Introduction to PyTorch

Tim Rocktäschel & Sebastian Riedel
COMP0087 Natural Language Processing

Outline

- Linear algebra refresher and introduction to notation
  - Scalars
  - Vectors
  - Matrices
  - Tensors
  - …
- Introduction to PyTorch tensor operations
- Introduction to PyTorch computation graphs, gradient calculation and basic machine learning and NLP infrastructure

What is PyTorch?

- Replacement for NumPy to use the power of GPUs
- Deep learning research platform that provides flexibility and speed
- Installation: https://pytorch.org/get-started/locally/
- Many alternatives based on similar principles

W_h h W_x x

TensorFlow

theano

Chainer

Keras

dynet

Why PyTorch?

François Chollet @fchollet
Following

PyTorch has a lot of marketing firepower behind it, and as a result there's a common misconception that it has "momentum". Does it? I can't tell for sure, but the handful of traction indicators I monitor are showing that its user base has likely peaked around April-May 2018

> **Vincenzo Manzoni** @vincenzomanzoni
> #TensorFlow and #Keras are the most diffused framework for deep learning (according to this study). #PyTorch is getting momentum. All the other are niche players. My personal suggestion is to start with Keras backed by Tensorflow for its simplicity and the number of resources. …

8:33 PM - 3 Oct 2018

Introducing Pytorch for fast.ai

Written: 08 Sep 2017 by *Jeremy Howard*

The next fast.ai courses will be based nearly entirely on a new framework we have developed, built on Pytorch. Pytorch is a different kind of deep learning library (dynamic, rather than static), which has been adopted by many (if not most) of the researchers that we most respect, and in a recent Kaggle competition was used by nearly all of the top 10 finishers.

Stanford NLP Group @stanfordnlp
Following

We're gearing up for the 2019 edition of Stanford CS224N: Natural Language Processing with Deep Learning. Starts Jan 8—over 500 students enrolled—using PyTorch—new Neural MT assignments—new lectures on transformers, subword models, and human language.

Andrej Karpathy @karpathy
Following

I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eye sight has improved.

7:56 PM - 26 May 2017

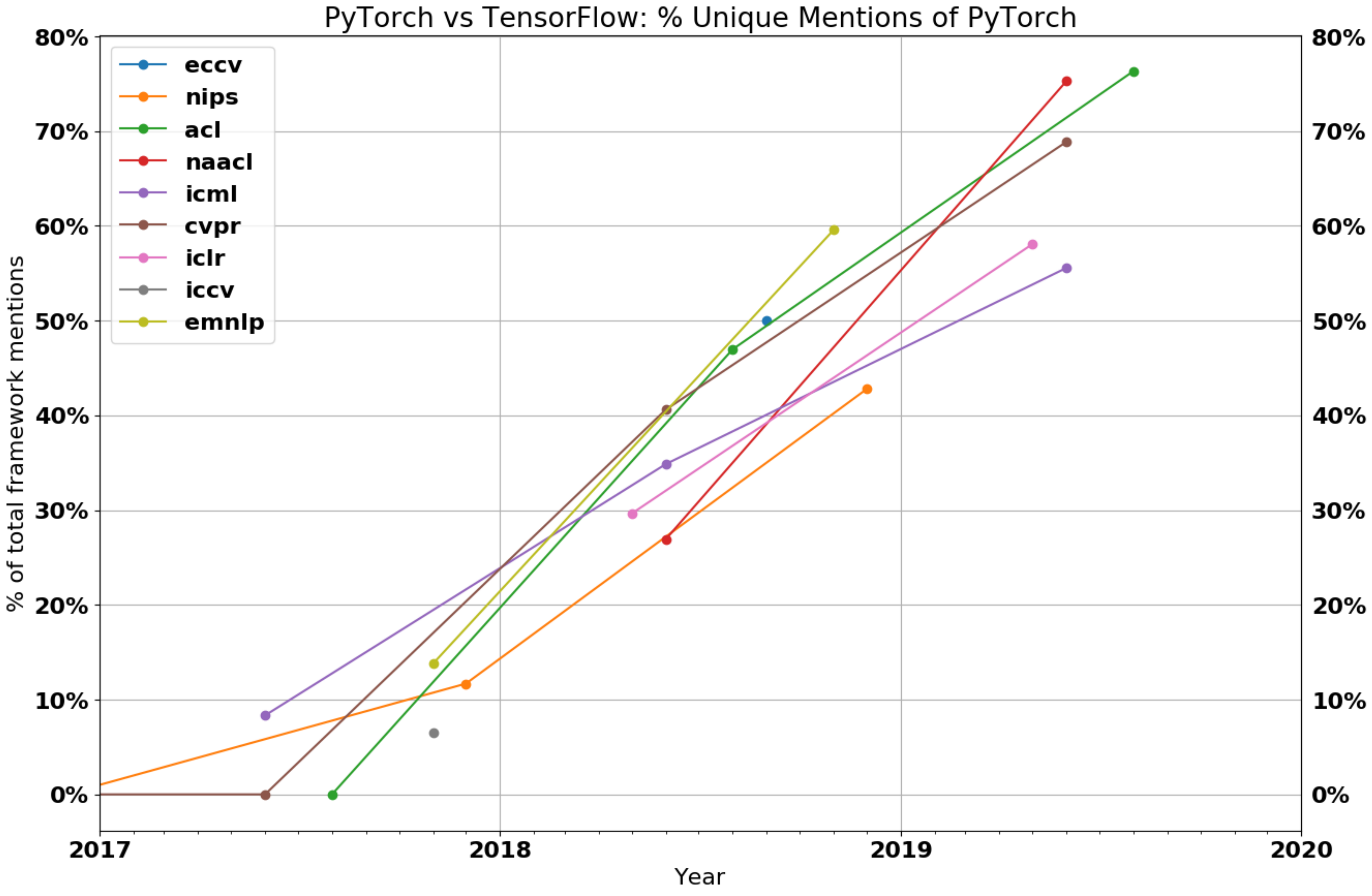
Soumith Chintala @soumithchintala
Following

