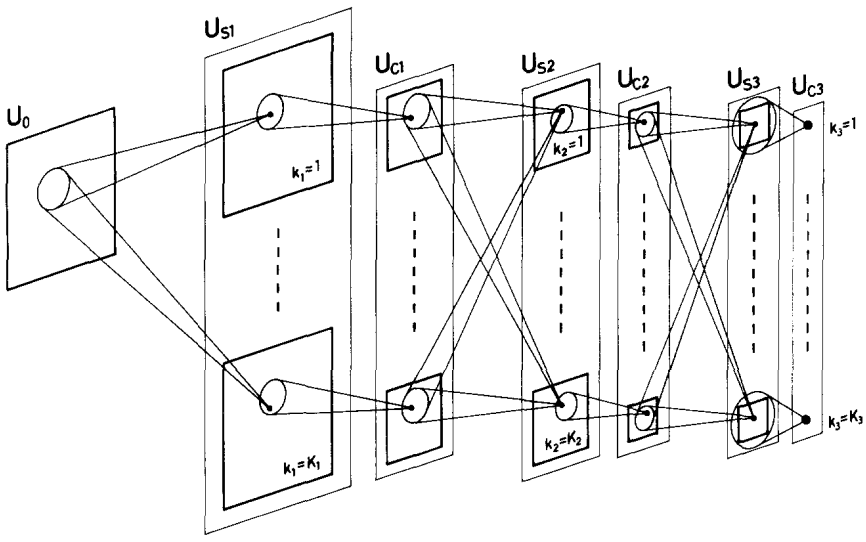
1951 – Marvin Minsky creates the first ANN (Hebbian learning, 40 neurons).

1958 – Frank Rosenblatt creates a perceptron to classify 20×20 images.

1959 – David H. Hubel and Torsten Wiesel demonstrate orientation selectivity and columnar organization in the cat's visual cortex.

1982 – Paul Werbos proposes back-propagation for ANNs.

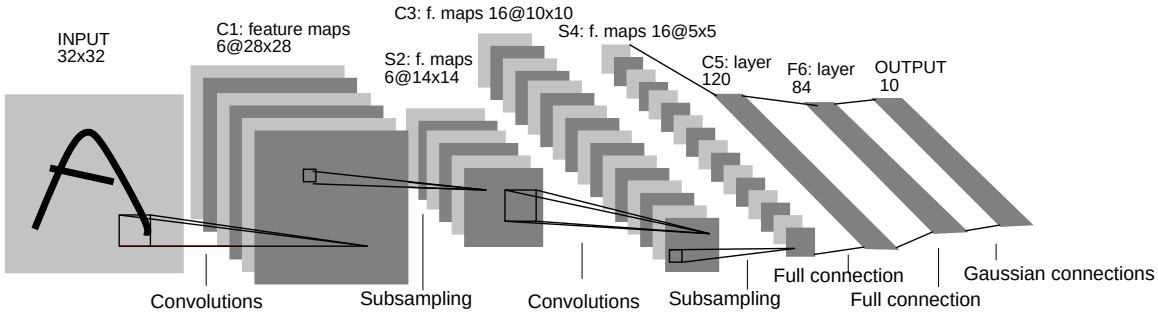
Neocognitron



Follows Hubel and Wiesel's results.

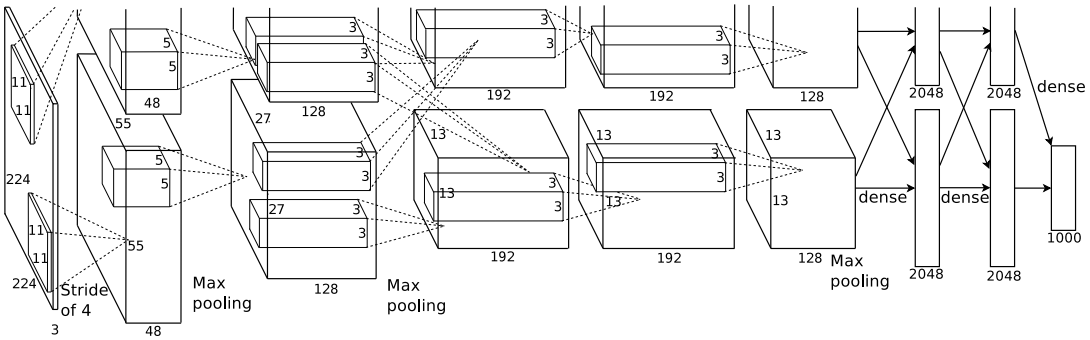
(Fukushima, 1980)

LeNet-5



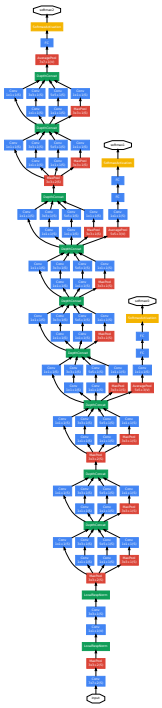
(leCun et al., 1998)

AlexNet



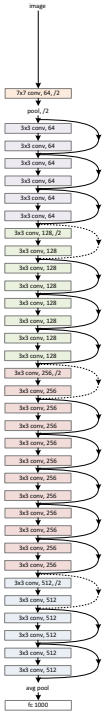
(Krizhevsky et al., 2012)

Inception network



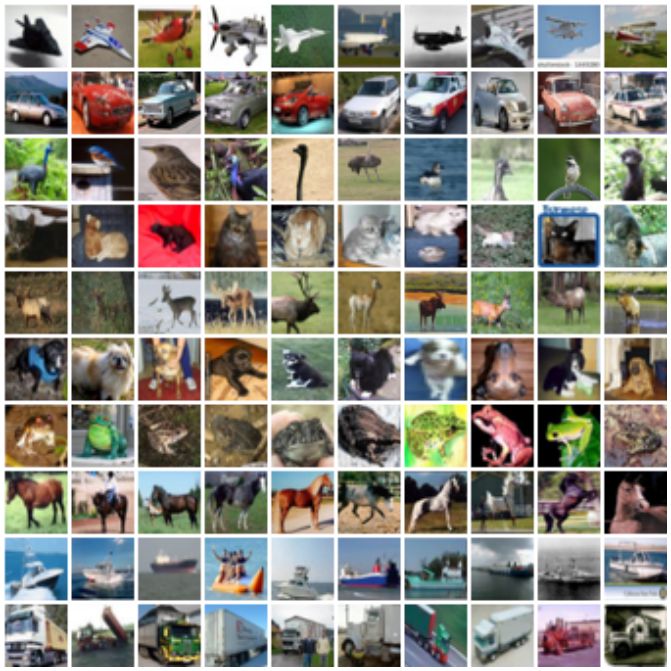
(Szegedy et al., 2015)

Residual networks



(He et al., 2015)

