IfI Summer School 2018 on Machine Learning

Deep Learning #2 – PyTorch's Tensors

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

What is a tensor?

A tensor is a generalized matrix, 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 matrix, 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.

```
>>> import torch
>>> x = torch.zeros(5)
>>> x.size()
torch.Size([5])
>>> x.fill_(1.125)
tensor([ 1.1250,  1.1250,  1.1250,  1.1250,  1.1250])
>>> x.mean()
tensor(1.1250)
>>> x.std()
tensor(0.)
>>> x.sum()
tensor(5.6250)
>>> x.sum().item()
5.625
```

The default tensor type is a 32 bits single precision float, and it is located in the CPU memory.

In-place operations are suffixed with an underscore, and a 0d tensor can be converted back to a Python scalar with `item()`.

As in `numpy`, the `:` symbol allows to take slices of tensors

```
>>> x = torch.empty(3, 2).random_(10)
>>> x
tensor([[ 3.,  7.],
        [ 6.,  8.],
        [ 3.,  3.]])
>>> x[0]
tensor([ 3.,  7.])
>>> x[0,:]
tensor([ 3.,  7.])
>>> x[:,0]
tensor([ 3.,  6.,  3.])
```

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

```
>>> y = torch.empty(3).normal_()
>>> y
tensor([ 0.6753, -0.6621,  0.1947])
>>> m = torch.empty(3, 3).normal_()
>>> q, _ = torch.gels(y, m)
>>> torch.mm(m, q)
tensor([[ 0.6753],
        [-0.6621],
        [ 0.1947]])
```


Example: linear regression

Given a list of points

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

can we find the "best line"