TF goes imperative with eager, pytorch getting static optimizations and production-ready with JIT and onnx. Worlds are slowly converging...

8:03 PM - 31 Oct 2017

Why PyTorch?

- Dynamic computation graphs (by now also supported in TF Eager) are from our experience more intuitive for newcomers
- Outside of Assignment 1, choose what you feel most comfortable and productive with



https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/

Scalars

lower-case letters: $x \in \mathbb{R}$

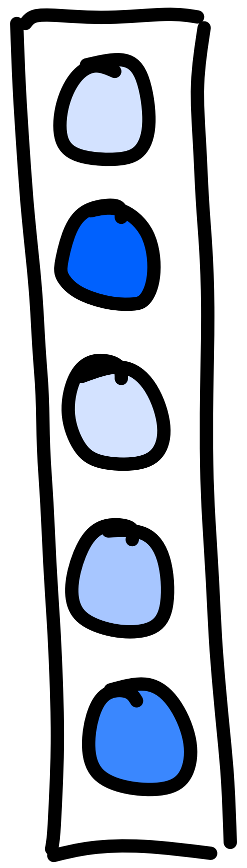
```
# uninitialized
torch.empty([])
> tensor(0.)

# randomly initialized
torch.rand([])
> tensor(0.4224)

torch.zeros([], dtype=torch.long)
> tensor(0)

torch.zeros([], dtype=torch.float)
> tensor(0.)

torch.tensor(1.337)
> tensor(1.3370)

# map to Python built-in type
torch.tensor(1.5).item()
> 1.5
```

```
x = torch.tensor(1.3)
y = torch.tensor(2.7)
x+y
> tensor(4.)

x-y
> tensor(-1.4000)

x/y
> tensor(0.4815)

x*y
> tensor(3.5100)

# exponentiation
x**y
> tensor(2.0307)
```

Vectors

lower-case boldface letters: $\mathbf{x} \in \mathbb{R}^n$

$$\vec{x} = \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$



```
x = torch.tensor([1, 2, 3])
y = torch.tensor([4, 5, 6])
x+y
> tensor([5, 7, 9])
```

$2\mathbf{x}$

```
x*2
> tensor([2, 4, 6])
```

$\mathbf{x}^2$

```
x**2
> tensor([1, 4, 9])
```

$\mathbf{x} \odot \mathbf{y}$

```
x*y
> tensor([ 4, 10, 18])

x**y
> tensor([  1,  32, 729])
```

$\mathbf{x}^\top \mathbf{y}$

```
x.dot(y)
> tensor(32)
```

$\mathbf{x}\mathbf{y}^\top$

```
# vector outer product
torch.ger(x,y)
> tensor([[ 4,  5,  6],
          [ 8, 10, 12],
          [12, 15, 18]])
```

Matrices

upper-case boldface letters: $\mathbf{X} \in \mathbb{R}^{m \times n}$

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & \ddots & & x_{2n} \\ \vdots & & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}$$

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
y = torch.tensor([[7, 8, 9], [1, 2, 3]])
x+y
> tensor([[ 8, 10, 12],
          [ 5,  7,  9]])
```