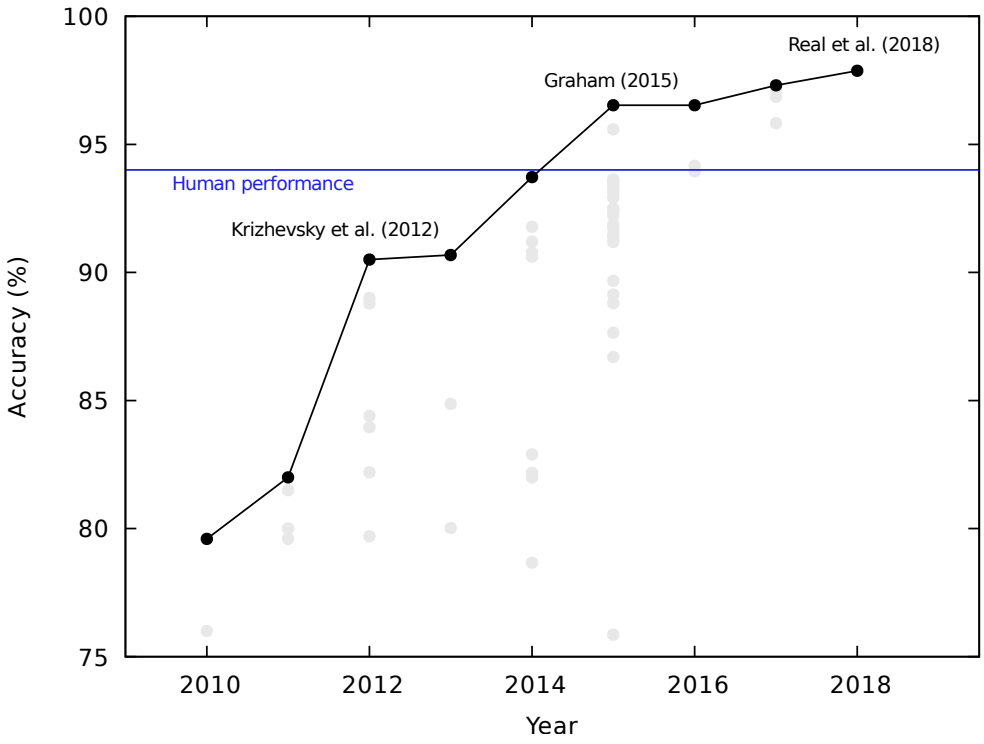
CIFAR10



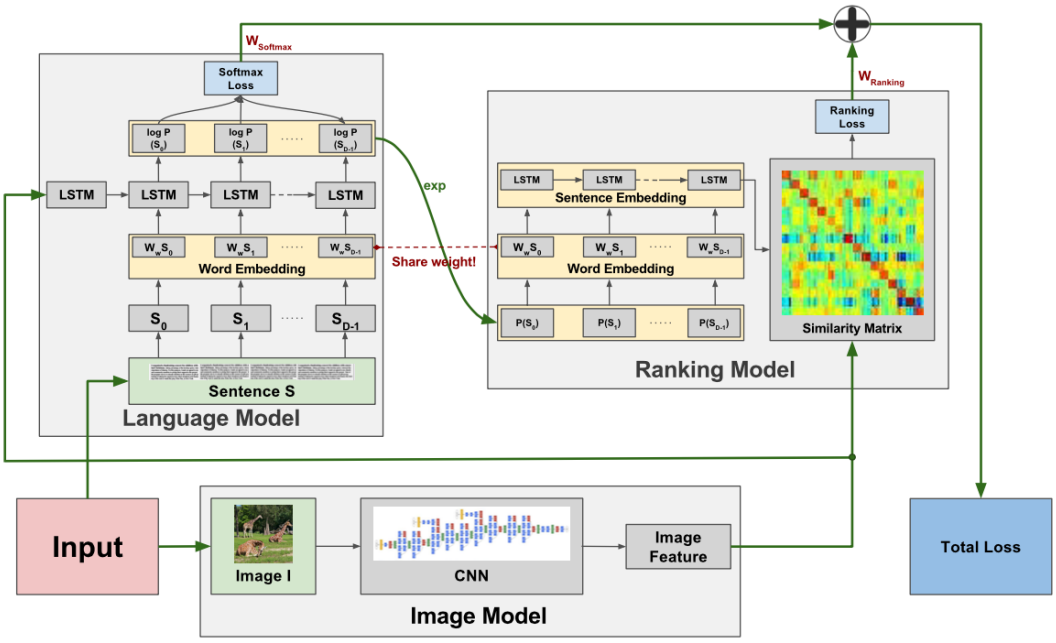
32×32 color images, 50k train samples, 10k test samples.

(Krizhevsky, 2009, chap. 3)

Performance on CIFAR10



This generalization allows to design complex networks of operators dealing with images, sound, text, sequences, etc. and to train them end-to-end.



(Yeung et al., 2015)

Tensor basics

The fundamental data-structure we will manipulate is a tensor: a finite table of numerical values indexed along several discrete dimensions.

- A 0d tensor is a scalar,
- A 1d tensor is a vector (*e.g.* a sound sample),
- A 2d tensor is a matrix (*e.g.* a grayscale image),
- A 3d tensor can be seen as a vector of identically sized matrix (*e.g.* a multi-channel image),
- A 4d tensor can be seen as a matrix of identically sized matrices, or a sequence of 3d tensors (*e.g.* a sequence of multi-channel images),
- etc.

Tensors are used to encode the signal to process, but also the internal states and parameters of models.

Manipulating data through this constrained structure allows to use CPUs and GPUs at peak performance.

PyTorch is a Python library built on top of Torch's THNN computational backend.

Its main features are:

- Efficient tensor operations on CPU/GPU,
- automatic on-the-fly differentiation (autograd),
- optimizers,
- data I/O.

"Efficient tensor operations" encompass both standard linear algebra and, as we will see later, deep-learning specific operations (convolution, pooling, etc.)

A key specificity of PyTorch is the central role of autograd to compute derivatives of *anything!* We will come back to this.


```
>>> x = torch.empty(2, 5)
>>> x.size()
torch.Size([2, 5])
>>> x.fill_(1.125)
tensor([[ 1.1250,  1.1250,  1.1250,  1.1250,  1.1250],
        [ 1.1250,  1.1250,  1.1250,  1.1250,  1.1250]])
>>> x.mean()
tensor(1.1250)
>>> x.std()
tensor(0.)
>>> x.sum()
tensor(11.2500)
>>> x.sum().item()
11.25
```

PyTorch provides interfacing to standard linear operations, such as linear system solving or Eigen-decomposition.

```
>>> y = torch.empty(3).normal_()
>>> y
tensor([ 0.0477,  0.8834, -1.5996])
>>> m = torch.empty(3, 3).normal_()
>>> q, _ = torch.gels(y, m)
>>> torch.mm(m, q)
tensor([[ 0.0477],
        [ 0.8834],
        [-1.5996]])
```


High dimension tensors

A tensor can be of several types:

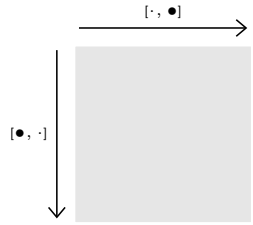
- `torch.float16`, `torch.float32`, `torch.float64`,
- `torch.uint8`,
- `torch.int8`, `torch.int16`, `torch.int32`, `torch.int64`

and can be located either in the CPU's or in a GPU's memory.

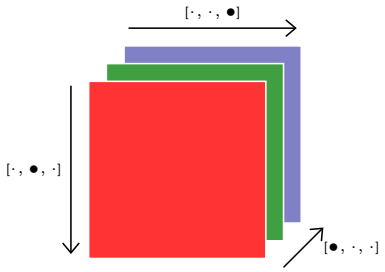
Operations with tensors stored in a certain device's memory are done by that device. We will come back to that later.


```
>>> x = torch.zeros(1, 3)
>>> x.dtype, x.device
(torch.float32, device(type='cpu'))
>>> x = x.long()
>>> x.dtype, x.device
(torch.int64, device(type='cpu'))
>>> x = x.to('cuda')
>>> x.dtype, x.device
(torch.int64, device(type='cuda', index=0))
```

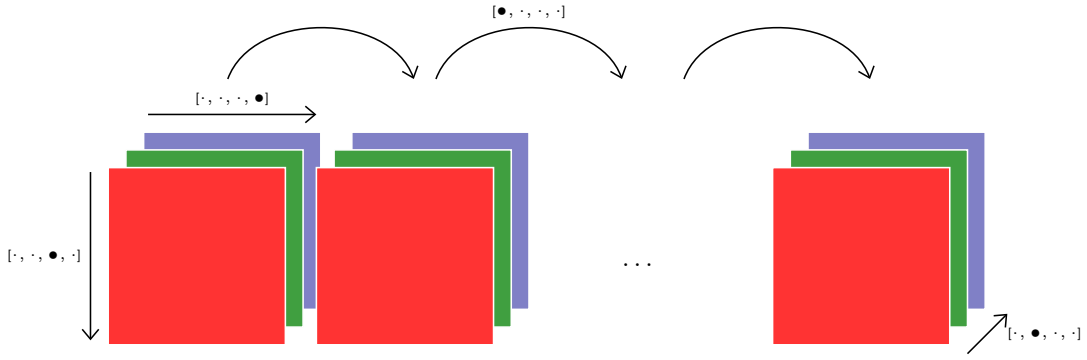
2d tensor (*e.g.* grayscale image)



3d tensor (*e.g.* rgb image)



4d tensor (*e.g.* sequence of rgb images)



Here are some examples from the vast library of tensor operations:

Creation

- `torch.empty(*size, ...)`
- `torch.zeros(*size, ...)`
- `torch.full(size, value, ...)`
- `torch.tensor(sequence, ...)`
- `torch.eye(n, ...)`
- `torch.from_numpy(ndarray)`

Indexing, Slicing, Joining, Mutating

- `torch.Tensor.view(*size)`
- `torch.cat(inputs, dimension=0)`
- `torch.chunk(tensor, chunks, dim=0)[source]`
- `torch.split(tensor, split_size, dim=0)[source]`
- `torch.index_select(input, dim, index, out=None)`
- `torch.t(input, out=None)`
- `torch.transpose(input, dim0, dim1, out=None)`