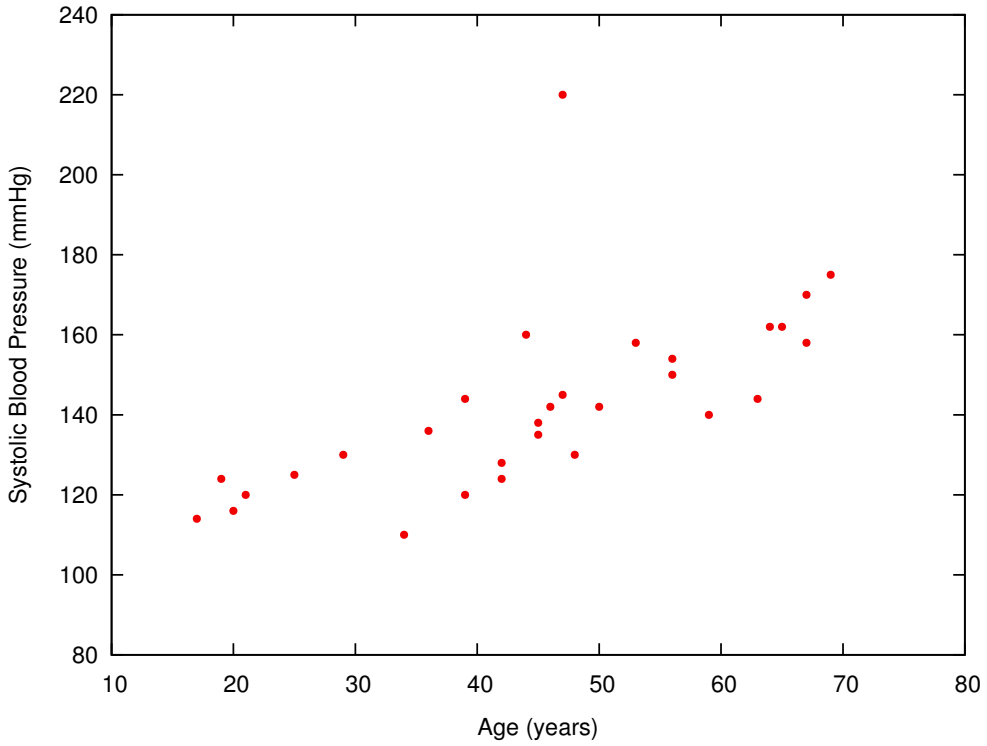
$$f(x; a, b) = ax + b$$

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

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

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

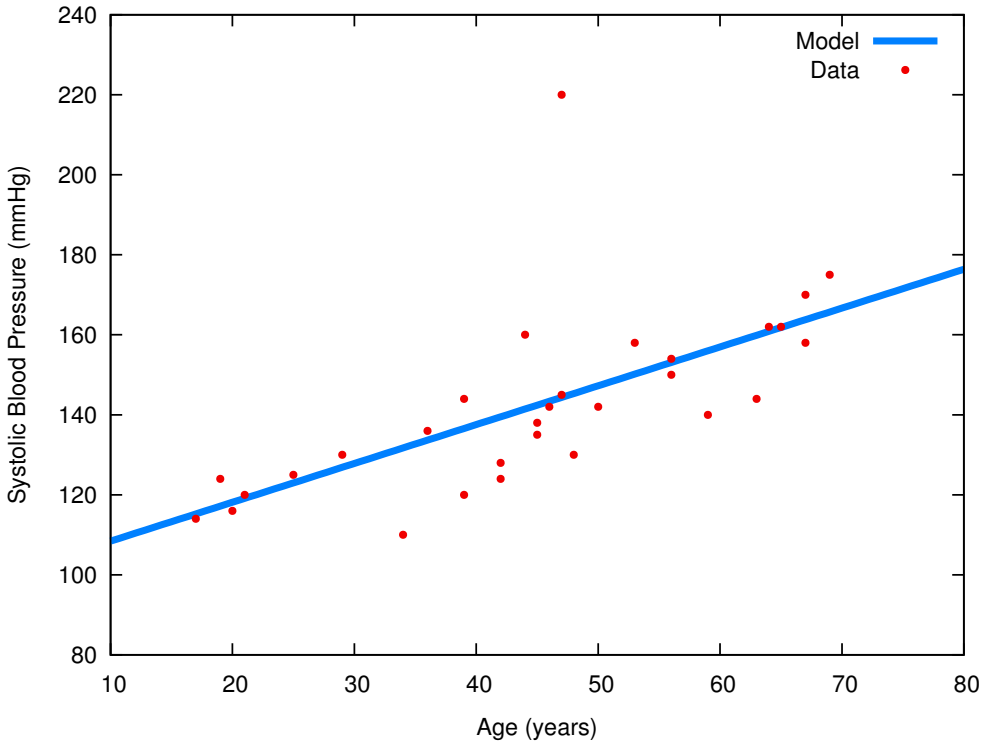

```
bash> cat systolic-blood-pressure-vs-age.dat
39  144
47  220
45  138
47  145
65  162
46  142
67  170
42  124
67  158
56  154
64  162
56  150
59  140
34  110
42  128
48  130
45  135
17  114
20  116
19  124
36  136
50  142
39  120
21  120
44  160
53  158
63  144
29  130
25  125
69  175
```



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

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

```
import torch, numpy

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

nb = data.size(0)

x, y = torch.empty(nb, 2), torch.empty(nb, 1)

x[:,0] = data[:,0]
x[:,1] = 1

y[:,0] = data[:,1]

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

a, b = alpha[0, 0].item(), alpha[1, 0].item()
```



Manipulating high-dimension signals

The tensor coefficients 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.

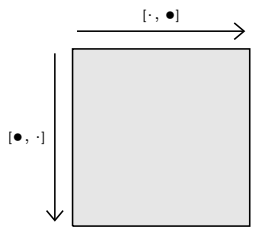
Operations on tensors located in a GPU memory are done by the GPU. We will come back to that later.


```
>>> import torch
>>> x = torch.zeros(2, 3, dtype = torch.int64)
>>> x.dtype
torch.int64
>>> x.device
device(type='cpu')
>>> x
tensor([[ 0,  0,  0],
        [ 0,  0,  0]])
>>> x = x.float()
>>> x.dtype
torch.float32
>>> x.device
device(type='cpu')
>>> x
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])
>>> x = x.cuda()
>>> x.dtype
torch.float32
>>> x
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]], device='cuda:0')
>>> q = torch.tensor([2, 3], dtype = torch.int8, device = torch.device('cuda:1'))
>>> q
tensor([ 2,  3], dtype=torch.int8, device='cuda:1')
```

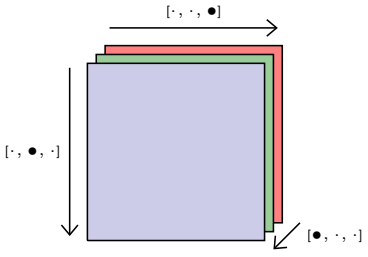
The default tensor type can be set with `torch.set_default_dtype`.

```
>>> import torch
>>> x = torch.empty(2, 3)
>>> x.dtype
torch.float32
>>> torch.set_default_dtype(torch.float64)
>>> x = torch.empty(2, 3)
>>> x.dtype
torch.float64
```