```
x.dim()
> 2

x.shape
> torch.Size([2, 3])

x.numel()
> 6
```

$\mathbf{X} \odot \mathbf{Y}$

```
x*y
> tensor([[ 7, 16, 27],
          [ 4, 10, 18]])
```

$\mathbf{X}^\top$

```
x.t()
> tensor([[1, 4],
          [2, 5],
          [3, 6]])
```

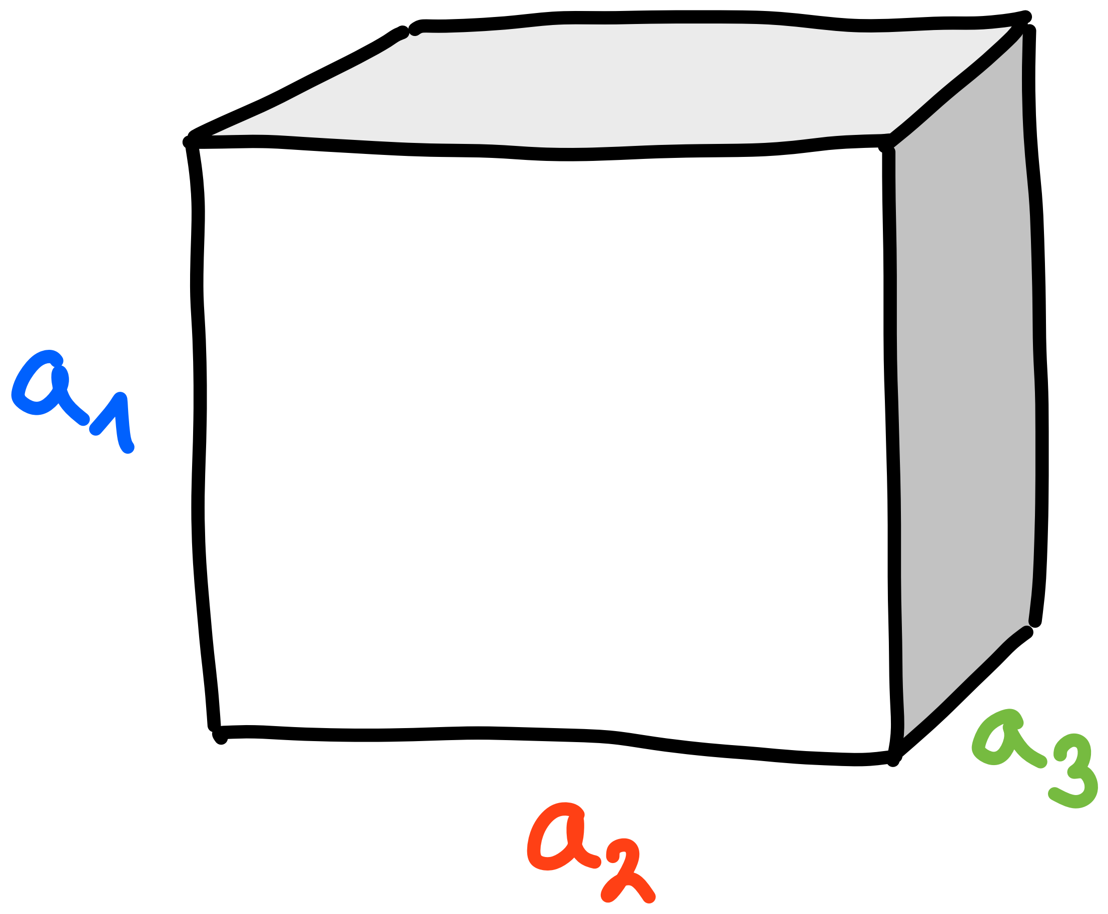
$\mathbf{X}\mathbf{Y}^\top$

```
x.mm(y.t())
> tensor([[ 50,  14],
          [122,  32]])
```

Tensors

upper-case Euler script letters: $\mathscr{X} \in \mathbb{R}^{a_1 \times \cdots \times a_n}$

- Generalization of scalars, vectors and matrices to arbitrary number of dimensions
- Equivalent to NumPy's multidimensional array (numpy.ndarray)




```
torch.rand(3, 2, 4)
> tensor([[[0.5621, 0.3252, 0.9327, 0.3415],
          [0.6985, 0.5490, 0.0502, 0.8289]],

         [[0.4670, 0.8705, 0.4228, 0.6695],
          [0.0448, 0.7635, 0.5609, 0.2032]],

         [[0.9706, 0.5218, 0.9011, 0.7341],
          [0.9576, 0.1091, 0.0297, 0.3370]]])
```

a_1 a_2 a_2 a_2

a_3

- You can think of higher-order tensors as stacked matrices
- Most of the operations introduced so far are generalized to tensors — they work for scalars, vectors matrices, and higher-order tensors

Tensor Initialization

```
# uninitialized
torch.empty(2, 3)
> tensor([[4.47e+21, 4.43e+27, 1.38e-14],
          [3.97e-14, 1.40e-44, 0.00e+00]])

torch.zeros(2, 3)
> tensor([[0., 0., 0.],
          [0., 0., 0.]])

torch.ones(2, 3)
> tensor([[1., 1., 1.],
          [1., 1., 1.]])

