EE-559 – Deep learning

1b. PyTorch Tensors

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

[version of: June 14, 2018]

RESEARCH INSTITUTE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

PyTorch's tensors

A tensor is a generalized matrix, a finite table of numerical values indexed along several discrete dimensions.

A tensor is a generalized matrix, a finite table of numerical values indexed along several discrete dimensions.

- 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 is a vector of identically sized matrices (*e.g.* a multi-channel image),
- A 4d tensor is a matrix of identically sized matrices (*e.g.* a sequence of multi-channel images),
- etc.

A tensor is a generalized matrix, a finite table of numerical values indexed along several discrete dimensions.

- 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 is a vector of identically sized matrices (*e.g.* a multi-channel image),
- A 4d tensor is a matrix of identically sized matrices (*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 the "neural networks".

A tensor is a generalized matrix, a finite table of numerical values indexed along several discrete dimensions.

- 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 is a vector of identically sized matrices (*e.g.* a multi-channel image),
- A 4d tensor is a matrix of identically sized matrices (*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 the "neural networks".

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

A tensor is a generalized matrix, a finite table of numerical values indexed along several discrete dimensions.

- 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 is a vector of identically sized matrices (*e.g.* a multi-channel image),
- A 4d tensor is a matrix of identically sized matrices (*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 the "neural networks".

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

Compounded data structures can represent more diverse data types.

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.

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.)

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: tensor operations are specified dynamically as Python operations. We will come back to this.

```
>>> from torch import Tensor
>>> x = Tensor(5)
>>> x.size()
torch.Size([5])
>>> x.fill_(1.125)

 1.1250
 1.1250
 1.1250
 1.1250
 1.1250
[torch.FloatTensor of size 5]

>>> x.sum()
5.625
>>> x.mean()
1.125
>>> x.std()
0.0
```

```
>>> from torch import Tensor
>>> x = Tensor(5)
>>> x.size()
torch.Size([5])
>>> x.fill_(1.125)

 1.1250
 1.1250
 1.1250
 1.1250
 1.1250
[torch.FloatTensor of size 5]

>>> x.sum()
5.625
>>> x.mean()
1.125
>>> x.std()
0.0
```

```
>>> from torch import Tensor
>>> x = Tensor(5)
>>> x.size()
torch.Size([5])
>>> x.fill_(1.125)

 1.1250
 1.1250
 1.1250
 1.1250
 1.1250
[torch.FloatTensor of size 5]

>>> x.sum()
5.625
>>> x.mean()
1.125
>>> x.std()
0.0
```

```
>>> from torch import Tensor
>>> x = Tensor(5)
>>> x.size()
torch.Size([5])
>>> x.fill_(1.125)

 1.1250
 1.1250
 1.1250
 1.1250
 1.1250
[torch.FloatTensor of size 5]

>>> x.sum()
5.625
>>> x.mean()
1.125
>>> x.std()
0.0
```

```
>>> from torch import Tensor
>>> x = Tensor(5)
>>> x.size()
torch.Size([5])
>>> x.fill_(1.125)

 1.1250
 1.1250
 1.1250
 1.1250
 1.1250
[torch.FloatTensor of size 5]

>>> x.sum()
5.625
>>> x.mean()
1.125
>>> x.std()
0.0
```

The default tensor type `torch.Tensor` is an alias for `torch.FloatTensor`, but there are others with greater/lesser precision and on CPU/GPU.

It can be set to a different type with `torch.set_default_tensor_type`

```
>>> from torch import Tensor
>>> x = Tensor(5)
>>> x.size()
torch.Size([5])
>>> x.fill_(1.125)

 1.1250
 1.1250
 1.1250
 1.1250
 1.1250
[torch.FloatTensor of size 5]

>>> x.sum()
5.625
>>> x.mean()
1.125
>>> x.std()
0.0
```

The default tensor type `torch.Tensor` is an alias for `torch.FloatTensor`, but there are others with greater/lesser precision and on CPU/GPU.

It can be set to a different type with `torch.set_default_tensor_type`

In-place operations are suffixed with an underscore.

`torch.Tensor.narrow` creates a new tensor which is a sub-part of an existing tensor, by constraining one of the indexes. **It shares its content with the original tensor, and modifying one modifies the other.**

```
>>> a = Tensor(4, 5).zero_()
>>> a

 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
[torch.FloatTensor of size 4x5]

>>> a.narrow(1, 2, 2).fill_(1.0)

 1  1
 1  1
 1  1
 1  1
[torch.FloatTensor of size 4x2]

>>> a

 0  0  1  1  0
 0  0  1  1  0
 0  0  1  1  0
 0  0  1  1  0
[torch.FloatTensor of size 4x5]
```

`torch.Tensor.narrow` creates a new tensor which is a sub-part of an existing tensor, by constraining one of the indexes. **It shares its content with the original tensor, and modifying one modifies the other.**

```
>>> a = Tensor(4, 5).zero_()
>>> a

 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
[torch.FloatTensor of size 4x5]

>>> a.narrow(1, 2, 2).fill_(1.0)

 1  1
 1  1
 1  1
 1  1
[torch.FloatTensor of size 4x2]

>>> a

 0  0  1  1  0
 0  0  1  1  0
 0  0  1  1  0
 0  0  1  1  0
[torch.FloatTensor of size 4x5]
```

`torch.Tensor.narrow` creates a new tensor which is a sub-part of an existing tensor, by constraining one of the indexes. **It shares its content with the original tensor, and modifying one modifies the other.**

```
>>> a = Tensor(4, 5).zero_()
>>> a

 0   0   0   0   0
 0   0   0   0   0
 0   0   0   0   0
 0   0   0   0   0
[torch.FloatTensor of size 4x5]

>>> a.narrow(1, 2, 2).fill_(1.0)

 1   1
 1   1
 1   1
 1   1
[torch.FloatTensor of size 4x2]

>>> a

 0   0   1   1   0
 0   0   1   1   0
 0   0   1   1   0
 0   0   1   1   0
[torch.FloatTensor of size 4x5]
```