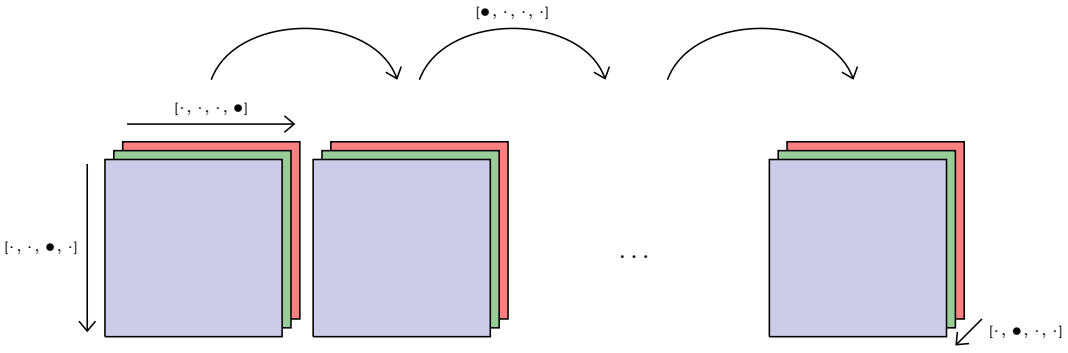

2d tensor (*e.g.* grayscale image)



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.tensor(sequence)`
- `torch.from_numpy(array)`

Indexing, Slicing, Joining, Mutating

- `torch.Tensor.view(*args)`
- `torch.Tensor.expand(*sizes)`
- `torch.Tensor.narrow(dimension, start, length)`
- `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)`

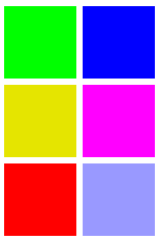
Math reduction

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

BLAS and LAPACK operations

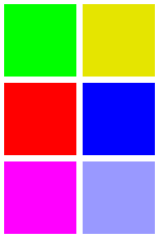
- `torch.eig(a, eigenvectors=False, out=None)`
- `torch.gels(B, A, out=None)`
- `torch.inverse(input, out=None)`
- `torch.mm(mat1, mat2, out=None)`
- `torch.mv(mat, vec, out=None)`

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



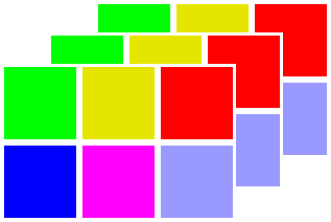
x.t()

x.view(-1)

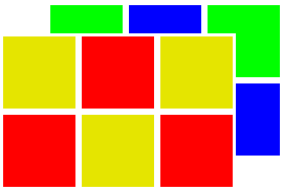


x.view(3, -1)

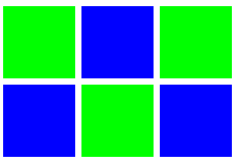
x.narrow(1, 1, 2)



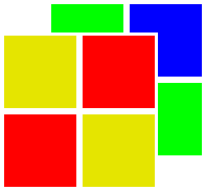
x.view(1, 2, 3).expand(3, 2, 3)



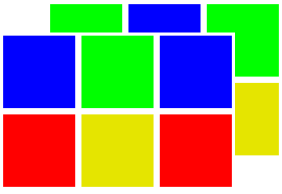
```
x = torch.tensor([ [ [ 1, 2, 1 ],
                     [ 2, 1, 2 ] ],
                   [ [ 3, 0, 3 ],
                     [ 0, 3, 0 ] ] ])
```



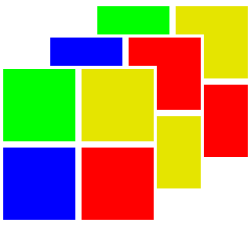
x.narrow(0, 0, 1)



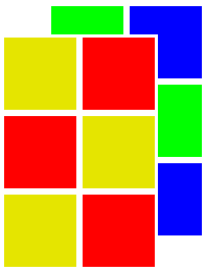
x.narrow(2, 0, 2)



x.transpose(0, 1)



x.transpose(0, 2)



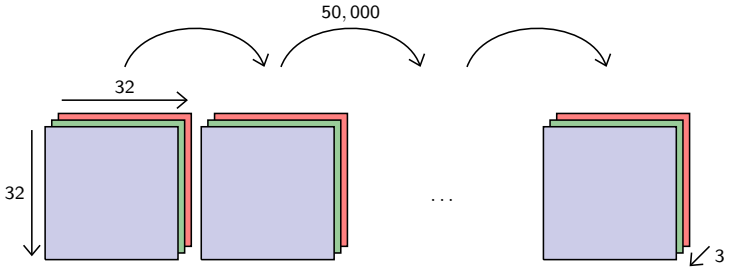
x.transpose(1, 2)

PyTorch offers simple interfaces to standard image data-bases.

