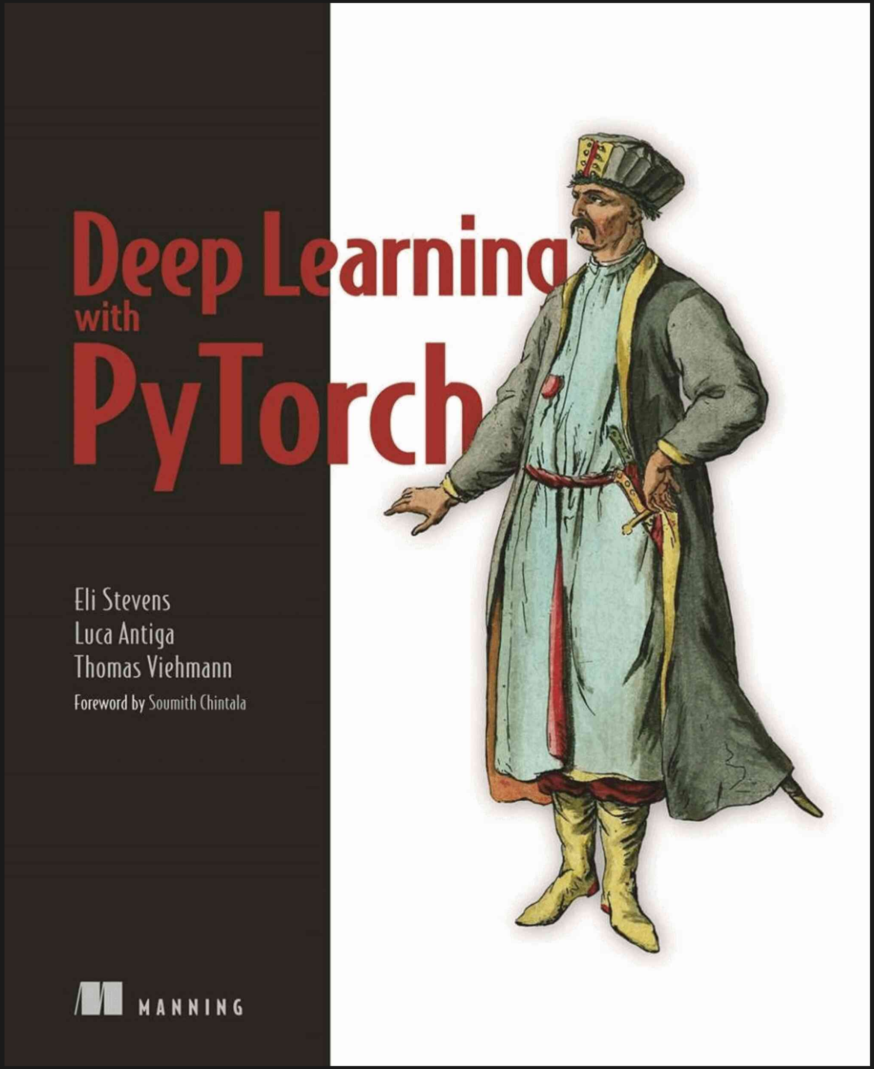
# Everything you wanted to know (and more) about