`torch.Tensor.narrow` creates a new tensor which is a sub-part of an existing tensor, by constraining one of the indexes. **It shares its content with the original tensor, and modifying one modifies the other.**

```
>>> a = Tensor(4, 5).zero_()
>>> a

 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
[torch.FloatTensor of size 4x5]

>>> a.narrow(1, 2, 2).fill_(1.0)

 1  1
 1  1
 1  1
 1  1
[torch.FloatTensor of size 4x2]

>>> a

 0  0  1  1  0
 0  0  1  1  0
 0  0  1  1  0
 0  0  1  1  0
[torch.FloatTensor of size 4x5]
```

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

```
>>> y = Tensor(3).normal_()
>>> y

-1.6978
-0.6911
-1.1713
[torch.FloatTensor of size 3]

>>> m = Tensor(3, 3).normal_()
>>> q, _ = torch.gels(y, m)
>>> torch.mm(m, q)

-1.6978
-0.6911
-1.1713
[torch.FloatTensor of size 3x1]
```

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

```
>>> y = Tensor(3).normal_()
>>> y

-1.6978
-0.6911
-1.1713
[torch.FloatTensor of size 3]

>>> m = Tensor(3, 3).normal_()
>>> q, _ = torch.gels(y, m)
>>> torch.mm(m, q)

-1.6978
-0.6911
-1.1713
[torch.FloatTensor of size 3x1]
```

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

```
>>> y = Tensor(3).normal_()
>>> y

-1.6978
-0.6911
-1.1713
[torch.FloatTensor of size 3]

>>> m = Tensor(3, 3).normal_()
>>> q, _ = torch.gels(y, m)
>>> torch.mm(m, q)

-1.6978
-0.6911
-1.1713
[torch.FloatTensor of size 3x1]
```

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

```
>>> y = Tensor(3).normal_()
>>> y

-1.6978
-0.6911
-1.1713
[torch.FloatTensor of size 3]

>>> m = Tensor(3, 3).normal_()
>>> q, _ = torch.gels(y, m)
>>> torch.mm(m, q)

-1.6978
-0.6911
-1.1713
[torch.FloatTensor of size 3x1]
```

Example: linear regression

Given a list of points

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

can we find the "best line"

$$f(x; a, b) = ax + b$$

going "through the points"

Given a list of points

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

can we find the "best line"

$$f(x; a, b) = ax + b$$

going "through the points", *e.g.* minimizing the mean square error

$$\underset{a,b}{\operatorname{argmin}} \frac{1}{N} \sum_{n=1}^{N} \big(\underbrace{a x_n + b}_{f(x_n; a, b)} - y_n \big)^2 .$$

Given a list of points

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

can we find the "best line"

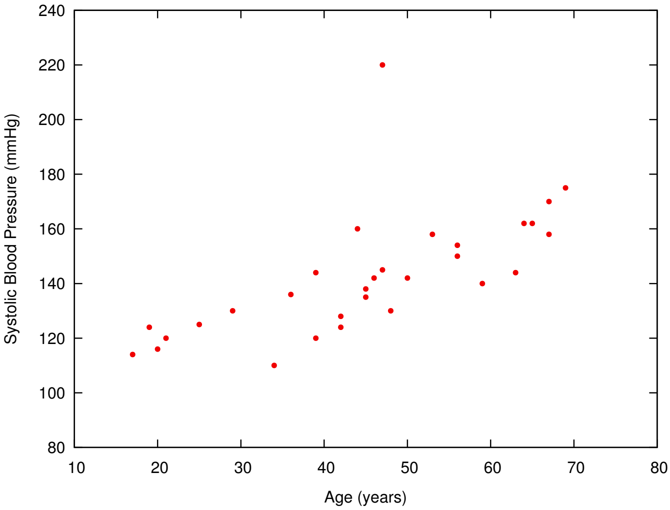
$$f(x; a, b) = ax + b$$

going "through the points", *e.g.* minimizing the mean square error

$$\underset{a,b}{\operatorname{argmin}} \frac{1}{N} \sum_{n=1}^{N} \big(\underbrace{ax_n + b}_{f(x_n;a,b)} - y_n \big)^2.$$

Such a model would allow to predict the y associated to a new x, simply by calculating $f(x; a, b)$.

```
bash> cat systolic-blood-pressure-vs-age.dat
39  144
47  220
45  138
47  145
65  162
46  142
67  170
42  124
67  158
56  154
64  162
56  150
59  140
34  110
42  128
48  130
45  135
17  114
20  116
19  124
36  136
50  142
39  120
21  120
44  160
53  158
63  144
29  130
25  125
69  175
```



240
220
200
180
160
140
120
100
80
10
20
30
40
50
60
70
80
Systolic Blood Pressure (mmHg)
Age (years)

$$\underbrace{\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{pmatrix}}_{\mathrm{data} \in \mathbb{R}^{N \times 2}}$$

$$\underbrace{\begin{pmatrix} x_1 & 1.0 \\ x_2 & 1.0 \\ \vdots & \vdots \\ x_N & 1.0 \end{pmatrix}}_{x \in \mathbb{R}^{N \times 2}} \underbrace{\begin{pmatrix} a \\ b \end{pmatrix}}_{\alpha \in \mathbb{R}^{2 \times 1}} \simeq \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}}_{y \in \mathbb{R}^{N \times 1}}$$

