IfI Summer School 2018 on Machine Learning

Deep Learning #4 – Convolution Neural Networks

François Fleuret
http://fleuret.org/ifi/
June 26, 2018

Convolutional layers

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

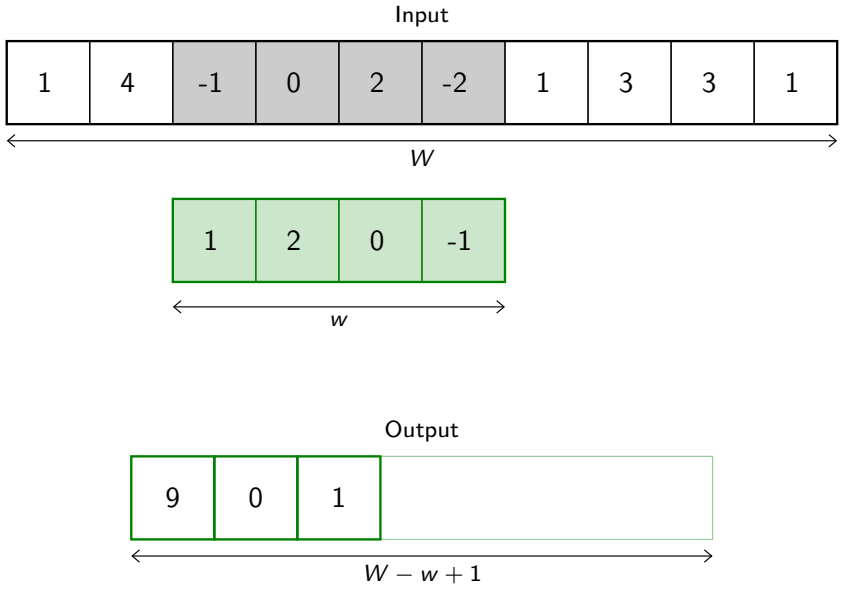
For instance a linear layer taking a 256×256 RGB image as input, and producing an image of same size would require

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

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

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

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



Formally, in 1d, given

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

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

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

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

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

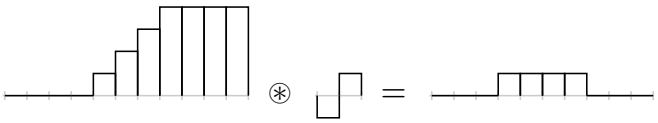
for instance

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

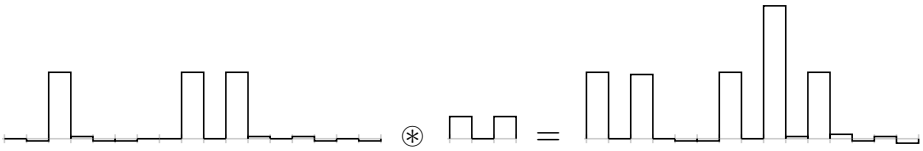
This differs from the usual convolution since the kernel and the signal are both visited in increasing index order.

Convolution can implement a differential operator

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



or a crude "template matcher"