Filling

- `Tensor.fill_(value)`
- `torch.bernoulli_(proba)`
- `torch.normal_([mu, [std]])`

Pointwise math

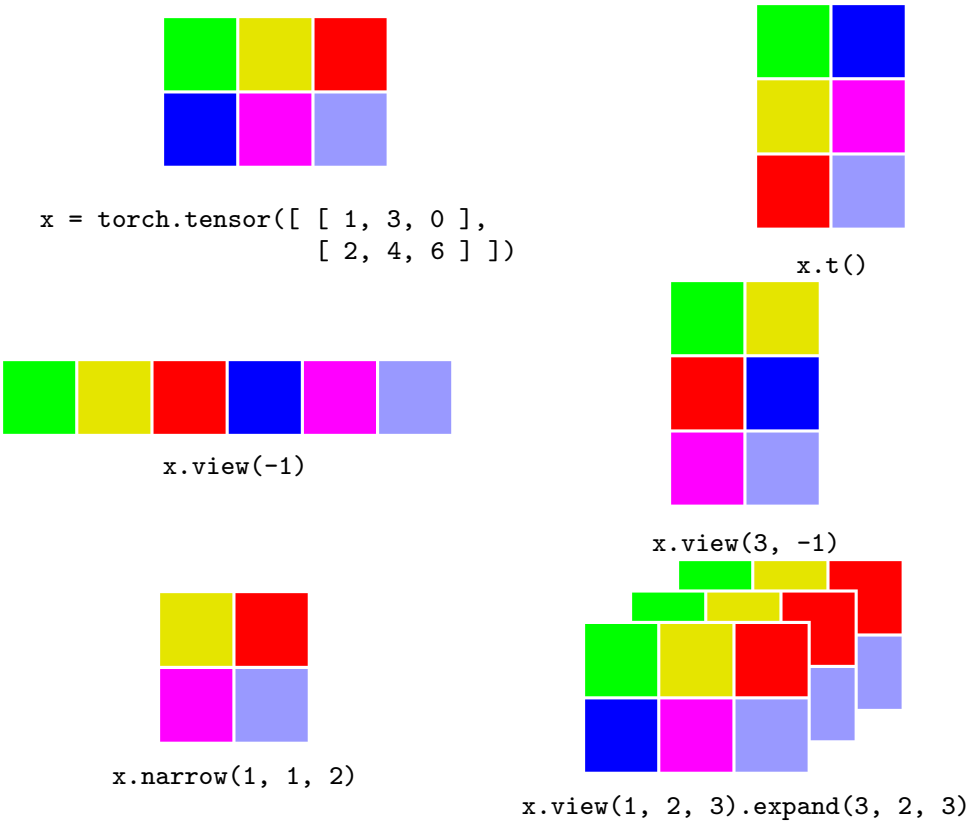
- `torch.abs(input, out=None)`
- `torch.add()`
- `torch.cos(input, out=None)`
- `torch.sigmoid(input, out=None)`
- `(+ many operators)`

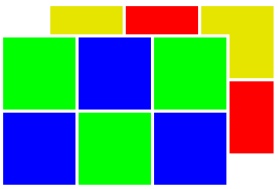
Math reduction

- `torch.dist(input, other, p=2, out=None)`
- `torch.mean()`
- `torch.norm()`
- `torch.std()`
- `torch.sum()`

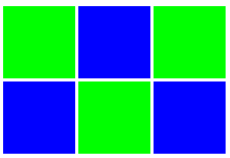
BLAS and LAPACK Operations

- `torch.eig(a, eigenvectors=False, out=None)`
- `torch.gels(B, A, out=None)`
- `torch.inverse(input, out=None)`
- `torch.mm(mat1, mat2, out=None)`
- `torch.mv(mat, vec, out=None)`

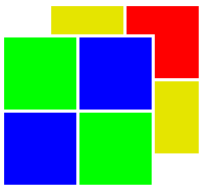




```
x = torch.tensor([ [ [ 1, 2, 1 ],
                     [ 2, 1, 2 ] ],
                   [ [ 3, 0, 3 ],
                     [ 0, 3, 0 ] ] ])
```

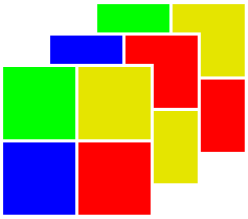


x.narrow(0, 0, 1)

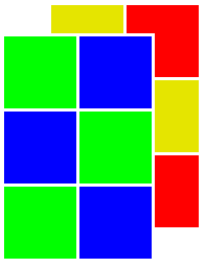


x.narrow(2, 0, 2)

x.transpose(0, 1)



x.transpose(0, 2)



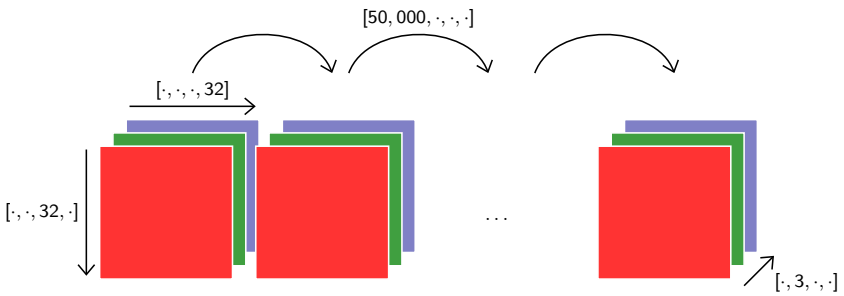
x.transpose(1, 2)

A tensor can encode heavy samples sets.

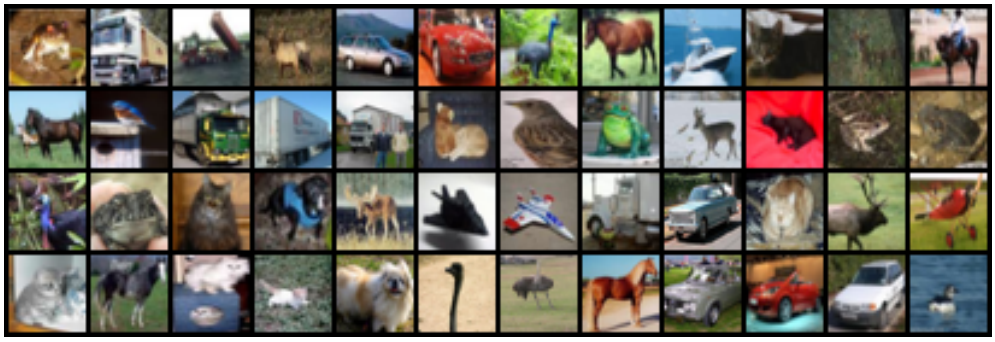
```
import torch, torchvision
cifar = torchvision.datasets.CIFAR10('./cifar10/', train = True, download = True)
x = torch.from_numpy(cifar.data).permute(0, 3, 1, 2).float()
x = x / 255
print(x.type(), x.size(), x.min().item(), x.max().item())
```

prints

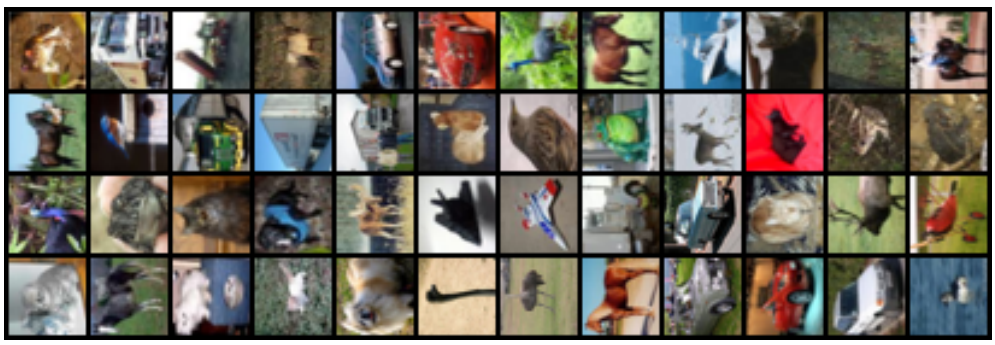
```
Files already downloaded and verified
torch.FloatTensor torch.Size([50000, 3, 32, 32]) 0.0 1.0
```