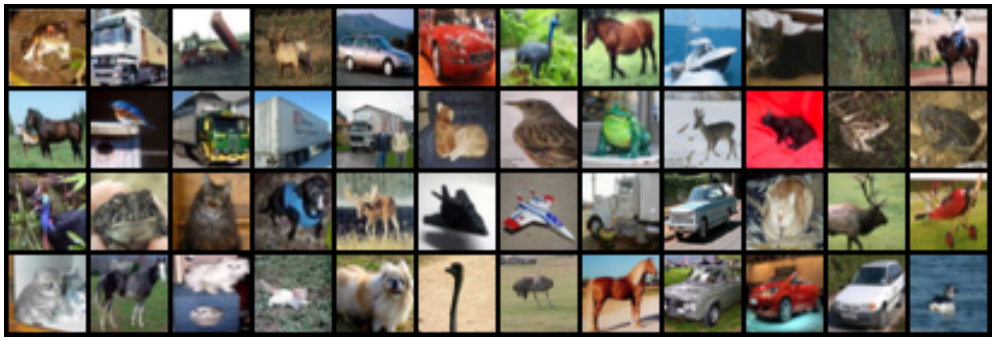
```
import torch, torchvision

# 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(x.dtype, x.size(), x.min().item(), x.max().item())
```

prints

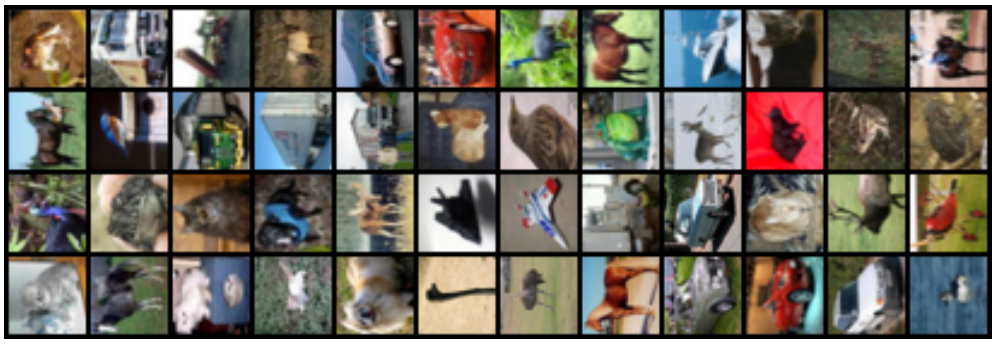
```
Files already downloaded and verified
torch.uint8 torch.Size([50000, 3, 32, 32]) 0 255
```




```
# 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)
```



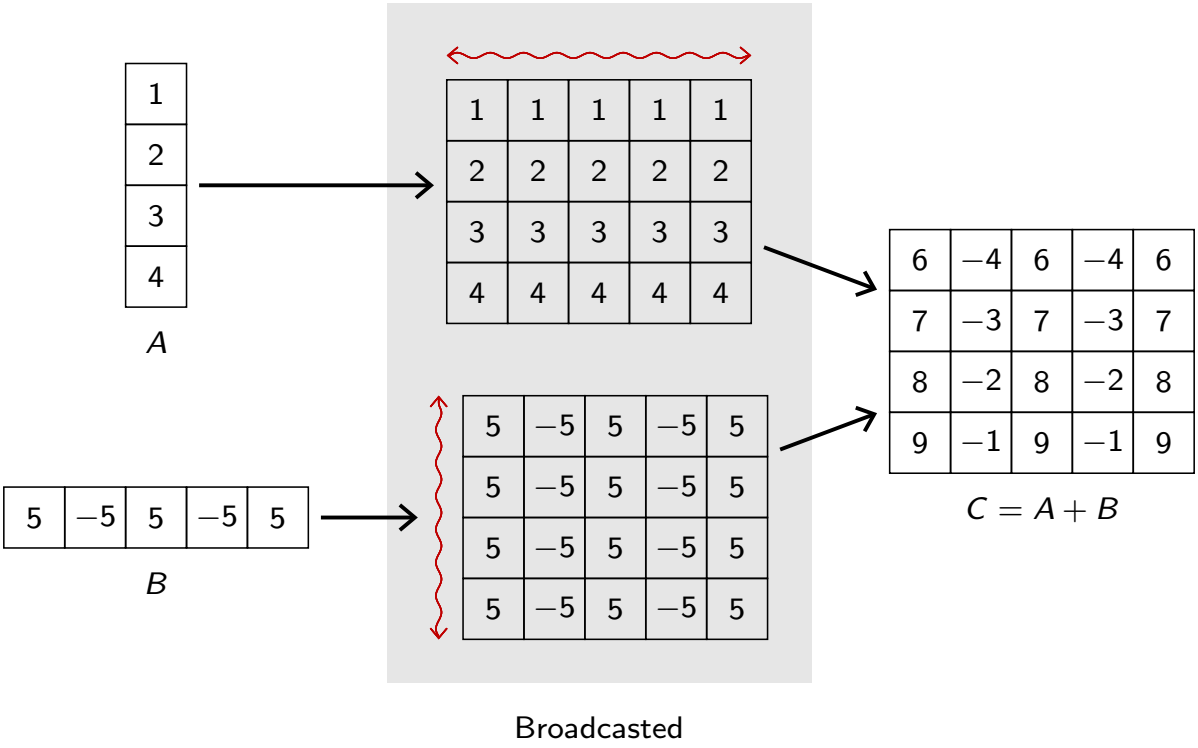

```
# Kill the green (1) and blue (2) channels
x.narrow(1, 1, 2).fill_(-1)

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.