# identity tensor
torch.eye(3, 3)
> tensor([[1., 0., 0.],
          [0., 1., 0.],
          [0., 0., 1.]])

torch.full((2, 3), 7)
> tensor([[7., 7., 7.],
          [7., 7., 7.]])
```

```
# randomly initialized from [0,1]
torch.rand(2, 3)
> tensor([[0.6973, 0.5121, 0.7239],
          [0.7017, 0.6737, 0.5170]])

# randomly initialized from N(0,1)
torch.randn(2, 3)
> tensor([[-0.5630, -1.5636, -0.6183],
          [ 0.1433, -1.1500, -0.6952]])

torch.randint(low=0, high=10, size=(2, 3))
> tensor([[8, 1, 8],
          [6, 2, 5]])

# random permutation of ints from 0 to n - 1
torch.randperm(n=5)
> tensor([3, 0, 1, 4, 2])
```

Data Types

Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	`torch.float32` or `torch.float`	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit floating point	`torch.float64` or `torch.double`	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
16-bit floating point	`torch.float16` or `torch.half`	`torch.HalfTensor`	`torch.cuda.HalfTensor`
8-bit integer (unsigned)	`torch.uint8`	`torch.ByteTensor`	`torch.cuda.ByteTensor`
8-bit integer (signed)	`torch.int8`	`torch.CharTensor`	`torch.cuda.CharTensor`
16-bit integer (signed)	`torch.int16` or `torch.short`	`torch.ShortTensor`	`torch.cuda.ShortTensor`
32-bit integer (signed)	`torch.int32` or `torch.int`	`torch.IntTensor`	`torch.cuda.IntTensor`
64-bit integer (signed)	`torch.int64` or `torch.long`	`torch.LongTensor`	`torch.cuda.LongTensor`

NumPy Bridge

- NumPy to PyTorch

```
import numpy as np
n = np.ones(5)
torch.from_numpy(n)
> tensor([1., 1., 1., 1., 1.],
    dtype=torch.float64)

n = np.ones(5)
x = torch.from_numpy(n)
# still pointing to the same memory!
np.add(n, 1, out=n)
x
> tensor([2., 2., 2., 2., 2.],
    dtype=torch.float64)
```

- PyTorch to NumPy

```
x = torch.ones(5)
x.numpy()
> array([1., 1., 1., 1., 1.],
    dtype=float32)

x = torch.ones(5)
n = x.numpy()
# still pointing to the same memory!
x.add_(1)
n
> array([2., 2., 2., 2., 2.],
    dtype=float32)
```

Basic Operations

- As with scalars, vectors and matrices, we can perform element-wise operations:

  `+  -   *   **   /`

```
x = torch.rand(2, 3, 2)
y = torch.ones(2, 3, 2)
x + y
> tensor([[[1.4733, 1.0785],
           [1.4746, 1.8736],
           [1.6190, 1.2342]],

          [[1.4081, 1.1775],
           [1.9615, 1.0118],
           [1.4661, 1.7458]]])
```

- Many operations have aliases and in-place equivalents

```
z = torch.add(x, y)
# in-place; no extra memory allocation
x.add_(y)
# equal up to predefined tolerance
x.allclose(z)
> True
```

Tensor Contraction

- Generalization of vector-vector, matrix-vector, matrix-matrix product etc.
- @ sums over variable n and n-1 unless Y has dimension 1

$$\mathcal{X} \in \mathbb{R}^{a_1 \times \cdots \times a_n \times b_1 \times b_2}$$

$$\mathcal{Y} \in \mathbb{R}^{a_n \times \cdots \times a_n \times b_2 \times b_3}$$

$$\mathcal{Z} = \mathcal{X} @ \mathcal{Y} \in \mathbb{R}^{a_1 \times \cdots \times a_n \times b_1 \times b_3}$$

$$\mathcal{Z}_{a_1,\cdots,a_n,b_1,b_3} = \sum_{b_2} \mathcal{X}_{a_1,\cdots,a_n,b_1,b_2} \mathcal{Y}_{a_1,\cdots,a_n,b_2,b_3}$$

```
x = torch.rand(2, 3)
y = torch.rand(3)
x @ y
> tensor([0.2703, 0.2689])

x = torch.rand(2, 4)
y = torch.rand(4, 3)
x @ y
> tensor([[1.2979, 0.9207, 0.8239],
          [1.0564, 0.8383, 0.7090]])
```