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

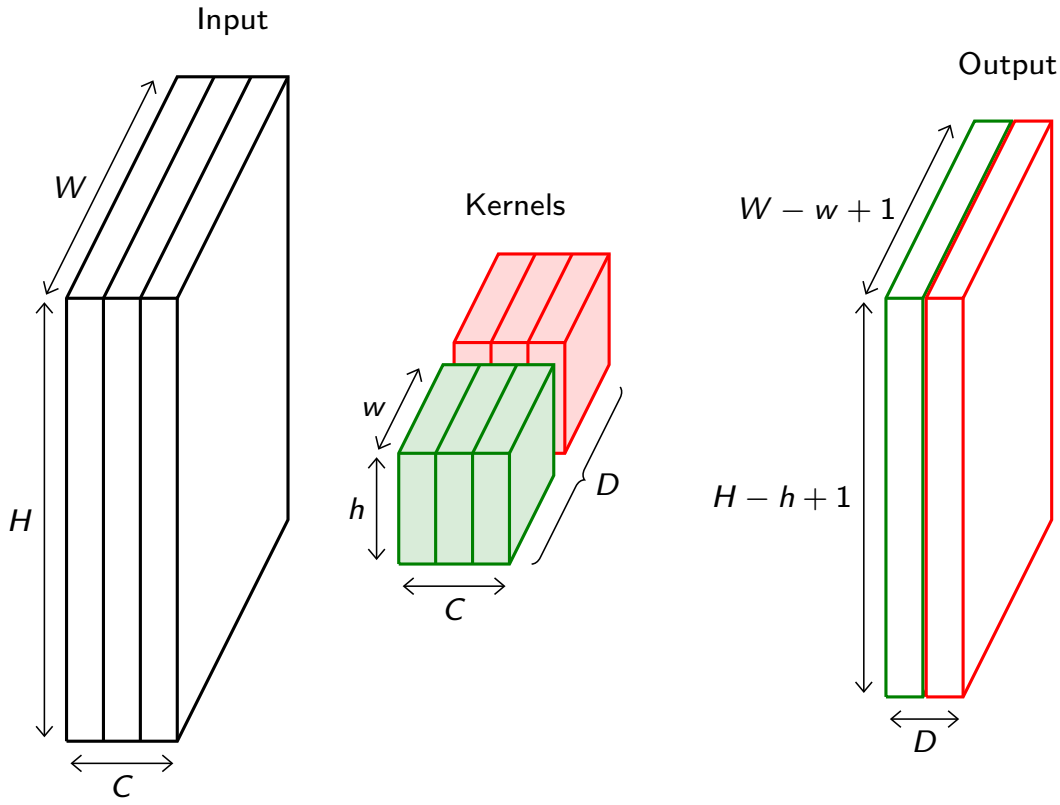
It generalizes naturally to a multi-dimensional input, although specification can become complicated.

Its most usual form for "convolutional networks" processes a 3d tensor as input (*i.e.* a multi-channel 2d signal) to output a 2d tensor.

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

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

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



Note that convolution **preserves the signal support structure**.

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

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

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

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

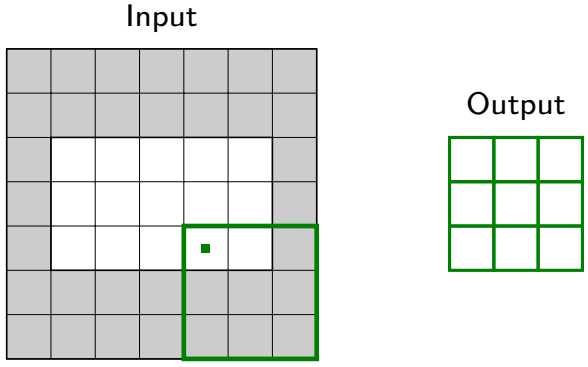
In the context of convolutional networks, a standard linear layer is called a **fully connected layer** since every input influences every output.

Stride, padding, and dilation

Convolution operations have two more standard parameters:

- The **padding** specifies the size of a zeroed frame added around the signal,
- The **stride** specifies a step size when moving the filter 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 $C \times 3 \times 3$.