```
>>> A = torch.tensor([[1.], [2.], [3.], [4.]])
>>> A
tensor([[ 1.],
        [ 2.],
        [ 3.],
        [ 4.]])
>>> B = torch.tensor([[5., -5., 5., -5., 5.]])
>>> B
tensor([[ 5., -5.,  5., -5.,  5.]])
>>> C = A + B
>>> C
tensor([[ 6., -4.,  6., -4.,  6.],
        [ 7., -3.,  7., -3.,  7.],
        [ 8., -2.,  8., -2.,  8.],
        [ 9., -1.,  9., -1.,  9.]])
```

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



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 = torch.tensor([1., 2., 3., 4., 5.])
>>> y = torch.empty(3, 5).fill_(2.0)
>>> z = x + y
>>> z
tensor([[ 3.,  4.,  5.,  6.,  7.],
        [ 3.,  4.,  5.,  6.,  7.],
        [ 3.,  4.,  5.,  6.,  7.]])


>>> a = torch.empty(3, 1, 5).fill_(1.0)
>>> b = torch.empty(1, 3, 5).fill_(2.0)
>>> c = a * b + a
>>> c
tensor([[[ 3.,  3.,  3.,  3.,  3.],
         [ 3.,  3.,  3.,  3.,  3.],
         [ 3.,  3.,  3.,  3.,  3.]],

        [[ 3.,  3.,  3.,  3.,  3.],
         [ 3.,  3.,  3.,  3.,  3.],
         [ 3.,  3.,  3.,  3.,  3.]],

        [[ 3.,  3.,  3.,  3.,  3.],
         [ 3.,  3.,  3.,  3.,  3.],
         [ 3.,  3.,  3.,  3.,  3.]]])
```


Tensor internals

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

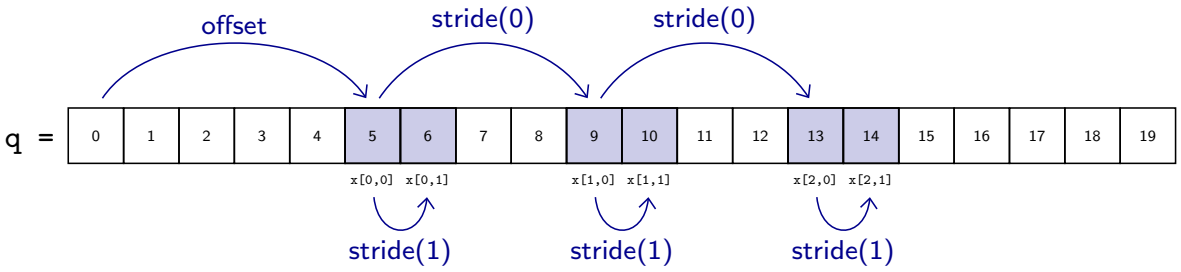
```
>>> q = torch.zeros(2, 4)
>>> q.storage()
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
[torch.FloatStorage of size 8]
>>> s = q.storage()
>>> s[4] = 1.0
>>> s
 0.0
 0.0
 0.0
 0.0
 1.0
 0.0
 0.0
 0.0
[torch.FloatStorage of size 8]
>>> q
tensor([[ 0.,  0.,  0.,  0.],
        [ 1.,  0.,  0.,  0.]])
```