$$\underbrace{\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{pmatrix}}_{\text{data} \in \mathbb{R}^{N \times 2}} \qquad \underbrace{\begin{pmatrix} x_1 & 1.0 \\ x_2 & 1.0 \\ \vdots & \vdots \\ x_N & 1.0 \end{pmatrix}}_{x \in \mathbb{R}^{N \times 2}} \underbrace{\begin{pmatrix} a \\ b \end{pmatrix}}_{\alpha \in \mathbb{R}^{2 \times 1}} \simeq \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}}_{y \in \mathbb{R}^{N \times 1}}$$

```
import torch, numpy

data = torch.from_numpy(numpy.loadtxt('systolic-blood-pressure-vs-age.dat')).float()

nb = data.size(0)

x, y = torch.Tensor(nb, 2), torch.Tensor(nb, 1)
x[:,0] = data[:,0]
x[:,1] = 1
y[:,0] = data[:,1]

alpha, _ = torch.gels(y, x)

a, b = alpha[0,0], alpha[1, 0]
```

$$\underbrace{\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{pmatrix}}_{\text{data} \in \mathbb{R}^{N \times 2}}$$

$$\underbrace{\begin{pmatrix} x_1 & 1.0 \\ x_2 & 1.0 \\ \vdots & \vdots \\ x_N & 1.0 \end{pmatrix}}_{x \in \mathbb{R}^{N \times 2}} \underbrace{\begin{pmatrix} a \\ b \end{pmatrix}}_{\alpha \in \mathbb{R}^{2 \times 1}} \simeq \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}}_{y \in \mathbb{R}^{N \times 1}}$$

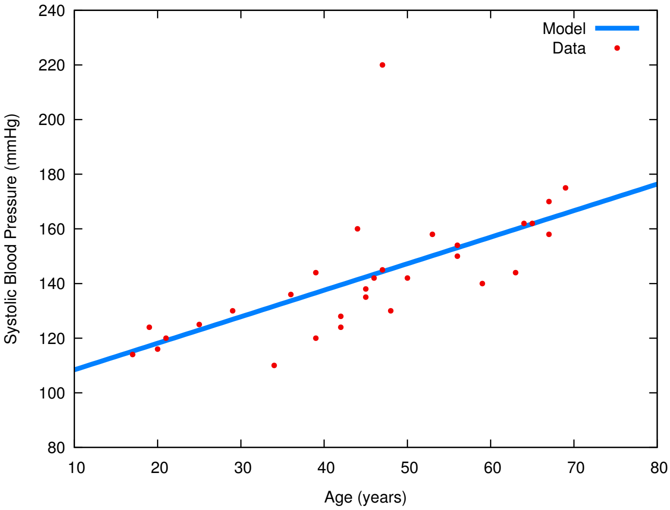
```
import torch, numpy

data = torch.from_numpy(numpy.loadtxt('systolic-blood-pressure-vs-age.dat')).float()

nb = data.size(0)

x, y = torch.Tensor(nb, 2), torch.Tensor(nb, 1)
x[:,0] = data[:,0]
x[:,1] = 1
y[:,0] = data[:,1]

alpha, _ = torch.gels(y, x)

a, b = alpha[0,0], alpha[1, 0]
```

$$\underbrace{\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{pmatrix}}_{\text{data} \in \mathbb{R}^{N \times 2}} \qquad \underbrace{\begin{pmatrix} x_1 & 1.0 \\ x_2 & 1.0 \\ \vdots & \vdots \\ x_N & 1.0 \end{pmatrix}}_{x \in \mathbb{R}^{N \times 2}} \underbrace{\begin{pmatrix} a \\ b \end{pmatrix}}_{\alpha \in \mathbb{R}^{2 \times 1}} \simeq \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}}_{y \in \mathbb{R}^{N \times 1}}$$

```
import torch, numpy

data = torch.from_numpy(numpy.loadtxt('systolic-blood-pressure-vs-age.dat')).float()

nb = data.size(0)

x, y = torch.Tensor(nb, 2), torch.Tensor(nb, 1)
x[:,0] = data[:,0]
x[:,1] = 1
y[:,0] = data[:,1]

alpha, _ = torch.gels(y, x)

a, b = alpha[0,0], alpha[1, 0]
```

$$\underbrace{\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{pmatrix}}_{\text{data} \in \mathbb{R}^{N \times 2}} \qquad \underbrace{\begin{pmatrix} x_1 & 1.0 \\ x_2 & 1.0 \\ \vdots & \vdots \\ x_N & 1.0 \end{pmatrix}}_{x \in \mathbb{R}^{N \times 2}} \underbrace{\begin{pmatrix} a \\ b \end{pmatrix}}_{\alpha \in \mathbb{R}^{2 \times 1}} \simeq \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}}_{y \in \mathbb{R}^{N \times 1}}$$

```
import torch, numpy

data = torch.from_numpy(numpy.loadtxt('systolic-blood-pressure-vs-age.dat')).float()

nb = data.size(0)

x, y = torch.Tensor(nb, 2), torch.Tensor(nb, 1)
x[:,0] = data[:,0]
x[:,1] = 1
y[:,0] = data[:,1]

alpha, _ = torch.gels(y, x)

a, b = alpha[0,0], alpha[1, 0]
```


Model
Data
Systolic Blood Pressure (mmHg)
Age (years)
240
220
200
180
160
140
120
100
80
10
20
30
40
50
60
70
80

Manipulating high-dimension signals

Data type	CPU tensor	GPU tensor
32-bit float	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit float	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
8-bit int (unsigned)	`torch.ByteTensor`	`torch.cuda.ByteTensor`
64-bit int (signed)	`torch.LongTensor`	`torch.cuda.LongTensor`

Data type	CPU tensor	GPU tensor
32-bit float	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit float	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
8-bit int (unsigned)	`torch.ByteTensor`	`torch.cuda.ByteTensor`
64-bit int (signed)	`torch.LongTensor`	`torch.cuda.LongTensor`

```
>>> x = torch.LongTensor(12)
>>> type(x)
<class 'torch.LongTensor'>
>>> x = x.float()
>>> type(x)
<class 'torch.FloatTensor'>
>>> x = x.cuda()
>>> type(x)
<class 'torch.cuda.FloatTensor'>
```

Data type	CPU tensor	GPU tensor
32-bit float	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit float	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
8-bit int (unsigned)	`torch.ByteTensor`	`torch.cuda.ByteTensor`
64-bit int (signed)	`torch.LongTensor`	`torch.cuda.LongTensor`

```
>>> x = torch.LongTensor(12)
>>> type(x)
<class 'torch.LongTensor'>
>>> x = x.float()
>>> type(x)
<class 'torch.FloatTensor'>
>>> x = x.cuda()
>>> type(x)
<class 'torch.cuda.FloatTensor'>
```

Tensors of the `torch.cuda` types are physically in the GPU memory, and operations on them are done by the GPU.