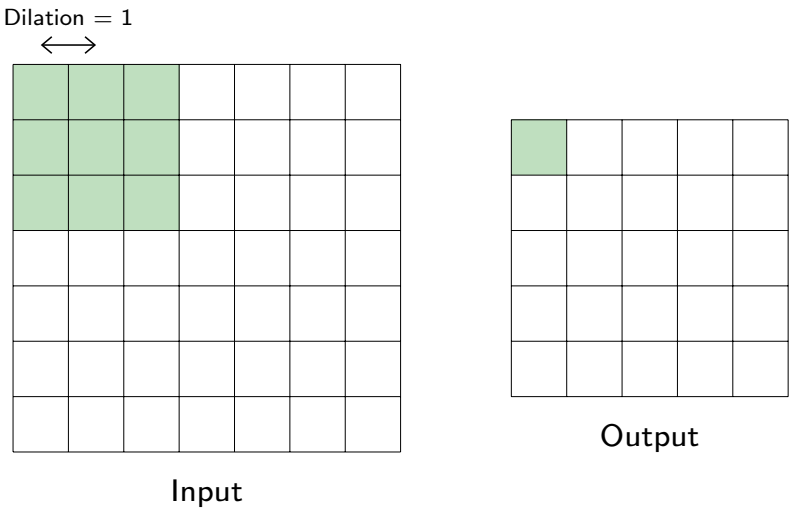


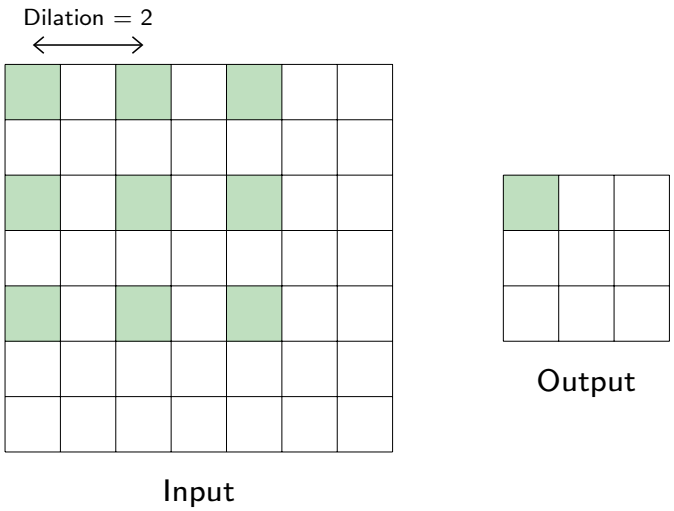
Padding can be useful to generate an output of same size as the input.

Recently, the notion of **dilated convolutions** was introduced to increase the receptive fields without increasing the number of parameters (Yu and Koltun, 2015).

The **dilation** parametrizes the expansion of the filter. 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".





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 $1 + (3 - 1)4 - 1 = 8$.

```
>>> from torch import nn, Tensor
>>> x = Tensor(1, 1, 20, 30).normal_()
>>> f = nn.Conv2d(1, 1, kernel_size = 3, dilation = 4)
>>> f(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 (combining multiple activations to produce one) are only in the convolutional layers.

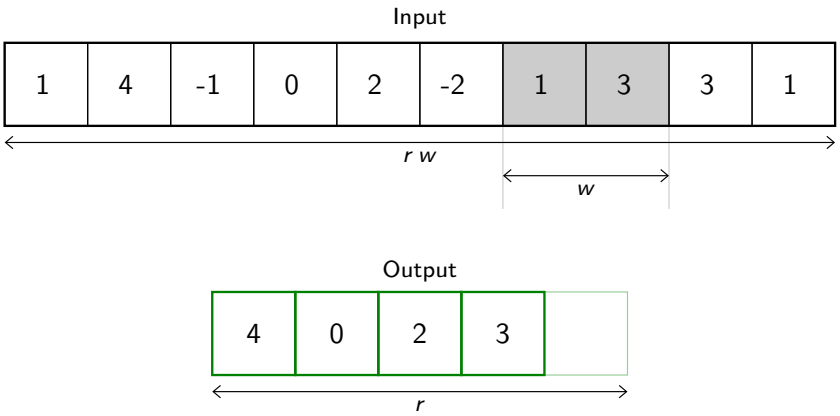
Pooling

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

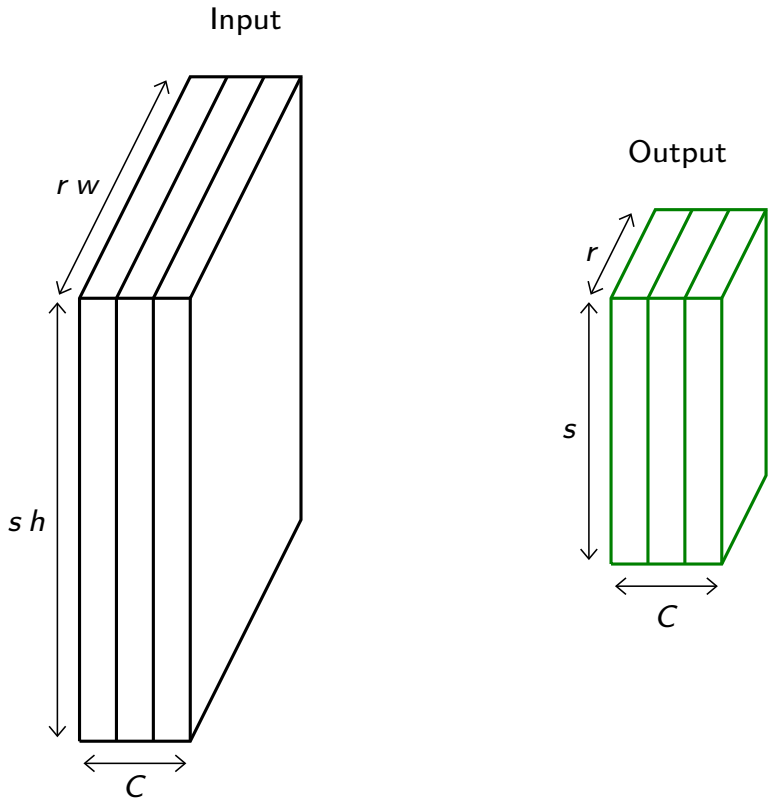
As for convolution, it makes sense to reduce the signal's size in a way that preserves its structure, just "down-scaling it".