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

```
>>> r = q.view(2, 2, 2)
>>> r
tensor([[[ 0.,  0.],
         [ 0.,  0.]],

        [[ 1.,  0.],
         [ 0.,  0.]]])
>>> r[1, 1, 0] = 7.0
>>> q
tensor([[ 0.,  0.,  0.,  0.],
        [ 1.,  0.,  7.,  0.]])
>>> r.narrow(0, 1, 1).fill_(3.0)
tensor([[[ 3.,  3.],
         [ 3.,  3.]]])
>>> q
tensor([[ 0.,  0.,  0.,  0.],
        [ 3.,  3.,  3.,  3.]])
```

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(0.).set_(q, storage_offset = 5, size = (3, 2), stride = (4, 1))
>>> x
tensor([[  5.,   6.],
        [  9.,  10.],
        [ 13.,  14.]])
```



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

```
>>> n = torch.linspace(1, 4, 4)
>>> n
tensor([ 1.,  2.,  3.,  4.])
>>> torch.tensor(0.).set_(n.storage(), 1, (3, 3), (0, 1))
tensor([[ 2.,  3.,  4.],
        [ 2.,  3.,  4.],
        [ 2.,  3.,  4.]])
>>> torch.tensor(0.).set_(n.storage(), 1, (2, 4), (1, 0))
tensor([[ 2.,  2.,  2.,  2.],
        [ 3.,  3.,  3.,  3.]])
```

This is in particular how transpositions and broadcasting are implemented.

This organization explains the following (maybe surprising) error