# PyTorch tensors

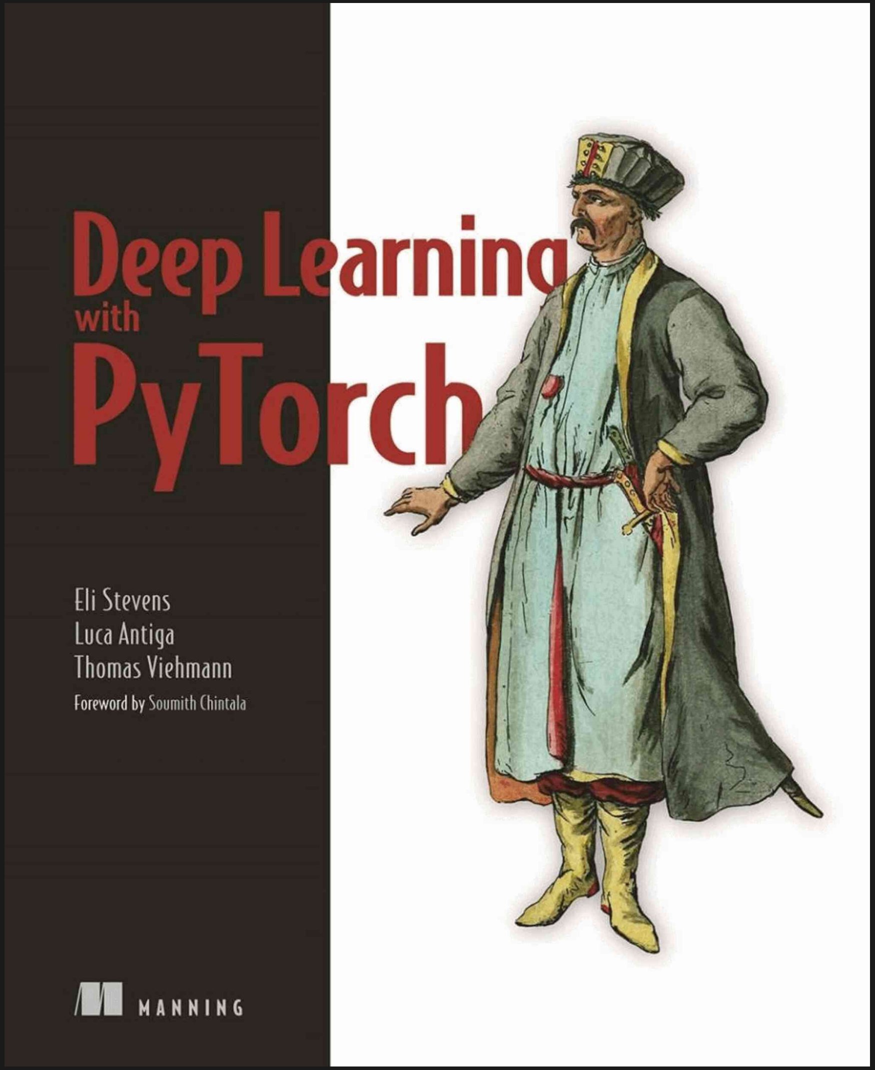
Marie-Hélène Burle
training@westgrid.ca

January 19, 2022

All drawings in this webinar come from the book:



The section on storage is also highly inspired by it



## Using tensors locally

You need to have Python and PyTorch installed

Additionally, you might want to use an IDE such as elpy if you are an Emacs user, JupyterLab , etc.

> Note that PyTorch does not yet support Python 3.10 except in some Linux distributions or on systems where a wheel has been built
>
> For the time being, you might have to use it with Python 3.9

# *Using tensors on CC clusters*

In the cluster terminal:

```
avail_wheels "torch*" # List available wheels & compatible Python versions
module avail python   # List available Python versions
module load python/3.9.6                # Load a sensible Python version
virtualenv --no-download env            # Create a virtual env
source env/bin/activate                 # Activate the virtual env
pip install --no-index --upgrade pip # Update pip
pip install --no-index torch            # Install PyTorch
```

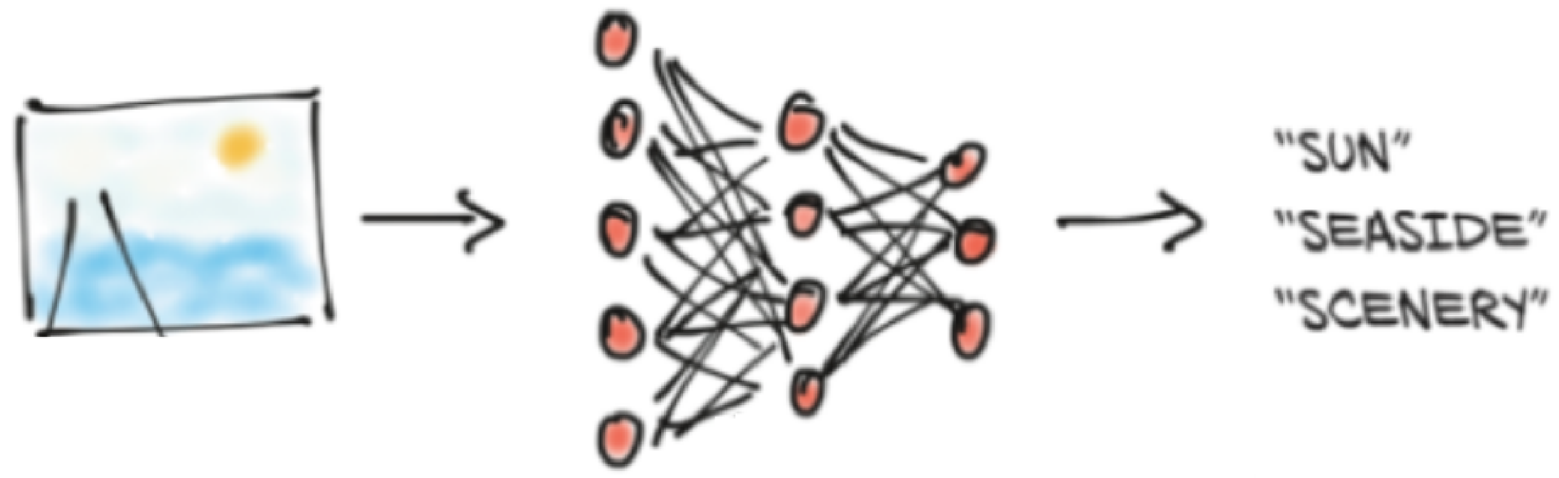
You can then run jobs with `sbatch` or `salloc`