The standard operation to do this is **pooling**, and aims at grouping several activations into a single "more meaningful" one.

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

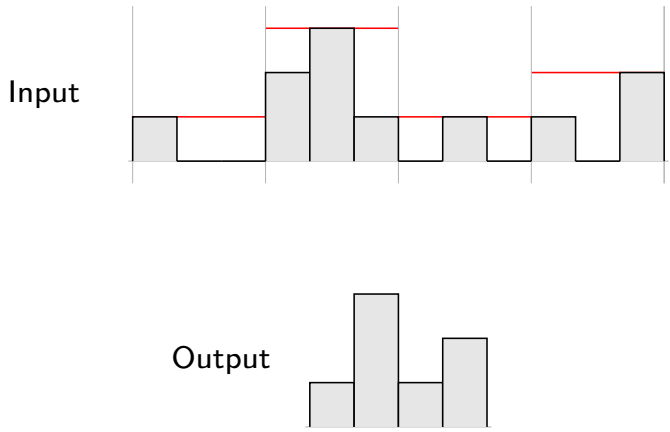


The **average pooling** would compute the mean per block instead of the max.



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

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



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

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

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


torch.nn.Module

PyTorch provides a vast collection of `Module`s, which implement standard operations and can be combined into complicated "deep" architectures.

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

Modules from `torch.nn` are components for networks which embed `torch.nn.Parameter`s to be optimized during training, and criteria (*e.g.* losses). They usually use `torch.nn.functional`s.

From now on, we will use the standard header