```
x = torch.rand(2, 3, 4)
y = torch.rand(4, 2)
x @ y
> tensor([[[0.8348, 0.9044],
           [0.3368, 0.4806],
           [0.7613, 1.0067]],

          [[1.0862, 1.1560],
           [0.3870, 0.5424],
           [0.9648, 0.9473]]])
```

For high-order tensors @ is a batch-matrix multiplication (see einsum for actual tensor contraction)!

Transpose

```
x = torch.arange(0,6).view(2,3)
x.t()
> tensor([[0, 3],
          [1, 4],
          [2, 5]])

x.t().allclose(x.transpose(0, 1))
> True

x.transpose(0, 1).allclose(
  x.transpose(1, 0)
)
> True
```

```
x = torch.arange(0,12).view(2,3,2)
x
> tensor([[[ 0,  1],
           [ 2,  3],
           [ 4,  5]],

          [[ 6,  7],
           [ 8,  9],
           [10, 11]]])


x.transpose(1, 2)
> tensor([[[ 0,  2,  4],
           [ 1,  3,  5]],

          [[ 6,  8, 10],
           [ 7,  9, 11]]])
```

Sum, Min, Max, and Mean

```
x = torch.arange(0, 12).view(2, 3, 2)
x
> tensor([[[ 0,  1],
           [ 2,  3],
           [ 4,  5]],

          [[ 6,  7],
           [ 8,  9],
           [10, 11]]])


x.sum()
> tensor(66)

x.sum(2)
> tensor([[ 1,  5,  9],
          [13, 17, 21]])

x.min()
> tensor(0)
```

```
values, indices = x.min(1)
values
> tensor([[0, 1],
          [6, 7]])

x.max()
> tensor(11)

values, indices = x.max(0)
values
> tensor([[ 6,  7],
          [ 8,  9],
          [10, 11]])

# mean is not defined for Long tensors
x = torch.arange(0, 12,
  dtype=torch.float32).view(2, 3, 2)
x.mean()
> tensor(5.5000)

# mean over last dimension
x.mean(-1)
> tensor([[ 0.5000,  2.5000,  4.5000],
          [ 6.5000,  8.5000, 10.5000]])
```

View & Reshape

```
# vector [0, 1, 2, ..., 11]
x = torch.arange(0, 12)
# viewed as a [2 x 3 x 2] tensor
x.view(2, 3, 2)
> tensor([[[ 0,  1],
           [ 2,  3],
           [ 4,  5]],

          [[ 6,  7],
           [ 8,  9],
           [10, 11]]])

# viewed as a [3 x 4] matrix
x.view(3, 4)
> tensor([[ 0,  1,  2,  3],
          [ 4,  5,  6,  7],
          [ 8,  9, 10, 11]])
```

```
# viewed as a [3 x 2 x 2] tensor via
# inferring one unspecified dimension!
x.view(-1, 2, 2)
> tensor([[[ 0,  1],
           [ 2,  3]],

          [[ 4,  5],
           [ 6,  7]],

          [[ 8,  9],
           [10, 11]]])
```

- view and reshape share API
  - view does not allocate new memory
  - reshape works with non-contiguous tensors, but can copy memory
  - If possible, use view

Squeeze & Unsqueeze

```
x = torch.rand(3, 1, 2, 1)
# remove all singleton dimensions
x.squeeze()
> tensor([[0.8710, 0.2562],
          [0.6167, 0.4434],
          [0.7147, 0.9117]])

x = torch.rand(3)
# [1 x 3] matrix / row-vector
x.unsqueeze(dim=0)
> tensor([[0.0820, 0.3074, 0.3773]])

# equiv.: expand using None indexing
y = x[None, :]
# equiv.: using view
z = x.view(-1, 3)
y.allclose(z)
> True
```

```
x = torch.arange(0, 12).view(2, 3, 2)
y = torch.rand(6, 2)
# equiv.: x.view(y.size())
x.view_as(y)
> tensor([[ 0,  1],
          [ 2,  3],
          [ 4,  5],
          [ 6,  7],
          [ 8,  9],
          [10, 11]])
```

Expand & Repeat