Data type	CPU tensor	GPU tensor
32-bit float	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit float	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
8-bit int (unsigned)	`torch.ByteTensor`	`torch.cuda.ByteTensor`
64-bit int (signed)	`torch.LongTensor`	`torch.cuda.LongTensor`

```
>>> x = torch.LongTensor(12)
>>> type(x)
<class 'torch.LongTensor'>
>>> x = x.float()
>>> type(x)
<class 'torch.FloatTensor'>
>>> x = x.cuda()
>>> type(x)
<class 'torch.cuda.FloatTensor'>
```

Tensors of the `torch.cuda` types are physically in the GPU memory, and operations on them are done by the GPU. We will come back to that later.

The default tensor type can be set with `torch.set_default_tensor_type` and used through `torch.Tensor`

```
>>> x = torch.Tensor()
>>> x
[torch.FloatTensor with no dimension]

>>> torch.set_default_tensor_type('torch.LongTensor')
>>> x = torch.Tensor()
>>> x
[torch.LongTensor with no dimension]
```

The default tensor type can be set with `torch.set_default_tensor_type` and used through `torch.Tensor`

```
>>> x = torch.Tensor()
>>> x
[torch.FloatTensor with no dimension]

>>> torch.set_default_tensor_type('torch.LongTensor')
>>> x = torch.Tensor()
>>> x
[torch.LongTensor with no dimension]
```

For concision we often start our code with `from torch import Tensor` and use `Tensor` in place of `torch.Tensor`.

The default tensor type can be set with `torch.set_default_tensor_type` and used through `torch.Tensor`

```
>>> x = torch.Tensor()
>>> x
[torch.FloatTensor with no dimension]

>>> torch.set_default_tensor_type('torch.LongTensor')
>>> x = torch.Tensor()
>>> x
[torch.LongTensor with no dimension]
```

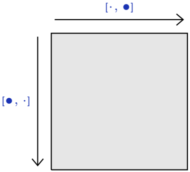
For concision we often start our code with `from torch import Tensor` and use `Tensor` in place of `torch.Tensor`.

Also, the `new` operator of a tensor allows to create one of same type

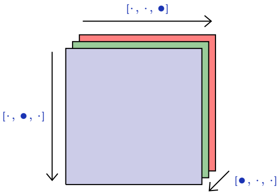
```
>>> y = torch.ByteTensor(10)
>>> u = y.new(3).fill_(123)
>>> u

 123
 123
 123
[torch.ByteTensor of size 3]
```

This is key to writing functions able to handle all the tensor types.

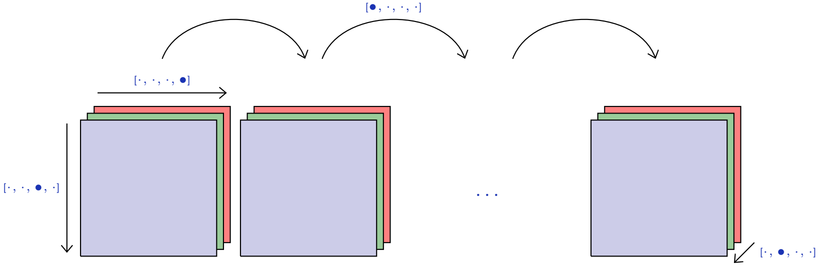
2d tensor (*e.g.* grayscale image)



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.Tensor()`
- `torch.Tensor(size)`
- `torch.Tensor(sequence)`
- `torch.eye(n)`
- `torch.from_numpy(ndarray)`

Indexing, Slicing, Joining, Mutating

- `torch.Tensor.view(*args)`
- `torch.Tensor.expand(*sizes)`
- `torch.cat(inputs, dimension=0)`
- `torch.chunk(tensor, chunks, 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(input, out=None)`
- `torch.normal()`

Pointwise math

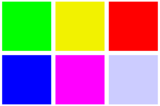
- `torch.abs(input, out=None)`
- `torch.add()`
- `torch.cos(input, out=None)`
- `torch.sigmoid(input, out=None)`
- `(+ many operators)`

Math reduction

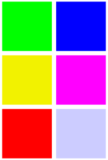
- `torch.dist(input, other, p=2, out=None)`
- `torch.mean()`
- `torch.norm()`
- `torch.std()`
- `torch.sum()`

BLAS and LAPACK Operations

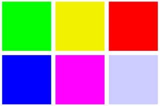
- `torch.eig(a, eigenvectors=False, out=None)`
- `torch.gels(B, A, out=None)`
- `torch.inverse(input, out=None)`
- `torch.mm(mat1, mat2, out=None)`
- `torch.mv(mat, vec, out=None)`



```
x = torch.LongTensor([ [ 1, 3, 0 ],
                       [ 2, 4, 6 ] ])
```

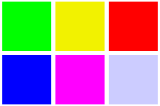


```
x.t()
```

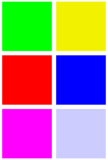


```
x = torch.LongTensor([ [ 1, 3, 0 ],
                       [ 2, 4, 6 ] ])
```