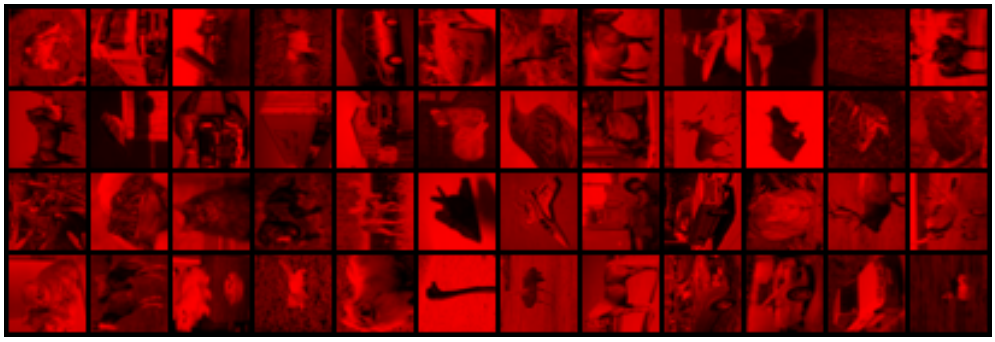
```
# Narrows to the first images, converts to float
x = x.narrow(0, 0, 48).float()

# Saves these samples as a single image
torchvision.utils.save_image(x, 'cifar-4x12.png', nrow = 12)
```




```
# Switches the row and column indexes
x.transpose_(2, 3)
torchvision.utils.save_image(x, 'cifar-4x12-rotated.png', nrow = 12)
```

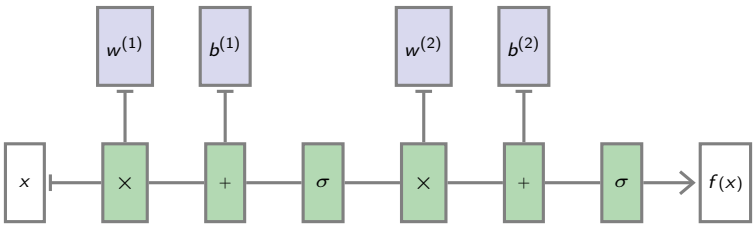


```
# Kills the green and blue channels
x.narrow(1, 1, 2).fill_(0)
torchvision.utils.save_image(x, 'cifar-4x12-rotated-and-red.png', nrow = 12)
```

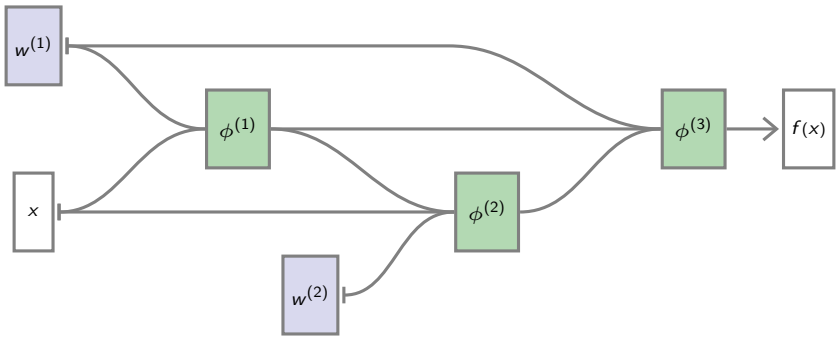


DAG networks

We can generalize an MLP



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



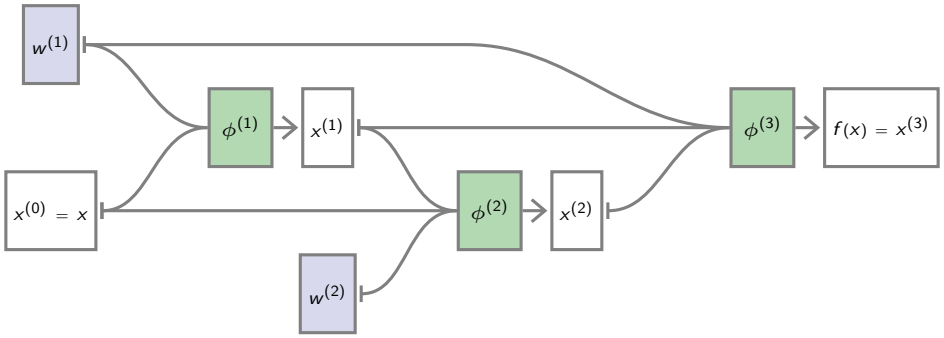
If $(a_1, \ldots, a_Q) = \phi(b_1, \ldots, b_R)$, we use the notation

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

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

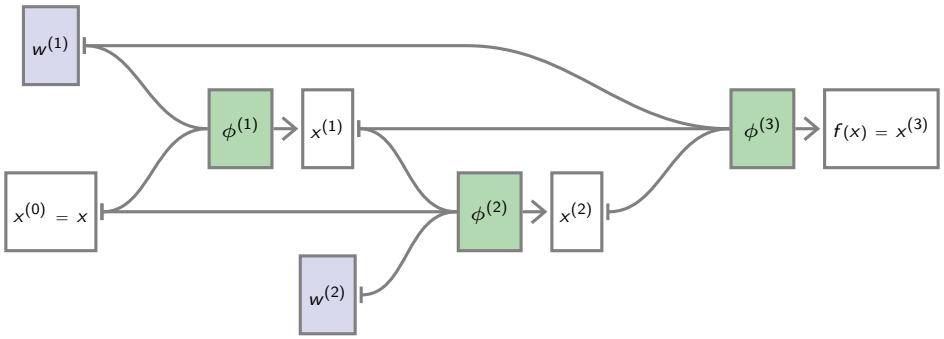
$$\left[\frac{\partial a}{\partial c}\right] = J_{\phi|c} = \begin{pmatrix} \frac{\partial a_1}{\partial c_1} & \cdots & \frac{\partial a_1}{\partial c_S} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_Q}{\partial c_1} & \cdots & \frac{\partial a_Q}{\partial c_S} \end{pmatrix}.$$

Evaluation (aka "forward pass")



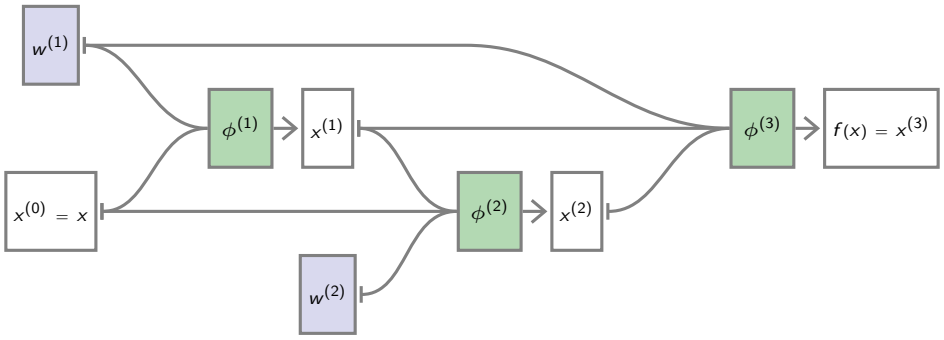
$$
\begin{aligned}
x^{(0)} &= x \\
x^{(1)} &= \phi^{(1)}(x^{(0)}; w^{(1)}) \\
x^{(2)} &= \phi^{(2)}(x^{(0)}, x^{(1)}; w^{(2)}) \\
f(x) = x^{(3)} &= \phi^{(3)}(x^{(1)}, x^{(2)}; w^{(1)})
\end{aligned}
$$

Derivatives w.r.t activations (aka "backward pass")



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

Derivatives w.r.t parameters



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

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

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

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

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.

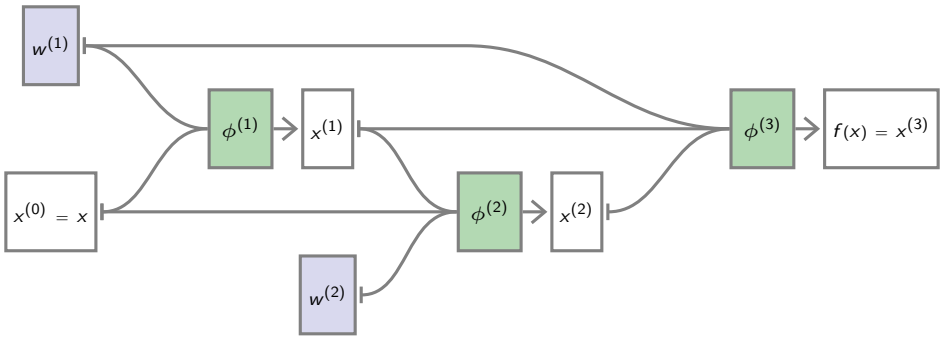
Writing from scratch a large neural network is complex and error-prone.