```
from torch import nn
from torch.nn import functional as F
```

And in the next slides, we will look at components to build our first convolutional neural network:

- `F.relu`
- `F.max_pool2d`
- `nn.Conv2d`
- `nn.Linear`
- `nn.CrossEntropyLoss`

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

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

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

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


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

Takes as input either a $C \times H \times W$ or $N \times C \times H \times W$ tensor, and a kernel size which can be a single integer k or a pair (k, l), and applies the max-pooling on each channel of each sample separately.

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

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

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

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

Implements a fully-connected layer with the given input and output dimensions.

```
>>> f = nn.Linear(in_features = 10, out_features = 4)
>>> f.weight.size()
torch.Size([4, 10])
>>> f.bias.size()
torch.Size([4])
>>> x = Tensor(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.


```
nn.Conv2d(in_channels, out_channels,
          kernel_size,
          stride=1, padding=0, dilation=1, groups=1, bias=True)
```

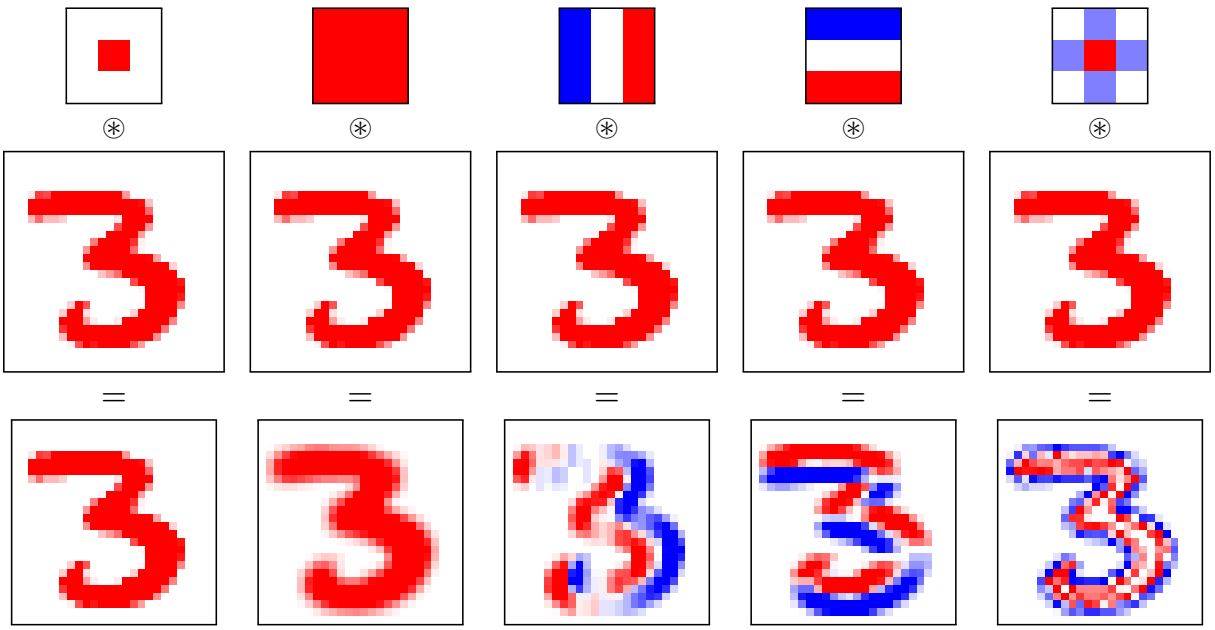
Implements a standard 2d convolutional layer.

It takes as input either a $C \times H \times W$ or $N \times C \times H \times W$ tensor, and a kernel size which can be a single integer k or a pair (k, l), and applies the convolution on each channel of each sample separately.

```
>>> f = nn.Conv2d(in_channels = 4, out_channels = 5, kernel_size = (2, 3))
>>> f.weight.size()
torch.Size([5, 4, 2, 3])
>>> f.bias.size()
torch.Size([5])
>>> x = Tensor(117, 4, 10, 3).normal_()
>>> y = f(x)
>>> y.size()
torch.Size([117, 5, 9, 1])
```

As for the fully connected layer, weights and biases are randomized.

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

f = nn.Conv2d(1, 5, kernel_size=3)

f.bias.detach().zero_()

f.weight[0] = Tensor([ [  0,  0,  0 ],
                       [  0,  1,  0 ],
                       [  0,  0,  0 ] ])

f.weight[1] = Tensor([ [  1,  1,  1 ],
                       [  1,  1,  1 ],
                       [  1,  1,  1 ] ])

f.weight[2] = Tensor([ [ -1,  0,  1 ],
                       [ -1,  0,  1 ],
                       [ -1,  0,  1 ] ])

f.weight[3] = Tensor([ [ -1, -1, -1 ],
                       [  0,  0,  0 ],
                       [  1,  1,  1 ] ])

f.weight[4] = Tensor([ [  0, -1,  0 ],
                       [ -1,  4, -1 ],
                       [  0, -1,  0 ] ])

y = f(x)

save_2d_tensor_as_image(f.weight[0], 'conv-filters-{:d}.png',
                        signed = True)

save_2d_tensor_as_image(x[0], 'conv-mnist-orig.png', signed = True)

save_2d_tensor_as_image(y[0], 'conv-mnist-results-{:d}.png',
                        signed = True)
```

`nn.CrossEntropyLoss()`

Implements the "Cross-Entropy" loss. It takes as operand a 2d float tensor $x \in \mathbb{R}^{N \times C}$ and a 1d long tensor $y \in \mathbb{N}^N$ and computes

$$l(x, y) = -\frac{1}{N} \sum_n \log \frac{\exp x[n, y[n]]}{\sum_c \exp x[n, c]}$$

This loss is high if the score of the correct label $x[n, y[n]]$ is not notably higher than the others.

We can justify it formally if we interpret

$$\hat{P}(Y_n = y) \propto \exp x[n, y].$$


```
>>> f = Tensor([[-1, -3, 4], [-3, 3, -1]])
>>> target = torch.LongTensor([0, 1])
>>> criterion = torch.nn.CrossEntropyLoss()
>>> criterion(f, target).item()
2.514101028442383
```

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 first parameter to a criterion is the "input" and the second the "target". As with the cross-entropy, they can be of different dimensions and types.

Criteria do not compute the gradient with respect to the target, and will not accept a `Tensor` with `requires_grad` to `True` as the target.