You can leave the Python virtual env with `deactivate`

- What is a PyTorch tensor?
- Memory storage
- Data type (dtype)
- Basic operations
- Working with GPUs
- Working with NumPy ndarrays
- Linear algebra
- Distributed operations

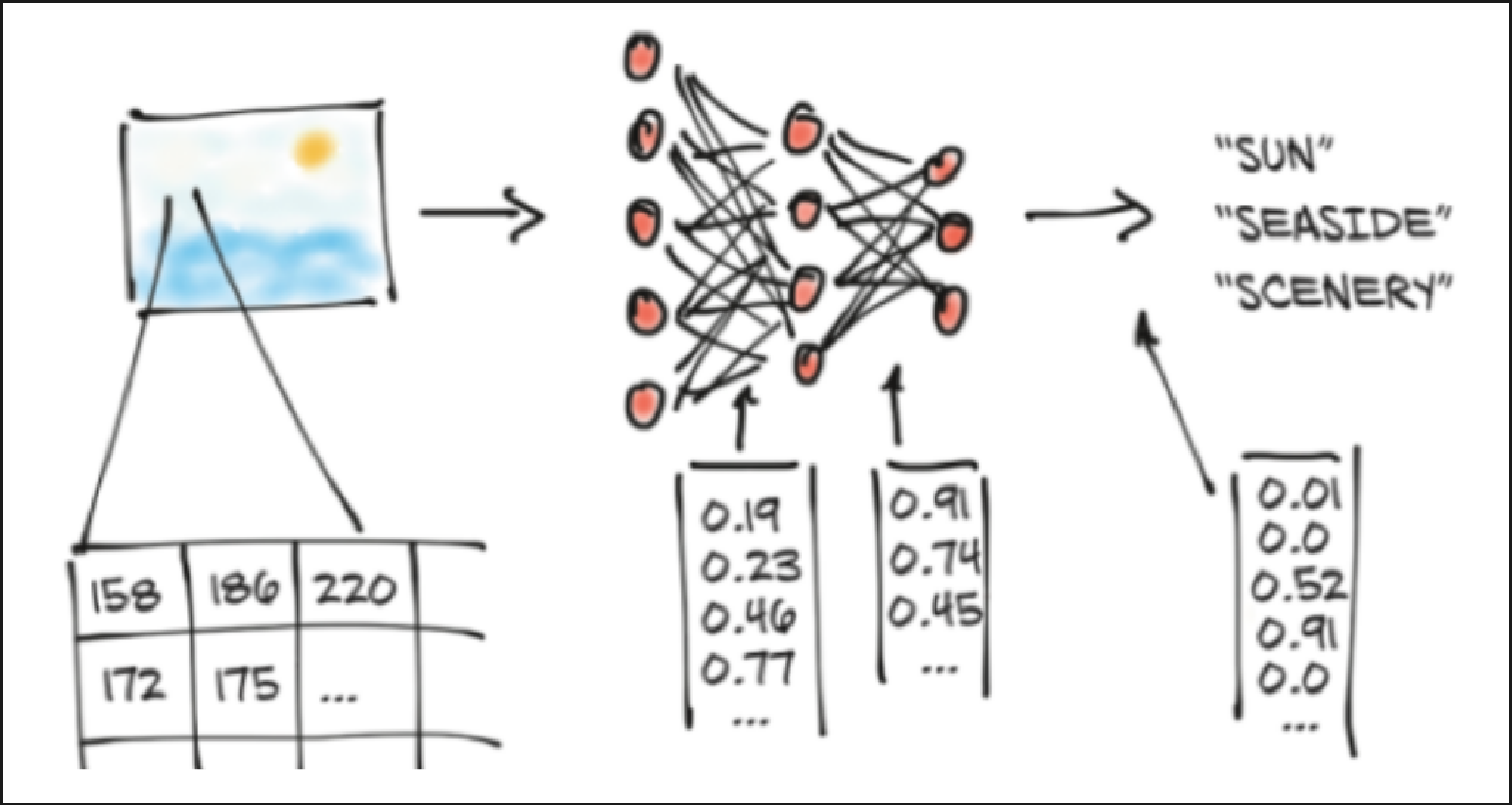
- **What is a PyTorch tensor?**
- Memory storage
- Data type (dtype)
- Basic operations
- Working with GPUs
- Working with NumPy ndarrays
- Linear algebra
- Distributed operations