```
>>> x = torch.empty(100, 100)
>>> y = x.t()
>>> y.view(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: invalid argument 2: view size is not compatible with input tensor's size
      and stride (at least one dimension spans across two contiguous subspaces). Call .
      contiguous() before .view(). at /opt/conda/conda-bld/pytorch_1524584710464/work/
      aten/src/TH/generic/THTensor.cpp:280
>>> y.stride()
(1, 100)
```

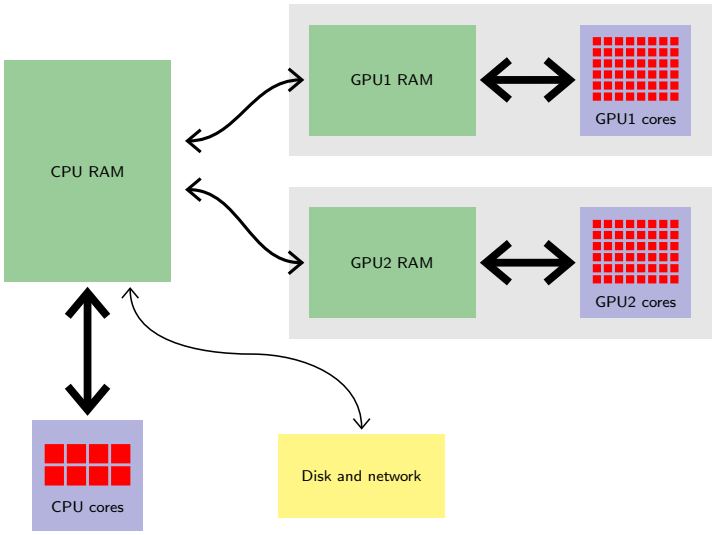
`t()` creates a tensor that shares the storage with the original tensor. It cannot be "flattened" into a 1d contiguous view without a memory copy.

Using GPUs

The size of current state-of-the-art networks makes computation a critical issue, in particular for training and optimizing meta-parameters.

Although they were historically developed for mass-market real-time CGI, their massively parallel architecture is extremely fitting to signal processing and high dimension linear algebra.

Their use is instrumental in the success of deep-learning.



A standard NVIDIA GTX 1080 has $2,560$ single-precision computing cores clocked at 1.6GHz, and deliver a peak performance of $\simeq 9$ TFlops.

The precise structure of a GPU memory and how its cores communicate with it is a complicated topic that we will not cover here.

TABLE 7. COMPARATIVE EXPERIMENT RESULTS (TIME PER MINI-BATCH IN SECOND)

		Desktop CPU (Threads used)				Server CPU (Threads used)						Single GPU		
		1	2	4	8	1	2	4	8	16	32	G980	G1080	K80
FCN-S	Caffe	1.324	0.790	**0.578**	15.444	1.355	0.997	0.745	**0.573**	0.608	1.130	0.041	**0.030**	0.071
	CNTK	1.227	0.660	**0.435**	-	1.340	0.909	0.634	0.488	**0.441**	1.000	0.045	**0.033**	0.074
	TF	7.062	4.789	2.648	**1.938**	9.571	6.569	3.399	1.710	0.946	**0.630**	0.060	**0.048**	0.109
	MXNet	4.621	2.607	2.162	**1.831**	5.824	3.356	2.395	2.040	**1.945**	2.670	-	**0.106**	0.216
	Torch	1.329	0.710	**0.423**	-	1.279	1.131	0.595	0.433	**0.382**	1.034	0.040	**0.031**	0.070
AlexNet-S	Caffe	1.606	0.999	**0.719**	-	1.533	1.045	**0.797**	0.850	0.903	1.124	0.034	**0.021**	0.073
	CNTK	3.761	1.974	**1.276**	-	3.852	2.600	1.567	1.347	**1.168**	1.579	0.045	**0.032**	0.091
	TF	6.525	2.936	1.749	**1.535**	5.741	4.216	2.202	1.160	**0.701**	0.962	0.059	**0.042**	0.130
	MXNet	2.977	2.340	2.250	**2.163**	3.518	3.203	2.926	2.828	**2.827**	2.887	0.020	**0.014**	0.042
	Torch	4.645	2.429	**1.424**	-	4.336	2.468	1.543	1.248	**1.090**	1.214	0.033	**0.023**	0.070
RenNet-50	Caffe	11.554	7.671	**5.652**	-	10.643	8.600	6.723	**6.019**	6.654	8.220	-	**0.254**	0.766
	CNTK	-	-	-	-	-	-	-	-	-	-	0.240	**0.168**	0.638
	TF	23.905	16.435	10.206	**7.816**	29.960	21.846	11.512	6.294	**4.130**	4.351	0.327	**0.227**	0.702
	MXNet	48.000	46.154	44.444	**43.243**	57.831	57.143	54.545	54.545	**53.333**	55.172	0.207	**0.136**	0.449
	Torch	13.178	7.500	**4.736**	4.948	12.807	8.391	5.471	4.164	**3.683**	4.422	0.208	**0.144**	0.523
FCN-R	Caffe	2.476	1.499	**1.149**	-	2.282	1.748	1.403	1.211	1.127	**1.127**	0.025	**0.017**	0.055
	CNTK	1.845	0.970	0.661	**0.571**	1.592	0.857	0.501	0.323	**0.252**	0.280	0.025	**0.017**	0.053
	TF	2.647	1.913	1.157	**0.919**	3.410	2.541	1.297	0.661	0.361	**0.325**	0.033	**0.020**	0.063
	MXNet	1.914	1.072	0.719	**0.702**	1.609	1.065	0.731	0.534	0.451	**0.447**	0.029	**0.019**	0.060
	Torch	1.670	0.926	**0.565**	0.611	1.379	0.915	0.662	0.440	0.402	**0.366**	0.025	**0.016**	0.051
AlexNet-R	Caffe	3.558	2.587	**2.157**	2.963	4.270	3.514	3.381	**3.364**	4.139	4.930	0.041	**0.027**	0.137
	CNTK	9.956	7.263	**5.519**	6.015	9.381	6.078	4.984	**4.765**	6.256	6.199	0.045	**0.031**	0.108
	TF	4.535	3.225	1.911	**1.565**	6.124	4.229	2.200	1.396	1.036	**0.971**	**0.227**	0.317	0.385
	MXNet	13.401	12.305	12.278	**11.950**	17.994	17.128	16.764	**16.471**	17.471	17.770	0.060	**0.032**	0.122
	Torch	5.352	3.866	**3.162**	3.259	6.554	5.288	4.365	**3.940**	4.157	4.165	0.069	**0.043**	0.141
RenNet-56	Caffe	6.741	5.451	**4.989**	6.691	7.513	**6.119**	6.232	6.689	7.313	9.302	-	**0.116**	0.378
	CNTK	-	-	-	-	-	-	-	-	-	-	0.206	**0.138**	0.562
	TF	-	-	-	-	-	-	-	-	-	-	0.225	**0.152**	0.523
	MXNet	34.409	31.255	**30.069**	31.388	44.878	43.775	**42.299**	42.965	43.854	44.367	0.105	**0.074**	0.270
	Torch	5.758	3.222	**2.368**	2.475	8.691	4.965	3.040	**2.560**	2.575	2.811	0.150	**0.101**	0.301
LSTM	Caffe	-	-	-	-	-	-	-	-	-	-	-	-	-
	CNTK	0.186	0.120	**0.090**	0.118	0.211	0.139	0.117	0.114	**0.114**	0.198	0.018	**0.017**	0.043
	TF	4.662	3.385	1.935	**1.532**	6.449	4.351	2.238	1.183	0.702	**0.598**	0.133	**0.065**	0.140
	MXNet	-	-	-	-	-	-	-	-	-	-	0.089	**0.079**	0.149
	Torch	6.921	3.831	**2.682**	3.127	7.471	4.641	3.580	**3.260**	5.148	5.851	0.399	**0.324**	0.560

Note: The mini-batch sizes for FCN-S, AlexNet-S, ResNet-50, FCN-R, AlexNet-R, ResNet-56 and LSTM are 64, 16, 16, 1024, 1024, 128 and 128 respectively.

(Shi et al., 2016)

The current standard to program a GPU is through the CUDA ("Compute Unified Device Architecture") model, defined by NVIDIA.

Alternatives are OpenCL, backed by many CPU/GPU manufacturers, and more recently AMD's HIP ("Heterogeneous-compute Interface for Portability").

Google developed its own processor for deep learning dubbed TPU ("Tensor Processing Unit") for in-house use. It is targeted at TensorFlow and offers excellent flops/watt performance.

In practice, as of today (27.01.2018), NVIDIA hardware remains the default choice for deep learning, and CUDA is the reference framework in use.

From a practical perspective, libraries interface the framework (*e.g.* PyTorch) with the "computational backend" (*e.g.* CPU or GPU)

- BLAS ("Basic Linear Algebra Subprograms"): vector/matrix products, and the cuBLAS implementation for NVIDIA GPUs,
- LAPACK ("Linear Algebra Package"): linear system solving, Eigen-decomposition, etc.
- cuDNN ("NVIDIA CUDA Deep Neural Network library") computations specific to deep-learning on NVIDIA GPUs.

The use of the GPUs in PyTorch is done by moving tensors in their memory.

Apart from `copy_()`, operations cannot mix different tensor types or devices, and an operation done on tensors in a given device's memory is executed by the said device:

