EE-559 – Deep learning

4b. PyTorch modules, batch processing

François Fleuret
https://fleuret.org/dlc/
February 15, 2018

`torch.nn.Module`

PyTorch provides a vast collection of “modules” which can be combined into complicated architectures. We will look at components to build our first convolutional neural network:

- `torch.nn.functional.relu`
- `torch.nn.functional.max_pool2d`
- `torch.nn.Conv2d`
- `torch.nn.Linear`
- `torch.nn.MSELoss`

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

Modules from `torch.nn` are losses and components for networks. The latter embed `torch.nn.Parameter`s to be optimized during training.

⚠ Since they are almost exclusively used with autograd, PyTorch’s `Module`s can only process `Variable`s.

We use the term “tensor” for both `Tensor`s and `Variable`s in what follows.

```
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.

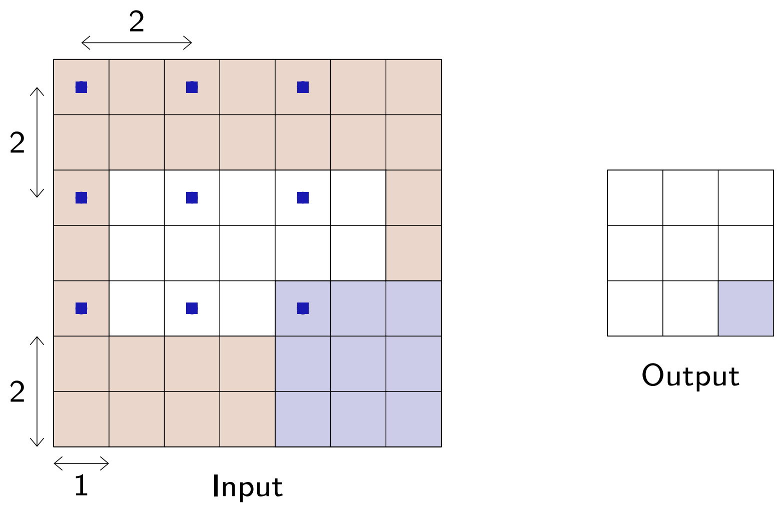
```
>>> x = Variable(Tensor(2, 5).normal_())
>>> x
Variable containing:
-0.2066 -1.7997 -0.0653  0.6481  0.0253
 1.0239  3.0324  1.6431 -1.8925  0.0890
[torch.FloatTensor of size 2x5]

>>> torch.nn.functional.relu(x)
Variable containing:
 0.0000  0.0000  0.0000  0.6481  0.0253
 1.0239  3.0324  1.6431  0.0000  0.0890
[torch.FloatTensor of size 2x5]
```

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

Both pooling and 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 $C \times 3 \times 3$, the output is $1 \times 3 \times 3$.



Pooling operations have a default stride equal to their kernel size, and convolutions have a default stride of 1. Padding can be useful for instance to generate an output of same size as the input.

```
torch.nn.functional.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 = Variable(Tensor(2, 2, 6).random_(3))
>>> x
Variable containing:
(0 ,.,.) =
  0  0  2  0  0  1
  2  1  1  2  0  1

(1 ,.,.) =
  2  0  0  2  2  0
  2  1  1  0  1  0
[torch.FloatTensor of size 2x2x6]

>>> torch.nn.functional.max_pool2d(x, (1, 2))
Variable containing:
(0 ,.,.) =
  0  2  1
  2  2  1

(1 ,.,.) =
  2  2  2
  2  1  1
[torch.FloatTensor of size 2x2x3]
```

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

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

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

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

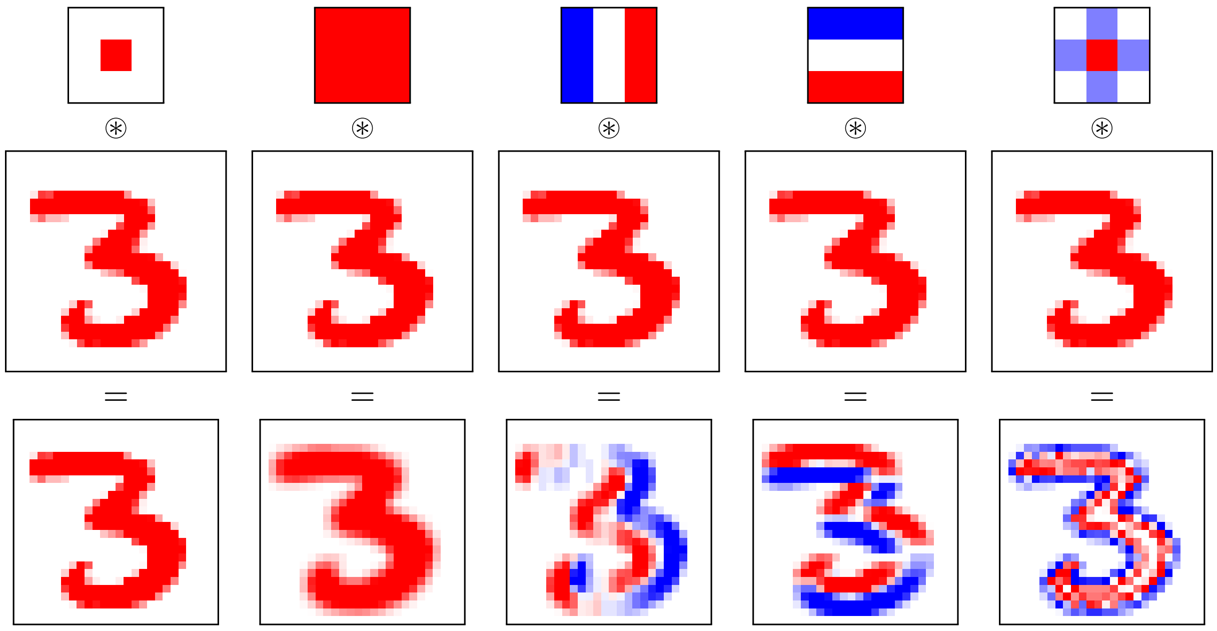
Implements a standard 2d convolutional layer.

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.

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

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