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



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

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

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

```
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

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
```


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

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

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

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

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

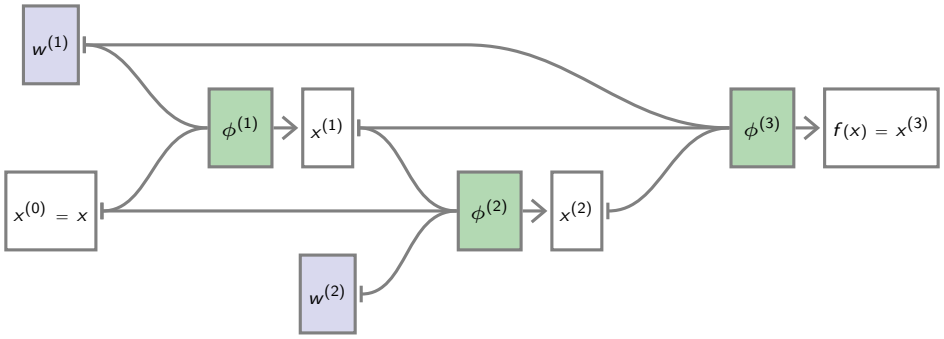
```
>>> x = torch.tensor([ -3., 2., 5. ]).requires_grad_()
>>> u = x.pow(3).sum()
>>> x.grad
>>> u.backward()
>>> x.grad
tensor([27., 12., 75.])
```

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

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

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

So we can run a forward/backward pass on

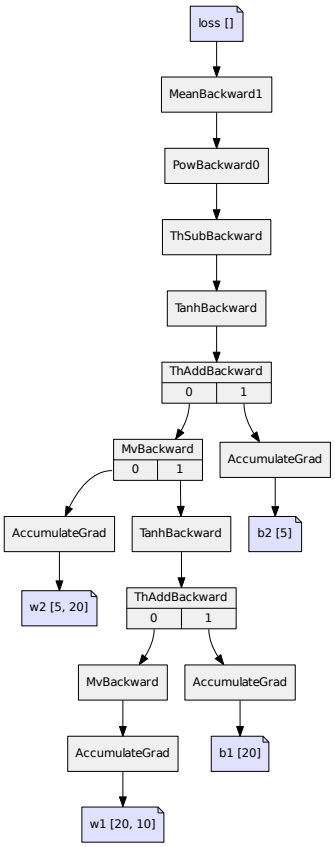


$$\phi^{(1)}\left(x^{(0)}; w^{(1)}\right) = w^{(1)}x^{(0)}$$
$$\phi^{(2)}\left(x^{(0)}, x^{(1)}; w^{(2)}\right) = x^{(0)} + w^{(2)}x^{(1)}$$
$$\phi^{(3)}\left(x^{(1)}, x^{(2)}; w^{(1)}\right) = w^{(1)}\left(x^{(1)} + x^{(2)}\right)$$

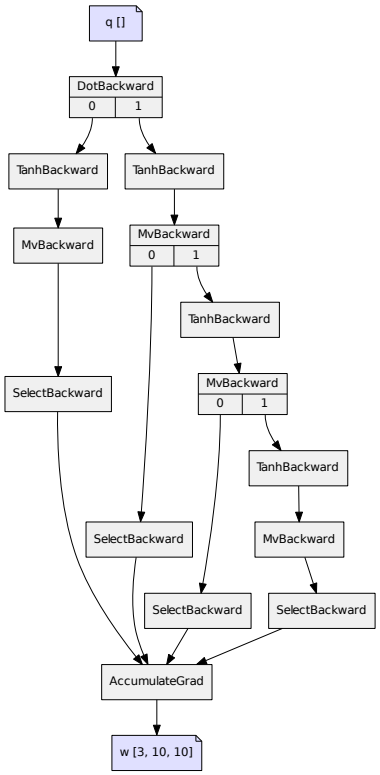
```
w1 = torch.rand(5, 5).requires_grad_()
w2 = torch.rand(5, 5).requires_grad_()
x = torch.empty(5).normal_()

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

q = x3.norm()

q.backward()
```



```
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)
```



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

```
>>> x = torch.empty(8).normal_()
>>> x
tensor([ 0.6502,  0.1161, -1.1509, -0.2337, -1.0798, -2.3269,  0.0165,  1.1905])
>>> F.relu(x)
tensor([0.6502, 0.1161, 0.0000, 0.0000, 0.0000, 0.0000, 0.0165, 1.1905])
```

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

```
>>> x = torch.empty(2, 3).normal_()
>>> m = torch.nn.Linear(3, 4)
>>> m(x)
tensor([[-0.6673,  0.2003, -0.7815, -0.9381],
        [ 0.1856, -0.1588, -0.0570, -0.2990]], grad_fn=<AddmmBackward>)
```

Convolutions

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

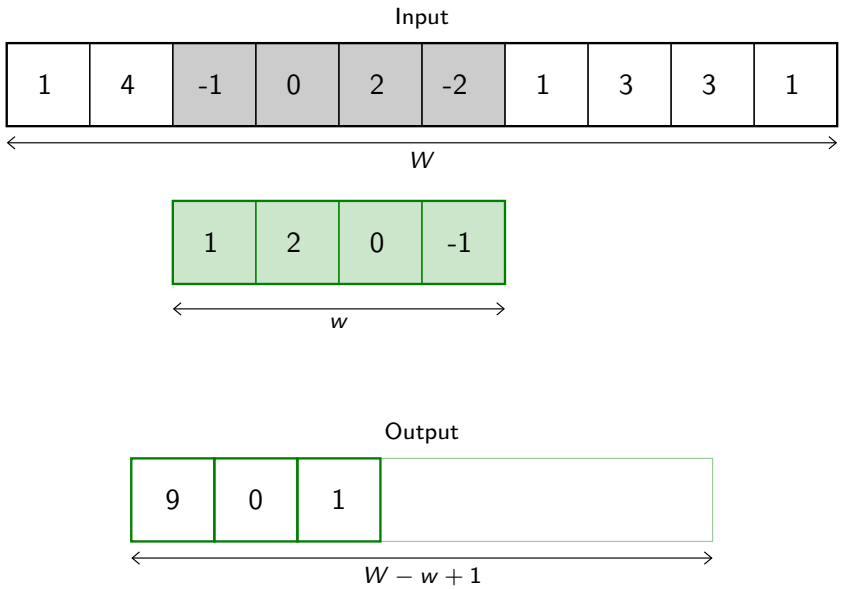
For instance a linear layer taking a 256×256 RGB image as input, and producing an image of same size would require

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

parameters, with the corresponding memory footprint ($\simeq$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 convolution layer embodies this idea. It applies the same linear transformation locally, everywhere, and preserves the signal structure.



Formally, in 1d, given

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

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

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

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

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

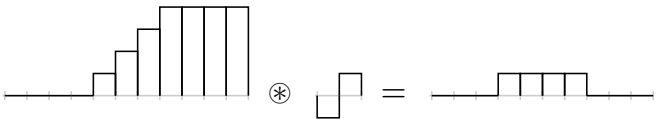
for instance

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

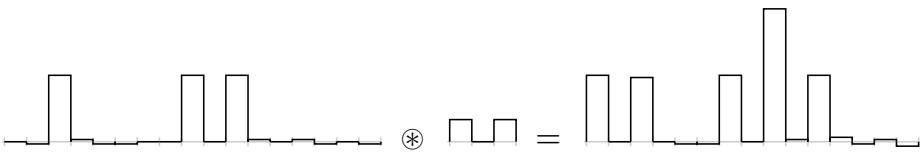
This differs from the usual convolution since the kernel and the signal are both visited in increasing index order.