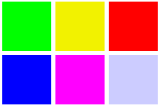
`x.view(-1)`



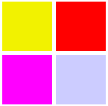
```
x = torch.LongTensor([ [ 1, 3, 0 ],
                       [ 2, 4, 6 ] ])
```



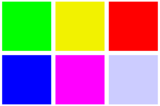
```
x.view(3, -1)
```



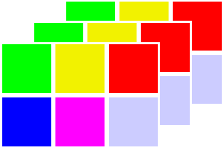
```
x = torch.LongTensor([ [ 1, 3, 0 ],
                       [ 2, 4, 6 ] ])
```



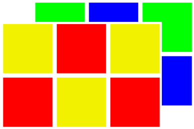
```
x.narrow(1, 1, 2)
```



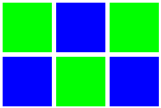
```
x = torch.LongTensor([ [ 1, 3, 0 ],
                       [ 2, 4, 6 ] ])
```



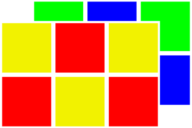
```
x.view(1, 2, 3).expand(3, 2, 3)
```



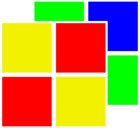
```
x = torch.LongTensor([ [ [ 1, 2, 1 ],
                         [ 2, 1, 2 ] ],
                       [ [ 3, 0, 3 ],
                         [ 0, 3, 0 ] ] ])
```



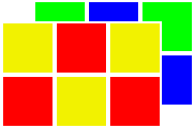
```
x.narrow(0, 0, 1)
```



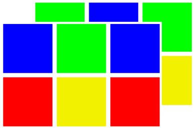
```
x = torch.LongTensor([ [ [ 1, 2, 1 ],
                         [ 2, 1, 2 ] ],
                       [ [ 3, 0, 3 ],
                         [ 0, 3, 0 ] ] ])
```



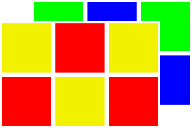
```
x.narrow(2, 0, 2)
```



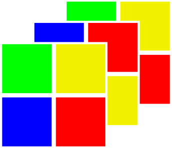
```
x = torch.LongTensor([ [ [ 1, 2, 1 ],
                         [ 2, 1, 2 ] ],
                       [ [ 3, 0, 3 ],
                         [ 0, 3, 0 ] ] ])
```



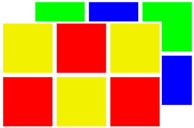
```
x.transpose(0, 1)
```



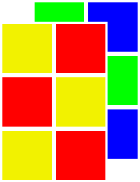
```
x = torch.LongTensor([ [ [ 1, 2, 1 ],
                         [ 2, 1, 2 ] ],
                       [ [ 3, 0, 3 ],
                         [ 0, 3, 0 ] ] ])
```



```
x.transpose(0, 2)
```



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



`x.transpose(1, 2)`

PyTorch offers simple interfaces to standard image data-bases.

```
import torch
import torchvision

# Get the CIFAR10 train images, download if necessary
cifar = torchvision.datasets.CIFAR10('./data/cifar10/', train=True, download=True)

# Converts the numpy tensor into a PyTorch one
x = torch.from_numpy(cifar.train_data).transpose(1, 3).transpose(2, 3)

# Prints out some info
print(str(type(x)), x.size(), x.min(), x.max())
```

PyTorch offers simple interfaces to standard image data-bases.

```
import torch
import torchvision

# Get the CIFAR10 train images, download if necessary
cifar = torchvision.datasets.CIFAR10('./data/cifar10/', train=True, download=True)

# Converts the numpy tensor into a PyTorch one
x = torch.from_numpy(cifar.train_data).transpose(1, 3).transpose(2, 3)

# Prints out some info
print(str(type(x)), x.size(), x.min(), x.max())
```

prints

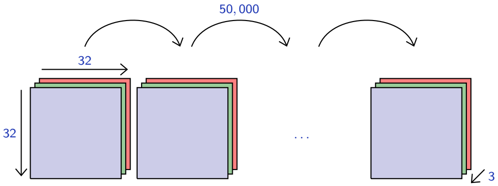
```
Files already downloaded and verified
<class 'torch.ByteTensor'> torch.Size([50000, 3, 32, 32]) 0 255
```

PyTorch offers simple interfaces to standard image data-bases.

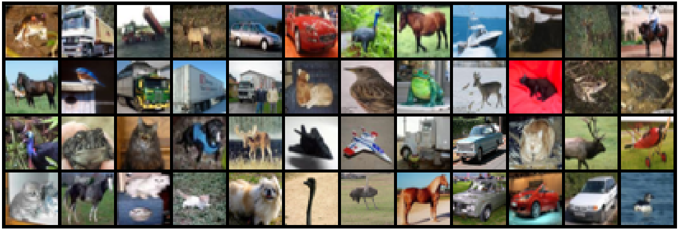
```
import torch
import torchvision

# Get the CIFAR10 train images, download if necessary
cifar = torchvision.datasets.CIFAR10('./data/cifar10/', train=True, download=True)

# Converts the numpy tensor into a PyTorch one
x = torch.from_numpy(cifar.train_data).transpose(1, 3).transpose(2, 3)

# Prints out some info
print(str(type(x)), x.size(), x.min(), x.max())
```

prints

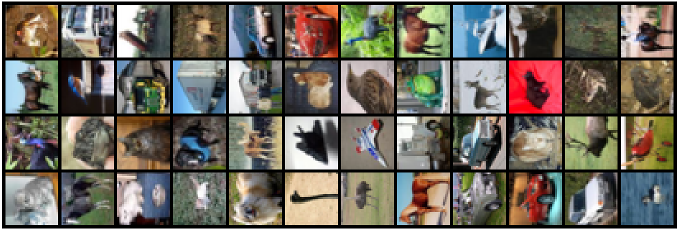
```
Files already downloaded and verified
<class 'torch.ByteTensor'> torch.Size([50000, 3, 32, 32]) 0 255
```