```
x = torch.arange(0, 3)
x = x.unsqueeze(1)
# returns an expanded view of x
x = x.expand(-1, 4)
x[0,1] = 7
x
> tensor([[7, 7, 7, 7],
          [1, 1, 1, 1],
          [2, 2, 2, 2]])
```

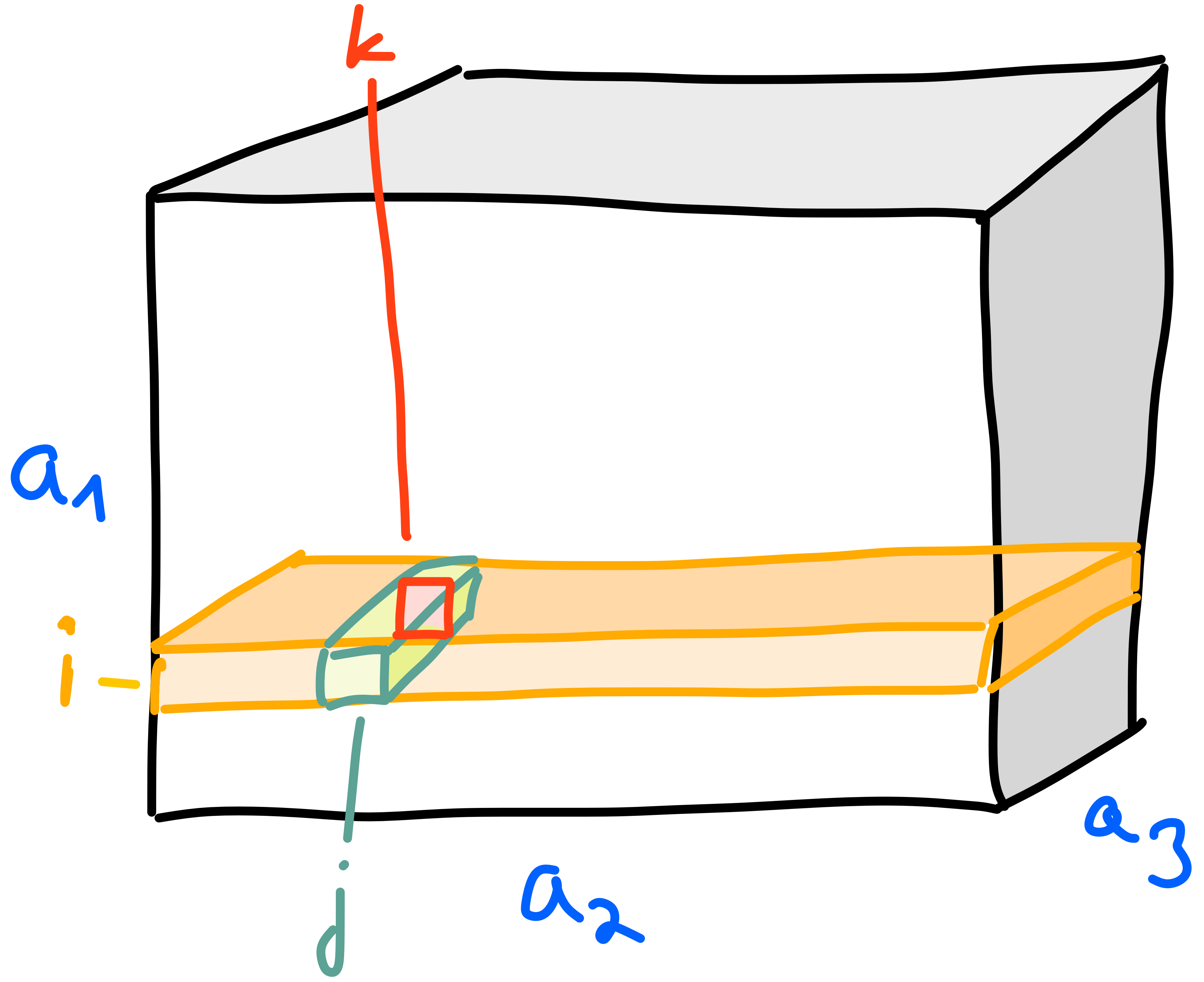
```
x = torch.arange(0, 3)
x = x.unsqueeze(1)
# allocates new memory
x = x.repeat(1, 4)
x[0,1] = 7
x
> tensor([[0, 7, 0, 0],
          [1, 1, 1, 1],
          [2, 2, 2, 2]])
```

```
x = torch.arange(0, 6).unsqueeze(1)
y = torch.rand(6, 3)
x.expand_as(y)
> tensor([[0, 0, 0],
          [1, 1, 1],
          [2, 2, 2],
          [3, 3, 3],
          [4, 4, 4],
          [5, 5, 5]])
```

- expand and reshape share API
  - reshape allocates new memory
  - expand does not
  - If possible, use expand but watch out for side effects!

Indexing

```
x = torch.arange(0, 24).view(2, 3, 4)
x
> tensor([[[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  9, 10, 11]],