Convolution can implement in particular differential operators, *e.g.*

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



or crude "template matcher", *e.g.*



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

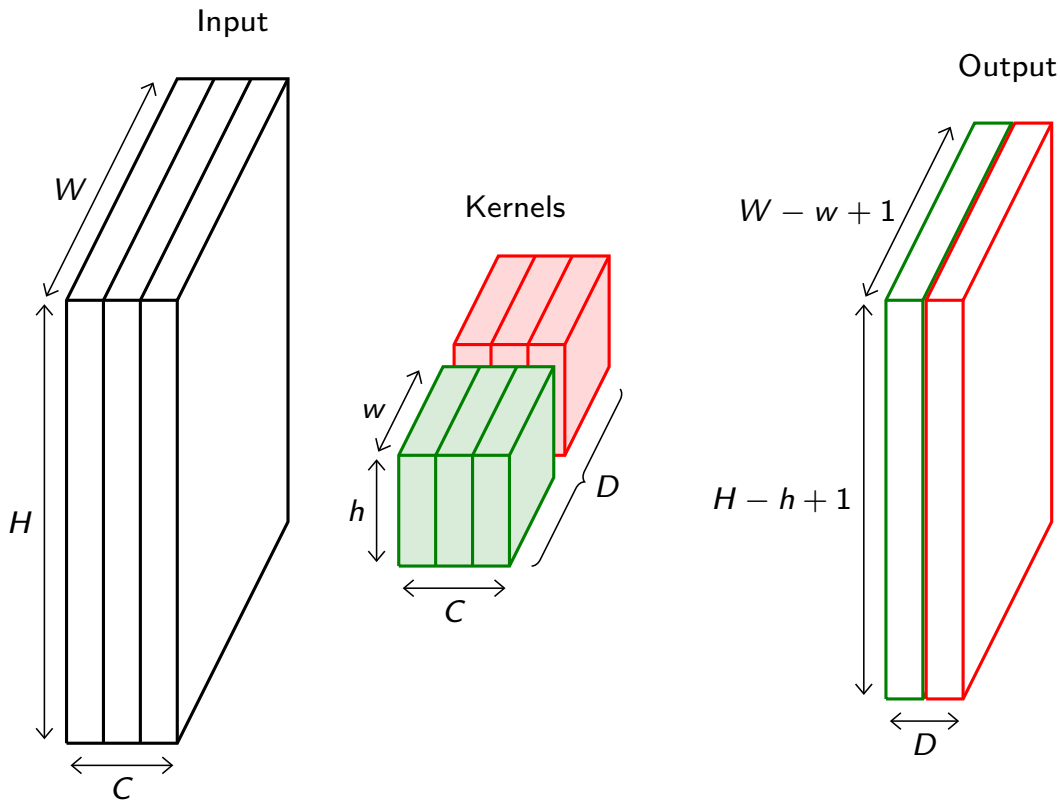
It generalizes naturally to a multi-dimensional input, although specification can become complicated.

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

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

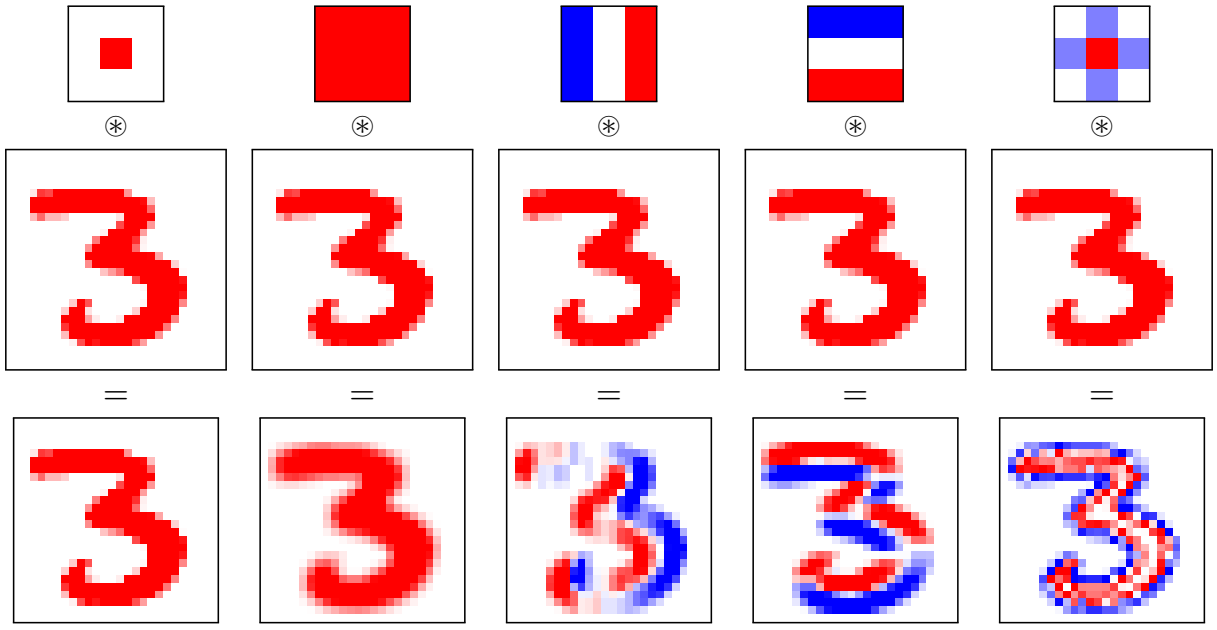
and the result is of dimension

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

```
>>> 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.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])
```

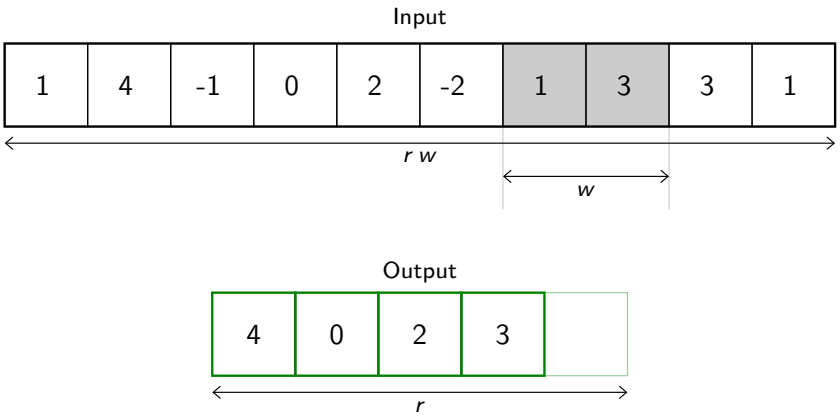
Pooling

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.

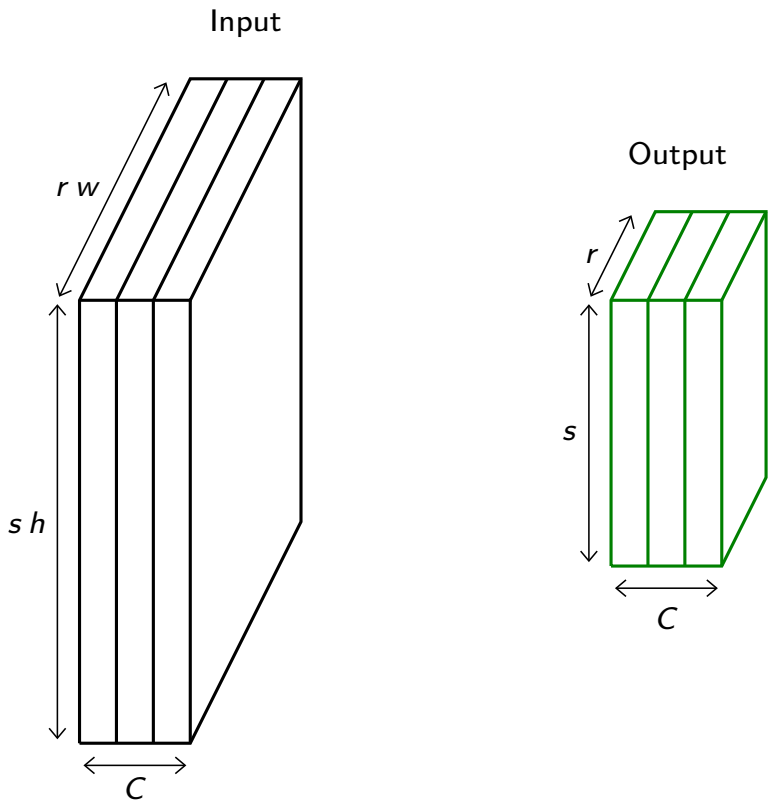
Such an operation aims at grouping several activations into a single "more meaningful" one.

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:

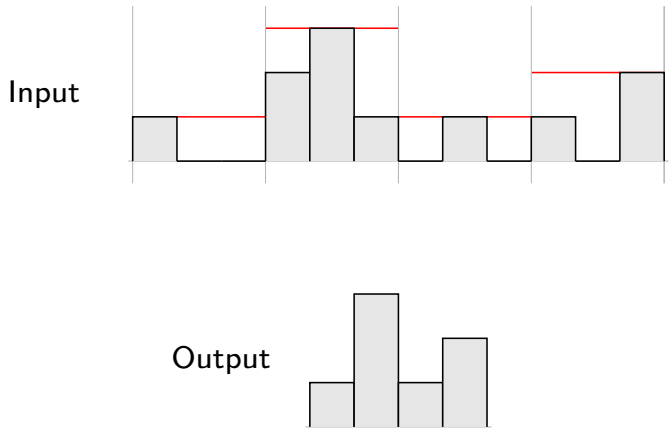


The **average pooling** computes average values per block instead of max values.