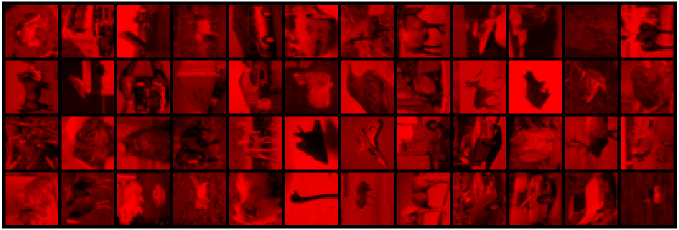
```
# Narrow to the first images, make the tensor Float, and move the
# values in [-1, 1]
x = x.narrow(0, 0, 48).float().div(255)

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



```
# Switch the row and column indexes
x.transpose_(2, 3)

torchvision.utils.save_image(x, 'images-cifar-4x12-rotated.png', nrow = 12)
```



```
# Kill the green (1) and blue (2) channels
x.narrow(1, 1, 2).fill_(-1)

torchvision.utils.save_image(x, 'images-cifar-4x12-rotated-and-red.png', nrow = 12)
```



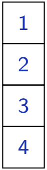
Broadcasting

Broadcasting automagically expands dimensions of size 1 by replicating coefficients, when it is necessary to perform operations.

Broadcasting automagically expands dimensions of size 1 by replicating coefficients, when it is necessary to perform operations.

```
>>> A = Tensor([[1], [2], [3], [4]])
>>> A

 1
 2
 3
 4
[torch.FloatTensor of size 4x1]

>>> B = Tensor([[5, -5, 5, -5, 5]])
>>> B

 5 -5  5 -5  5
[torch.FloatTensor of size 1x5]

>>> C = A + B
>>> C

 6 -4  6 -4  6
 7 -3  7 -3  7
 8 -2  8 -2  8
 9 -1  9 -1  9
[torch.FloatTensor of size 4x5]
```

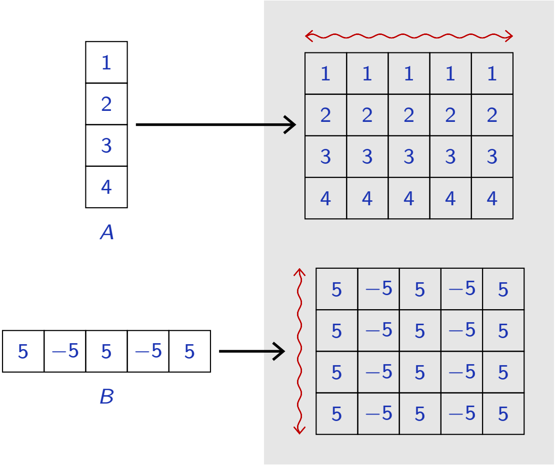
```
A = Tensor([[1], [2], [3], [4]])
B = Tensor([[5, -5, 5, -5, 5]])
C = A + B
```



A

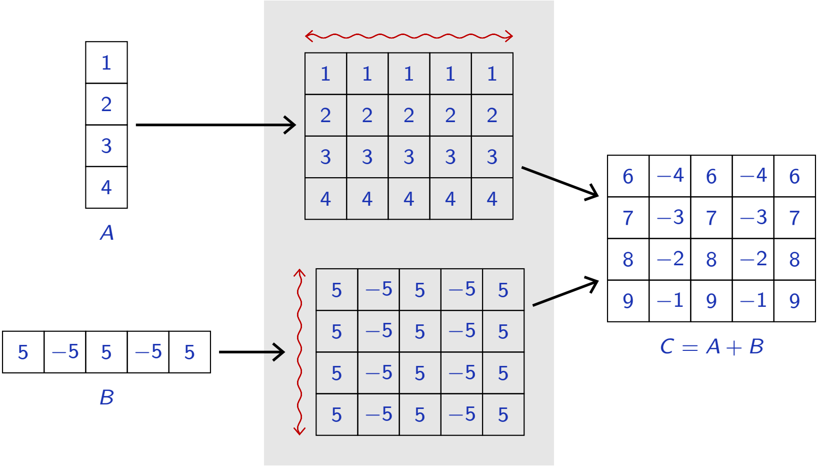
B

```
A = Tensor([[1], [2], [3], [4]])
B = Tensor([[5, -5, 5, -5, 5]])
C = A + B
```



Broadcasted

```
A = Tensor([[1], [2], [3], [4]])
B = Tensor([[5, -5, 5, -5, 5]])
C = A + B
```



Broadcasted

Precisely, broadcasting proceeds as follows:

1. If one of the tensors has fewer dimensions than the other, it is reshaped by adding as many dimensions of size 1 as necessary in the front; then
2. for every mismatch, **if one of the two sizes is one,** the tensor is expanded along this axis by replicating coefficients.

If there is a tensor size mismatch for one of the dimension and neither of them is one, the operation fails.

```
>>> x = Tensor([1, 2, 3, 4, 5])
>>> y = Tensor(3, 5).fill_(2.0)
>>> z = x + y
>>> z

 3  4  5  6  7
 3  4  5  6  7
 3  4  5  6  7
[torch.FloatTensor of size 3x5]
```

```
>>> x = Tensor([1, 2, 3, 4, 5])
>>> y = Tensor(3, 5).fill_(2.0)
>>> z = x + y
>>> z

 3  4  5  6  7
 3  4  5  6  7
 3  4  5  6  7
[torch.FloatTensor of size 3x5]
```

```
>>> a = Tensor(3, 1, 5).fill_(1.0)
>>> b = Tensor(1, 3, 5).fill_(2.0)
>>> c = a * b + a
>>> c

(0 ,.,.) =
  3  3  3  3  3
  3  3  3  3  3
  3  3  3  3  3

(1 ,.,.) =
  3  3  3  3  3
  3  3  3  3  3
  3  3  3  3  3

(2 ,.,.) =
  3  3  3  3  3
  3  3  3  3  3
  3  3  3  3  3
[torch.FloatTensor of size 3x3x5]
```

Tensor internals

A tensor is a view of a storage, which is a low-level 1d vector.