          [[12, 13, 14, 15],
           [16, 17, 18, 19],
           [20, 21, 22, 23]]])

x[1]
> tensor([[12, 13, 14, 15],
          [16, 17, 18, 19],
          [20, 21, 22, 23]])

x[1,0]
> tensor([12, 13, 14, 15])

x[1,0,3]
> tensor(15)
```



Advanced Indexing

```
x
> tensor([[[ 0,  1,  2,  3],
          [ 4,  5,  6,  7],
          [ 8,  9, 10, 11]],

         [[12, 13, 14, 15],
          [16, 17, 18, 19],
          [20, 21, 22, 23]]])

x[1, 1:, 1:-1]
> tensor([[17, 18],
          [21, 22]])
```

```
indices = torch.tensor([0, 0, 2, 1])
x[:, indices]
> tensor([[[ 0,  1,  2,  3],
           [ 0,  1,  2,  3],
           [ 8,  9, 10, 11],
           [ 4,  5,  6,  7]],

          [[12, 13, 14, 15],
           [12, 13, 14, 15],
           [20, 21, 22, 23],
           [16, 17, 18, 19]]])
```

Cat, Split, Chunk, and Stack

```
x = torch.arange(0, 6).view(3, 2)
torch.cat([x, x], dim=1)
> tensor([[0, 1, 0, 1],
          [2, 3, 2, 3],
          [4, 5, 4, 5]])

x = torch.arange(0, 24).view(4, 6)
torch.split(x, 3, dim=1)[1]
> tensor([[ 3,  4,  5],
          [ 9, 10, 11],
          [15, 16, 17],
          [21, 22, 23]])

x = torch.arange(0, 24).view(4, 6)
torch.chunk(x, 3, dim=1)[1]
> tensor([[ 2,  3],
          [ 8,  9],
          [14, 15],
          [20, 21]])
```

```
x = torch.arange(0, 6).view(2, 3)
y = torch.arange(6, 12).view(2, 3)
torch.stack([x, y, x])
> tensor([[[ 0,  1,  2],
           [ 3,  4,  5]],

          [[ 6,  7,  8],
           [ 9, 10, 11]],

          [[ 0,  1,  2],
           [ 3,  4,  5]]])
```

Broadcasting

Often you can make two tensors compatible for operations via unsqueezed and expand/view functions. In such cases, **broadcasting** does this for you, automatically.

Two tensors are "broadcastable" if each tensor has at least one dimension and when iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist.

```
x = torch.arange(0, 6).view(2, 3)
y = torch.tensor([[-3], [3]])
# x and y are broadcastable
# 1st trailing dimension: y has size 1
# 2nd trailing dimension: both have size 2
x * y
> tensor([[ 0, -3, -6],
          [ 9, 12, 15]])

x = torch.empty(3, 5, 7, 9)
y = torch.empty(3, 5, 7, 9)
# same shapes are always broadcastable

x = torch.empty((0,))
y = torch.empty(2, 2)
# x and y are not broadcastable
# x does not have at least one dimension
```

```
x = torch.empty(5, 3, 4, 1)
y = torch.empty(   3, 1, 1)
# x and y are broadcastable
# 1st trailing dimension: both have size 1
# 2nd trailing dimension: y has size 1
# 3rd trailing dimension: x size == y size
# 4th trailing dimension: y dim doesn't exist

x = torch.empty(5, 2, 4, 1)
y = torch.empty(   3, 1, 1)
# x and y are not broadcastable
# in the 3rd trailing dimension 2 != 3
```

Sparse Tensors

```python
indices = torch.LongTensor([[2, 4, 7],
                            [3, 2, 1]])
values = torch.FloatTensor([3, 4, 5])
# sparse
x = torch.sparse.FloatTensor(
  indices, values, torch.Size([10, 100000000]))
# dense
m = torch.rand(100000000, 1)
torch.sparse.mm(x, m)
> tensor([[0.0000],
          [0.0000],
          [1.3598],
          [0.0000],
          [1.9068],
          [0.0000],
          [0.0000],
          [1.8615],
          [0.0000],
          [0.0000]])
```

```python
indices = torch.LongTensor([[2, 4, 7],
                            [3, 2, 1]])
values = torch.FloatTensor([3, 4, 5])
# sparse
sparse_x = torch.sparse.FloatTensor(
  indices, values, torch.Size([10, 100000000]))
# dense
dense_x = torch.zeros(10, 100000000)
dense_x[2, 3] = 3
dense_x[4, 2] = 4
dense_x[7, 1] = 5
# dense
m = torch.rand(100000000, 1)