```
>>> f = nn.MSELoss()
>>> x = Tensor([ 3, 2 ]).requires_grad_()
>>> y = 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
```


We can now put all this together and define our first convolutional network for MNIST, with two convolutional layers, and two fully-connected layers:

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

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

The method `torch.Module.cuda()` moves all the parameters and buffers of the module (and registered sub-modules recursively) to the GPU, and conversely, `torch.Module.cpu()` moves them to the CPU.

Although they do not have a "_" in their names, these `Module` operations make changes in-place.

A typical snippet of code to use the GPU would be

```
if torch.cuda.is_available():
    model.cuda()
    criterion.cuda()
    train_input, train_target = train_input.cuda(), train_target.cuda()
    test_input, test_target = test_input.cuda(), test_target.cuda()
```


A very simple way to leverage multiple GPUs is to use

`nn.DataParallel(module, device_ids)`

The `forward` of the resulting module will

1. Split the input mini-batch along the first dimension in as many mini-batches as there are GPUs in `device_ids`
2. send them to the `forward`s of clones of `module` located on each GPU,
3. concatenate the results.

```
class Dummy(nn.Module):
    def __init__(self, m):
        super(Dummy, self).__init__()
        self.m = m

    def forward(self, x):
        print('Dummy.forward', x.size(), torch.cuda.current_device())
        return self.m(x)


x = Tensor(50, 10).normal_()
m = Dummy(nn.Linear(10, 5))
x = x.cuda()
m = m.cuda()

print('Without data_parallel')
y = m(x)

print()

mp = nn.DataParallel(m, range(torch.cuda.device_count()))

print('With data_parallel')
y = mp(x)
```

prints

```
Without data_parallel
Dummy.forward torch.Size([50, 10]) 0

With data_parallel
Dummy.forward torch.Size([25, 10]) 0
Dummy.forward torch.Size([25, 10]) 1
```


Creating a 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
```

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

`Module`s added as attributes are seen by `Module.parameters()`, which returns an iterator over the model's parameters for optimization.

```
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])
```

`Parameter`s added as attributes are also seen by `Module.parameters()`.

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(Tensor(123, 456))
        self.ouch = {}
        self.ouch[0] = 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])
```


The proper policy then is to use `Module.add_module(name, module)`

```
class NotBuggyAnymore(nn.Module):
    def __init__(self):
        super(NotBuggyAnymore, self).__init__()
        self.conv = nn.Conv2d(1, 32, kernel_size=5)
        self.param = Parameter(Tensor(123, 456))
        self.add_module('ahhh_0', nn.Linear(543, 21))

model = NotBuggyAnymore()

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])
```

These modules are added as attributes, and can be accessed with `getattr`.

`Module.register_parameter(name, parameter)` allows to similarly register `Parameter`s explicitly.

Another option is to add modules in a field of type `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(Tensor(123, 456))
        self.other_stuff = nn.ModuleList()
        self.other_stuff.append(nn.Linear(50, 75))
        self.other_stuff.append(nn.Linear(125, 999))

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([75, 50])
torch.Size([75])
torch.Size([999, 125])
torch.Size([999])
```


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 their size.

Networks from PyTorch 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 (400 -> 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)
)
```

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_clsid_to_human.txt', 'r').read())

# Load and normalize the image
img = torchvision.transforms.ToTensor()(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(0, 15):
    print('#{:d} ({:.02f}) {:s}'.format(k+1, scores[k].item(), class_names[indexes[k].
          item()]))
```

```
#1 (12.26) Weimaraner
#2 (10.95) Chesapeake Bay retriever
#3 (10.87) Labrador retriever
#4 (10.10) Staffordshire bullterrier, Staffordshire bull terrier
#5 (9.55) flat-coated retriever
#6 (9.40) Italian greyhound
#7 (9.31) American Staffordshire terrier, Staffordshire terrier, American pit bull
      terrier, pit bull terrier
#8 (9.12) Great Dane
#9 (8.94) German short-haired pointer
#10 (8.53) Doberman, Doberman pinscher
#11 (8.35) Rottweiler
#12 (8.25) kelpie
#13 (8.24) barrow, garden cart, lawn cart, wheelbarrow
#14 (8.12) bucket, pail
#15 (8.07) soccer ball
```

Weimaraner

Chesapeake Bay retriever

References

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.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

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