```
>>> q = Tensor(2, 4).zero_()
>>> q.storage()
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
[torch.FloatStorage of size 8]
>>> s = q.storage()
>>> s[4] = 1.0
>>> s
 0.0
 0.0
 0.0
 0.0
 1.0
 0.0
 0.0
 0.0
[torch.FloatStorage of size 8]
>>> q

 0  0  0  0
 1  0  0  0
[torch.FloatTensor of size 2x4]
```

A tensor is a view of a storage, which is a low-level 1d vector.

```
>>> q = Tensor(2, 4).zero_()
>>> q.storage()
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
[torch.FloatStorage of size 8]
>>> s = q.storage()
>>> s[4] = 1.0
>>> s
 0.0
 0.0
 0.0
 0.0
 1.0
 0.0
 0.0
 0.0
[torch.FloatStorage of size 8]
>>> q

 0   0   0   0
 1   0   0   0
[torch.FloatTensor of size 2x4]
```

A tensor is a view of a storage, which is a low-level 1d vector.

```
>>> q = Tensor(2, 4).zero_()
>>> q.storage()
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
[torch.FloatStorage of size 8]
>>> s = q.storage()
>>> s[4] = 1.0
>>> s
 0.0
 0.0
 0.0
 0.0
 1.0
 0.0
 0.0
 0.0
[torch.FloatStorage of size 8]
>>> q

 0   0   0   0
 1   0   0   0
[torch.FloatTensor of size 2x4]
```

A tensor is a view of a storage, which is a low-level 1d vector.

```
>>> q = Tensor(2, 4).zero_()
>>> q.storage()
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
[torch.FloatStorage of size 8]
>>> s = q.storage()
>>> s[4] = 1.0
>>> s
 0.0
 0.0
 0.0
 0.0
 1.0
 0.0
 0.0
 0.0
[torch.FloatStorage of size 8]
>>> q

 0  0  0  0
 1  0  0  0
[torch.FloatTensor of size 2x4]
```

Multiple tensors can share the same storage. It happens when using operations such as `narrow()`, `view()`, `expand()` or `transpose()`.

```
>>> r = q.view(2, 2, 2)
>>> r

(0 ,.,.) =
  0  0
  0  0

(1 ,.,.) =
  1  0
  0  0
[torch.FloatTensor of size 2x2x2]

>>> r[1, 1, 0] = 1.0
>>> q

 0  0  0  0
 1  0  1  0
[torch.FloatTensor of size 2x4]

>>> r.narrow(0, 1, 1).fill_(3.0)

(0 ,.,.) =
  3  3
  3  3
[torch.FloatTensor of size 1x2x2]

>>> q

 0  0  0  0
 3  3  3  3
[torch.FloatTensor of size 2x4]
```

Multiple tensors can share the same storage. It happens when using operations such as `narrow()`, `view()`, `expand()` or `transpose()`.

```
>>> r = q.view(2, 2, 2)
>>> r

(0 ,.,.) =
  0  0
  0  0

(1 ,.,.) =
  1  0
  0  0
[torch.FloatTensor of size 2x2x2]

>>> r[1, 1, 0] = 1.0
>>> q

 0  0  0  0
 1  0  1  0
[torch.FloatTensor of size 2x4]

>>> r.narrow(0, 1, 1).fill_(3.0)

(0 ,.,.) =
  3  3
  3  3
[torch.FloatTensor of size 1x2x2]

>>> q

 0  0  0  0
 3  3  3  3
[torch.FloatTensor of size 2x4]
```

Multiple tensors can share the same storage. It happens when using operations such as `narrow()`, `view()`, `expand()` or `transpose()`.

```
>>> r = q.view(2, 2, 2)
>>> r

(0 ,.,.) =
  0  0
  0  0

(1 ,.,.) =
  1  0
  0  0
[torch.FloatTensor of size 2x2x2]

>>> r[1, 1, 0] = 1.0
>>> q

 0  0  0  0
 1  0  1  0
[torch.FloatTensor of size 2x4]
```

```
>>> r.narrow(0, 1, 1).fill_(3.0)

(0 ,.,.) =
  3  3
  3  3
[torch.FloatTensor of size 1x2x2]

>>> q

 0  0  0  0
 3  3  3  3
[torch.FloatTensor of size 2x4]
```

The first coefficient of a tensor is the one at `storage_offset()` in `storage()`.

The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by 1, you have to move by `stride(k)` elements in the storage.

The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by 1, you have to move by `stride(k)` elements in the storage.

```
>>> q = torch.arange(0, 20).storage()
>>> x = torch.Tensor().set_(q, storage_offset = 5, size = (3, 2), stride = (4, 1))

>>> x

  5   6
  9  10
 13  14
[torch.FloatTensor of size 3x2]
```

The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by 1, you have to move by `stride(k)` elements in the storage.

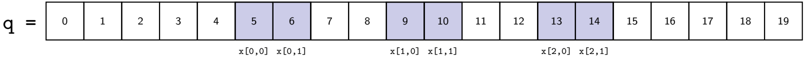
```
>>> q = torch.arange(0, 20).storage()
>>> x = torch.Tensor().set_(q, storage_offset = 5, size = (3, 2), stride = (4, 1))

>>> x

  5   6
  9  10
 13  14
[torch.FloatTensor of size 3x2]
```

q =

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

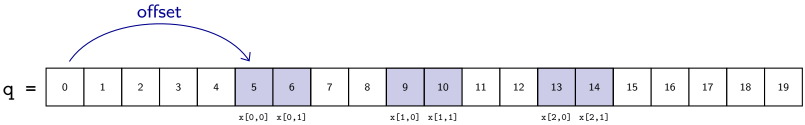
The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by 1, you have to move by `stride(k)` elements in the storage.