%%timeit
torch.mm(dense_x, m)
> 1 loop, best of 3: 441 ms per loop

%%timeit
torch.sparse.mm(sparse_x, m)
> 100000 loops, best of 3: 18.7 µs per loop
```

Einstein Summation Notation

"I admire the elegance of your method of computation; it must be nice to ride through these fields upon the horse of true mathematics while the like of us have to make our way laboriously on foot."
— *Albert Einstein in a letter to Tullio Levi-Civita*

- Einsum is a domain-specific language for tensor operations implemented in NumPy, TensorFlow and PyTorch
- Example: We want to calculate the multiplication of two matrices X and Y followed by summing up all columns

$$R_i = \sum_j \sum_k X_{ik} Y_{kj} = X_{ik} Y_{kj}$$

```
result = torch.einsum('ik,kj->i', [x, y])
```

- Summation Sigmas can be dropped since, by convention, repeated indices (k in this example) and indices not mentioned in the output tensor (j in this example) are implicitly summed over

https://obilaniu6266h16.wordpress.com/2016/02/04/einstein-summation-in-numpy/

Einsum Examples I

```
# matrix transpose
x = torch.arange(6).reshape(2, 3)
torch.einsum('ij->ji', [x])
> tensor([[0, 3],
          [1, 4],
          [2, 5]])

# sum
x = torch.arange(6).reshape(2, 3)
torch.einsum('ij->', [x])
> tensor(15)

# column sum
x = torch.arange(6).reshape(2, 3)
torch.einsum('ij->j', [x])
> tensor([3, 5, 7])
```

```
# row sum
x = torch.arange(6).reshape(2, 3)
torch.einsum('ij->i', [x])
> tensor([ 3, 12])

# matrix-vector multiplication
x = torch.arange(6).reshape(2, 3)
y = torch.arange(3)
torch.einsum('ik,k->i', [x, y])
> tensor([ 5, 14])

# matrix-matrix multiplication
x = torch.arange(6).reshape(2, 3)
y = torch.arange(15).reshape(3, 5)
torch.einsum('ik,kj->ij', [x, y])
> tensor([[ 25,  28,  31,  34,  37],
          [ 70,  82,  94, 106, 118]])
```

Einsum Examples II

```
# vector dot product
x = torch.arange(3)
y = torch.arange(3,6)
torch.einsum('i,i->', [x, y])
> tensor(14)

# matrix dot product
x = torch.arange(6).reshape(2, 3)
y = torch.arange(6,12).reshape(2, 3)
torch.einsum('ij,ij->', [x, y])
> tensor(145)

# Hadamard Product
x = torch.arange(6).reshape(2, 3)
y = torch.arange(6,12).reshape(2, 3)
torch.einsum('ij,ij->ij', [x, y])
> tensor([[ 0,  7, 16],
          [27, 40, 55]])
```

```
# vector outer product
x = torch.arange(3)
y = torch.arange(3,7)
torch.einsum('i,j->ij', [x, y])
> tensor([[ 0,  0,  0,  0],
          [ 3,  4,  5,  6],
          [ 6,  8, 10, 12]])

# matrix diagonal
x = torch.arange(0, 16).view(4, 4)
torch.einsum('ii->i', [x])
> tensor([ 0,  5, 10, 15])
```

Einsum Examples III

```
# batch matrix-multiplication
x = torch.randn(3,2,5)
y = torch.randn(3,5,3)
torch.einsum('ijk,ikl->ijl', [x, y])
> tensor([[[ 0.2245, -0.5202,  0.6898],
           [ 1.7316, -0.6911, -1.7396]],

          [[-0.8591,  0.0888,  0.3778],
           [-0.3694, -1.2621,  6.3152]],

          [[-0.7040,  4.3190, -0.4269],
           [-1.3730,  0.0434, -2.7862]]])
```

$$R_{ijl} = \sum_k X_{ijk} Y_{ikl} = Y_{ijk} X_{ikl}$$

```
# tensor contraction
x = torch.randn(2,3,5,7)
y = torch.randn(11,13,3,17,5)
torch.einsum('pqrs,tuqvr->pstuv', [x, y]).shape
> torch.Size([2, 7, 11, 13, 17])
```

$$R_{pstuv} = \sum_q \sum_r X_{pqrs} Y_{tuqvr} = X_{pqrs} Y_{tuqvr}$$

```
# bilinear transformation
x = torch.randn(2,3)
y = torch.randn(4,3,7)
z = torch.randn(2,7)
torch.einsum('ik,jkl,il->ij', [x, y, z])
> tensor([[-1.0250,  1.6350, -3.7091,  2.2055],
          [ 2.8283,  1.4494, -0.5114, -6.9879]])
```

$$R_{ij} = \sum_k \sum_l X_{ik} Y_{jkl} Z_{il} = X_{ik} Y_{jkl} Z_{il}$$