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).

Stochastic gradient descent

To minimize a loss of the form

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

the standard gradient-descent algorithm update has the form

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

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

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

and when we compute ℓ_n, we have already computed $\ell_1, \ldots, \ell_{n-1}$, 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

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

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

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

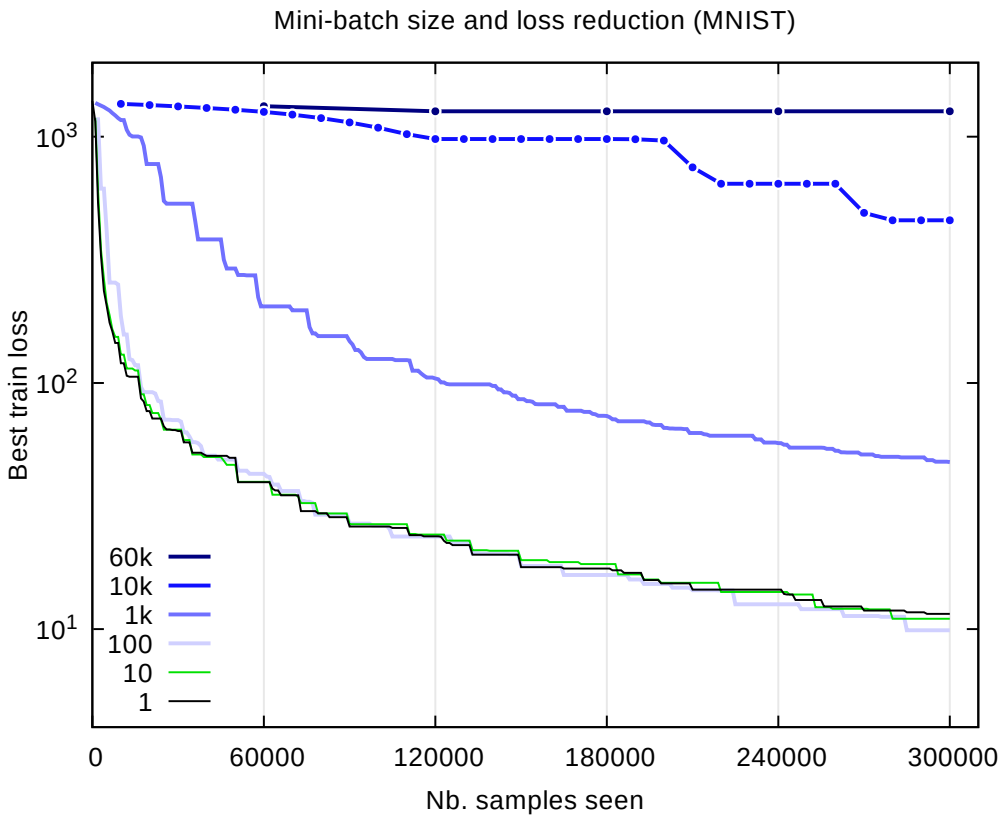
However this does not benefit from the speed-up of batch-processing.

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.



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.

Putting all this together

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

- various layers,
- automatic differentiation,
- 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.data.view(-1, 1, 28, 28).float()
train_targets = train_set.targets

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_targets = train_input.to(device), train_targets.to(device)

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

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


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).

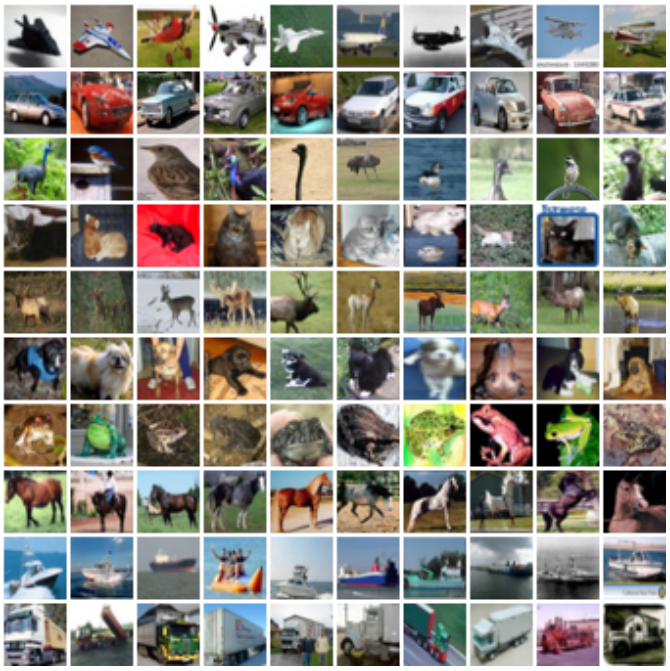
Regarding the learning rate, for training to succeed it has to

- 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 minima $\Rightarrow$ 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.

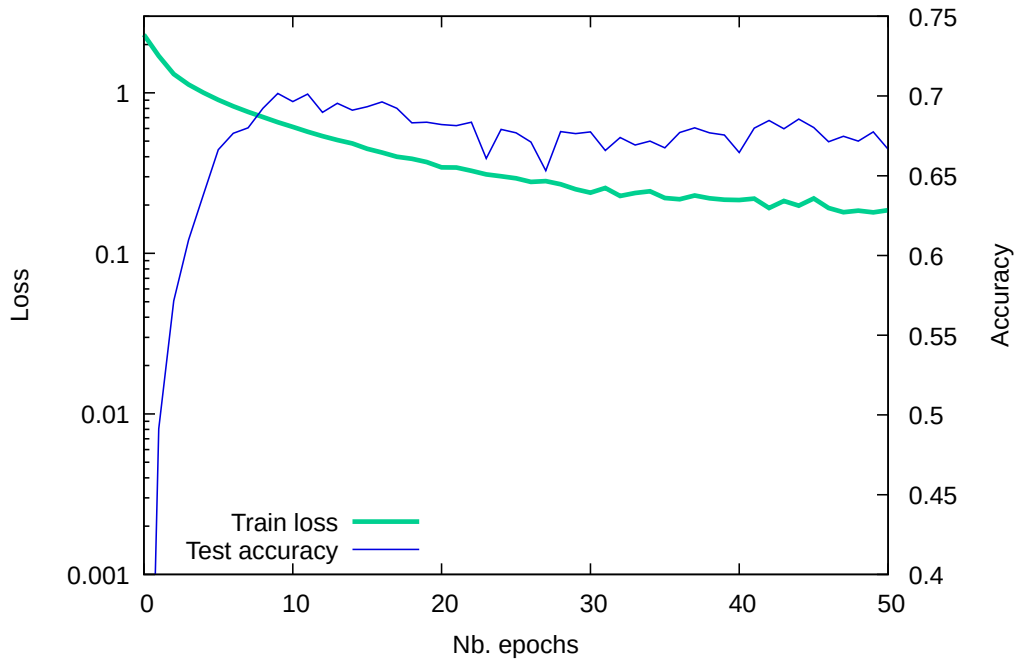
CIFAR10 data-set



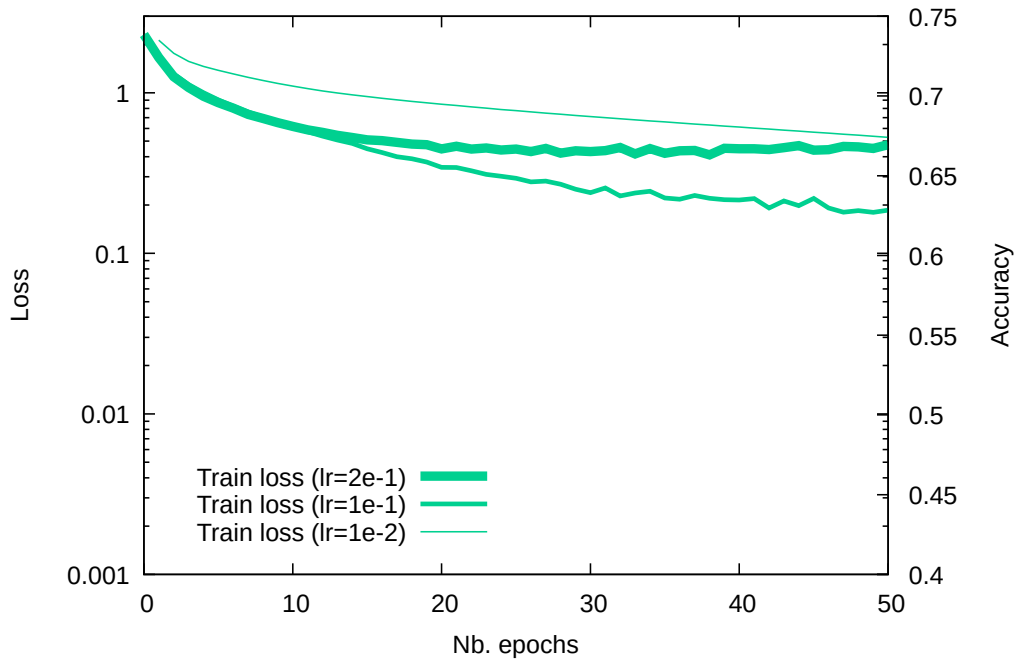
32×32 color images, $50,000$ train samples, $10,000$ test samples.