```
>>> q = torch.arange(0, 20).storage()
>>> x = torch.Tensor().set_(q, storage_offset = 5, size = (3, 2), stride = (4, 1))

>>> x

  5   6
  9  10
 13  14
[torch.FloatTensor of size 3x2]
```



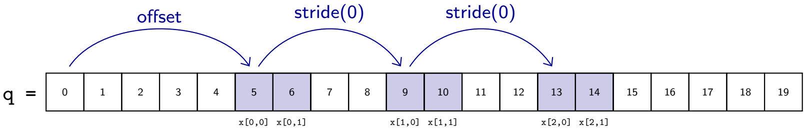
The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by 1, you have to move by `stride(k)` elements in the storage.

```
>>> q = torch.arange(0, 20).storage()
>>> x = torch.Tensor().set_(q, storage_offset = 5, size = (3, 2), stride = (4, 1))

>>> x

  5   6
  9  10
 13  14
[torch.FloatTensor of size 3x2]
```



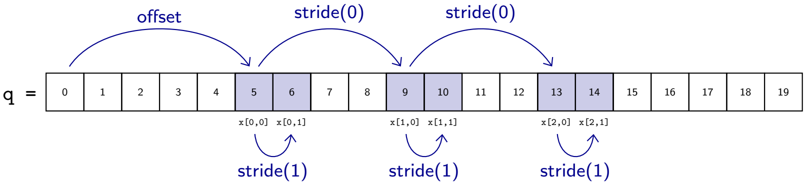
The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by 1, you have to move by `stride(k)` elements in the storage.

```
>>> q = torch.arange(0, 20).storage()
>>> x = torch.Tensor().set_(q, storage_offset = 5, size = (3, 2), stride = (4, 1))

>>> x

  5   6
  9  10
 13  14
[torch.FloatTensor of size 3x2]
```



The first coefficient of a tensor is the one at `storage_offset()` in `storage()`. To increment index `k` by 1, you have to move by `stride(k)` elements in the storage.

```
>>> q = torch.arange(0, 20).storage()
>>> x = torch.Tensor().set_(q, storage_offset = 5, size = (3, 2), stride = (4, 1))

>>> x

  5   6
  9  10
 13  14
[torch.FloatTensor of size 3x2]
```



We can explicitly create different "views" of the same storage

```
>>> n = torch.linspace(1, 4, 4)
>>> n

 1
 2
 3
 4
[torch.FloatTensor of size 4]

>>> Tensor().set_(n.storage(), 1, (3, 3), (0, 1))

 2  3  4
 2  3  4
 2  3  4
[torch.FloatTensor of size 3x3]

>>> Tensor().set_(n.storage(), 1, (2, 4), (1, 0))

 2  2  2  2
 3  3  3  3
[torch.FloatTensor of size 2x4]
```

We can explicitly create different “views” of the same storage

```
>>> n = torch.linspace(1, 4, 4)
>>> n

 1
 2
 3
 4
[torch.FloatTensor of size 4]
```

```
>>> Tensor().set_(n.storage(), 1, (3, 3), (0, 1))

 2  3  4
 2  3  4
 2  3  4
[torch.FloatTensor of size 3x3]

>>> Tensor().set_(n.storage(), 1, (2, 4), (1, 0))

 2  2  2  2
 3  3  3  3
[torch.FloatTensor of size 2x4]
```

We can explicitly create different “views” of the same storage

```
>>> n = torch.linspace(1, 4, 4)
>>> n

 1
 2
 3
 4
[torch.FloatTensor of size 4]

>>> Tensor().set_(n.storage(), 1, (3, 3), (0, 1))

 2  3  4
 2  3  4
 2  3  4
[torch.FloatTensor of size 3x3]

>>> Tensor().set_(n.storage(), 1, (2, 4), (1, 0))

 2  2  2  2
 3  3  3  3
[torch.FloatTensor of size 2x4]
```

We can explicitly create different “views” of the same storage

```
>>> n = torch.linspace(1, 4, 4)
>>> n

 1
 2
 3
 4
[torch.FloatTensor of size 4]

>>> Tensor().set_(n.storage(), 1, (3, 3), (0, 1))

 2  3  4
 2  3  4
 2  3  4
[torch.FloatTensor of size 3x3]

>>> Tensor().set_(n.storage(), 1, (2, 4), (1, 0))

 2  2  2  2
 3  3  3  3
[torch.FloatTensor of size 2x4]
```

We can explicitly create different "views" of the same storage

```
>>> n = torch.linspace(1, 4, 4)
>>> n

 1
 2
 3
 4
[torch.FloatTensor of size 4]

>>> Tensor().set_(n.storage(), 1, (3, 3), (0, 1))

 2  3  4
 2  3  4
 2  3  4
[torch.FloatTensor of size 3x3]

>>> Tensor().set_(n.storage(), 1, (2, 4), (1, 0))

 2  2  2  2
 3  3  3  3
[torch.FloatTensor of size 2x4]
```

This is in particular how transpositions and broadcasting are implemented.

This organization explains the following (maybe surprising) error

```
>>> x = Tensor(100, 100)
>>> y = x.t()
>>> y.view(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: input is not contiguous at /home/fleuret/misc/git/pytorch/torch/lib/TH/
      generic/THTensor.c:231
>>> y.stride()
(1, 100)
```

`t()` creates a tensor that shares the storage with the original tensor. It cannot be "flattened" into a 1d contiguous view without a memory copy.

Practical session:

https://fleuret.org/dlc/dlc-practical-1.pdf

The end