```
mnist_train = datasets.MNIST('./data/mnist/', train = True, download = True)

x = mnist_train.train_data.narrow(0, 12, 1).float()

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

f.bias.data.zero_()

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

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

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

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

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

y = f(Variable(x.view(1, x.size(0), x.size(1), x.size(2)))).data

save_2d_tensor_as_image(f.weight.data.squeeze(), 'conv-filters-{:d}.png',
                        signed = True)

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

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



`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 = Variable(Tensor([ 3 ]))
>>> y = Variable(Tensor([ 0 ]))
>>> f(x, y)
Variable containing:
 9
[torch.FloatTensor of size 1]

>>> x = Variable(Tensor([ 3, 0, 0 ]))
>>> y = Variable(Tensor([ 0, 0, 0 ]))
>>> f(x, y)
Variable containing:
 3
[torch.FloatTensor of size 1]

>>> x = Variable(Tensor([ 3, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]))
>>> y = Variable(Tensor([ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]))
>>> f(x, y)
Variable containing:
 0.9000
[torch.FloatTensor of size 1]
```

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 type for some losses (*e.g.* for classification).

Remember that `Module`s' inputs and outputs are `Variable`s. Data `Tensor`s should be wrapped before forwarding them into a `Module`.

```
>>> import torchvision
>>> mnist = torchvision.datasets.MNIST('./data/mnist/')
>>> x = mnist.train_data.float()
>>> x = x.view(x.size(0), -1)
>>> l = nn.Linear(x.size(1), 10)
>>> y = l(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/fleuret/misc/miniconda3/lib/python3.6/site-packages/torch/nn/modules/
        module.py", line 325, in __call__
    result = self.forward(*input, **kwargs)
  File "/home/fleuret/misc/miniconda3/lib/python3.6/site-packages/torch/nn/modules/
        linear.py", line 55, in forward
    return F.linear(input, self.weight, self.bias)
  File "/home/fleuret/misc/miniconda3/lib/python3.6/site-packages/torch/nn/functional.
        py", line 835, in linear
    return torch.addmm(bias, input, weight.t())
RuntimeError: addmm(): argument 'mat1' (position 1) must be Variable, not torch.
      FloatTensor
>>> x = torch.autograd.Variable(x)
>>> y = l(x)
```

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

```
>>> f = torch.nn.MSELoss()
>>> x = Variable(Tensor([ 3, 2 ]), requires_grad = True)
>>> y = Variable(Tensor([ 0, -2 ]), requires_grad = True)
>>> f(x, y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/fleuret/misc/miniconda3/lib/python3.6/site-packages/torch/nn/modules/
        module.py", line 325, in __call__
    result = self.forward(*input, **kwargs)
  File "/home/fleuret/misc/miniconda3/lib/python3.6/site-packages/torch/nn/modules/loss
        .py", line 328, in forward
    _assert_no_grad(target)
  File "/home/fleuret/misc/miniconda3/lib/python3.6/site-packages/torch/nn/modules/loss
        .py", line 11, in _assert_no_grad
    "nn criterions don't compute the gradient w.r.t. targets - please " \
AssertionError: nn criterions don't compute the gradient w.r.t. targets - please mark
      these variables as volatile or not requiring gradients
```

We can use the MSE loss for training, even though this is classification.

To do so, given a training set

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

we will consider an output with as many units as there are classes, and the target will be a tensor $z \in \mathbb{R}^{N \times C}$, with -1 everywhere but for the correct labels:

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

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

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

Although MSE is a regression loss, using it like this gives excellent results.

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.

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])
```

Batch processing

PyTorch's `Module`s take as input a batch of samples, that is a tensor whose first index is the sample's index.

However we have formalized the fully connected layers and back-prop with column vectors *e.g.*

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

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

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

which gives a tensorial expression for the full batch

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

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

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

and

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

Batch processing allows to use efficient matrix product implementations, which in particular deal properly with cache memory

```
import torch, time

def timing(x, w, nb = 51):
    t = torch.FloatTensor(nb)

    for u in range(t.size(0)):
        t0 = time.perf_counter()
        y = x.mm(w.t())
        y.is_cuda and torch.cuda.synchronize()
        t[u] = time.perf_counter() - t0
    tb = t.median()[0][0]

    for u in range(t.size(0)):
        t0 = time.perf_counter()
        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
    tl = t.median()[0][0]

    print('{:s} batch vs. loop speed ratio {:.01f}'
          .format((y.is_cuda and 'GPU') or 'CPU', tl / tb))

x = torch.FloatTensor(2500, 1000).normal_()
w = torch.FloatTensor(1500, 1000).normal_()
timing(x, w)

x = torch.cuda.FloatTensor(2500, 1000).normal_()
w = torch.cuda.FloatTensor(1500, 1000).normal_()
timing(x, w)
```

Prints:

```
CPU batch vs. loop speed ratio 10.0
GPU batch vs. loop speed ratio 80.7
```

Practical session:

https://fleuret.org/dlc/dlc-practical-4.pdf