```
>>> import torch
>>> x = torch.empty(3, 5).normal_()
>>> y = torch.zeros(3, 5).normal_().cuda()
>>> x.copy_(y)
tensor([[ 0.4071,  0.7589, -0.5321,  0.9103, -1.4985],
        [-0.1059,  2.1554, -0.0774, -0.4520,  1.5123],
        [ 0.1322,  0.1002, -0.4071,  1.8927, -0.5800]])

>>> x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: Expected object of type torch.FloatTensor but found type torch.cuda.
      FloatTensor for argument #3 'other'
```

Operations maintain the type of the tensors, so you generally do not need to worry about making your code generic regarding the tensor types.

However, if you have to explicitly create a new tensor, the best is to use `new_*()` methods.

```
>>> u = torch.empty(3, 5).normal_()
>>> u.new_zeros(1, 2)
tensor([[ 0.,  0.]])
>>> v = torch.empty(3, 5, dtype = torch.float16, device = torch.device('cuda:1')).fill_
      (1.0)
>>> v.new_full((2, 3), 1.4)
tensor([[ 1.4004,  1.4004,  1.4004],
        [ 1.4004,  1.4004,  1.4004]], dtype=torch.float16, device='cuda:1')
```


The method `torch.cuda.is_available()` returns a Boolean value indicating if a GPU is available.

The `Tensor`s' method `cuda()` returns a clone on the GPU **if the tensor is not already there** or returns the tensor itself if it was already there, keeping the bit precision. Conversely the method `cpu()` makes a clone on the CPU if needed.

They both keep the original tensor unchanged.

Moving data between the CPU and the GPU memories is far slower than moving it inside the GPU memory.

If multiple GPUs are available, cross-GPUs operations are not allowed by default, with the exception of `copy_()`.

An operation between tensors in the same GPU produces a results in the same GPU also.

Each GPU has a numerical id, and `torch.cuda.device(id)` allows to specify where GPU tensors should be created by `cuda()`. An explicit GPU id can also be provided to the latter.

`torch.cuda.device_of(obj)` selects the device to that of the specified tensor or storage.

References

S. Shi, Q. Wang, P. Xu, and X. Chu. Benchmarking state-of-the-art deep learning software tools. *CoRR*, abs/1608.07249, 2016.