(Krizhevsky, 2009, chap. 3)

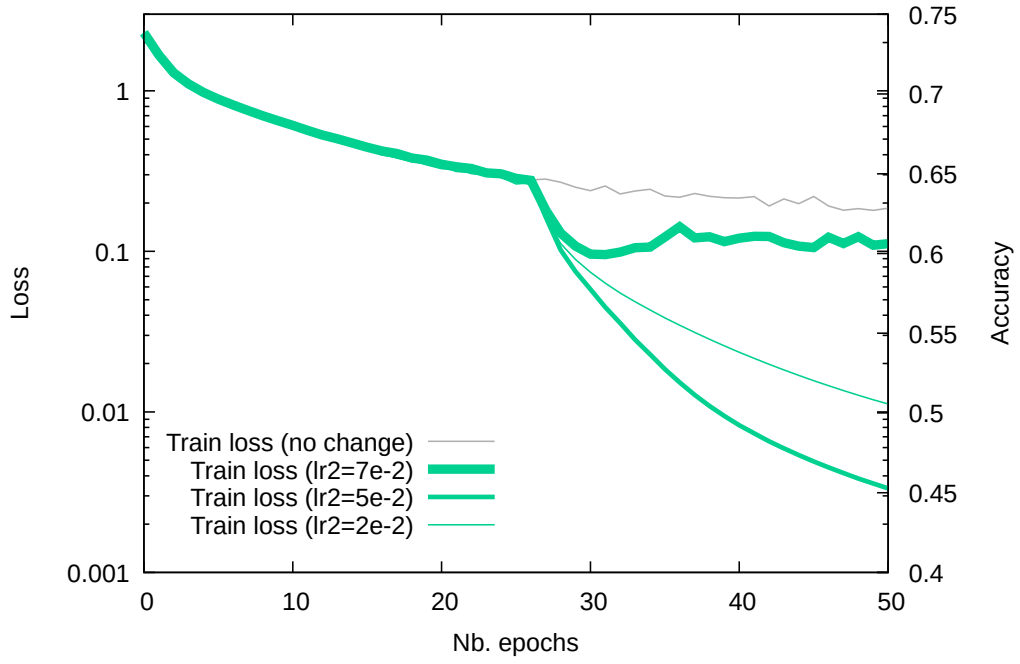
Small convnet on CIFAR10, cross-entropy, batch size 100, $\eta = 1e-1$.



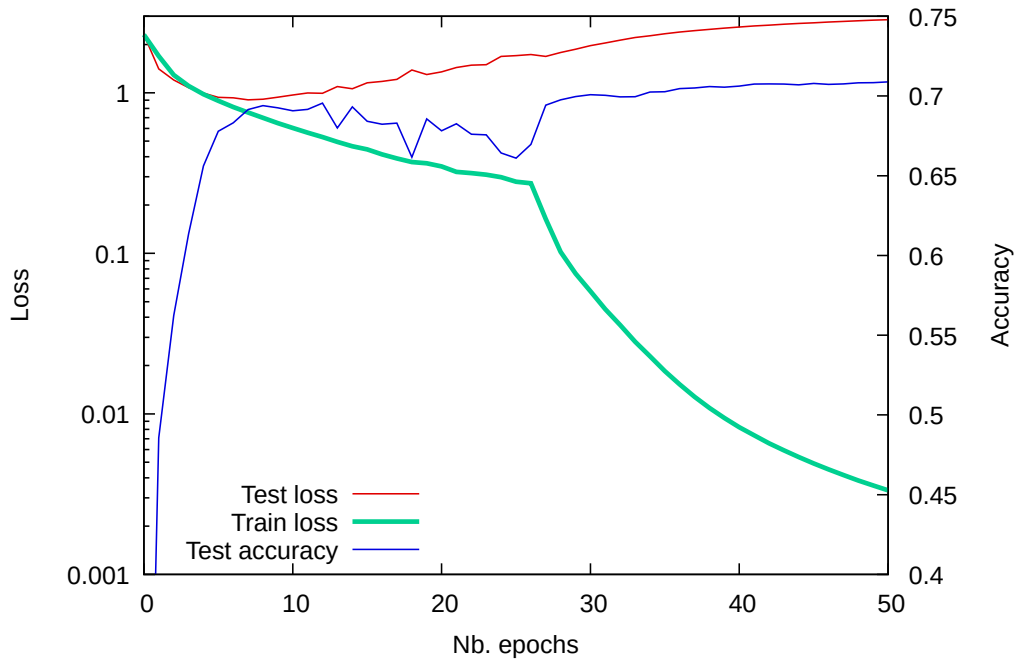
Small convnet on CIFAR10, cross-entropy, batch size 100



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



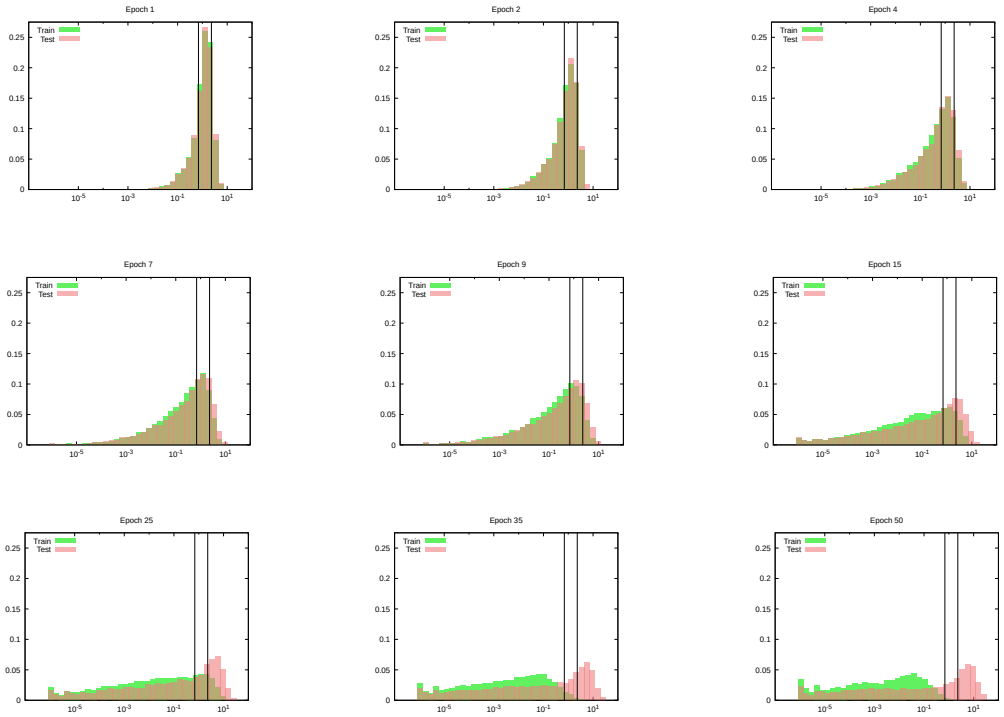
While the test error still goes down, the test loss may increase, as it gets even worse on misclassified examples, and decreases less on the ones getting fixed.



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.



References

K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics, 36(4): 193–202, April 1980.

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.

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, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278–2324, 1998.

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.

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.

S. Yeung, O. Russakovsky, G. Mori, and L. Fei-Fei. End-to-end learning of action detection from frame glimpses in videos. CoRR, abs/1511.06984, 2015.