## ANN do not process information directly



*Modified from Stevens, E., Antiga, L., & Viehmann, T. (2020). Deep learning with PyTorch. Manning Publications*

# It needs to be converted to numbers



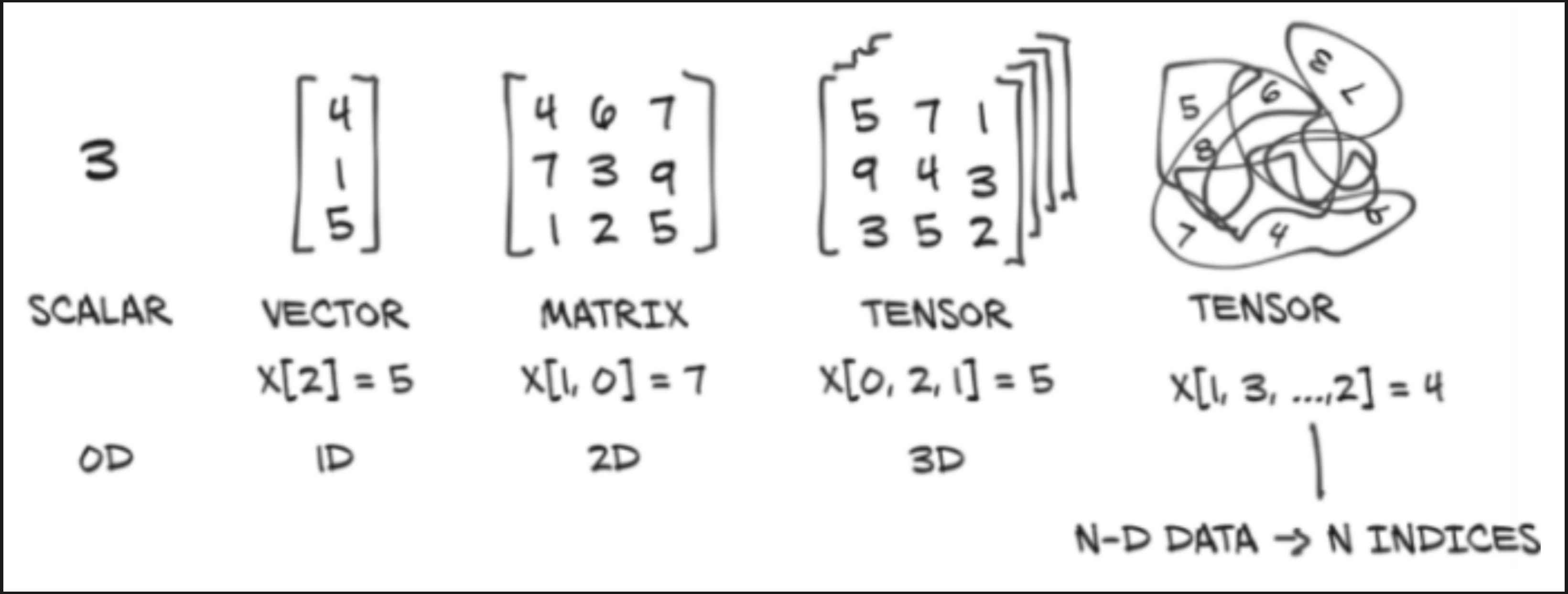
*Modified from Stevens, E., Antiga, L., & Viehmann, T. (2020). Deep learning with PyTorch. Manning Publications*

## All these numbers need to be stored in a data structure

*Stevens, E., Antiga, L., & Viehmann, T. (2020). Deep learning with PyTorch. Manning Publications*

## All these numbers need to be stored in a data structure

PyTorch tensors are Python objects holding multidimensional arrays



*Stevens, E., Antiga, L., & Viehmann, T. (2020). Deep learning with PyTorch. Manning Publications*

## Why a new object when NumPy ndarray already exists?

- Can run on GPUs
- Operations can be distributed
- Keep track of computation graphs for automatic differentiation

## What is a PyTorch tensor?

PyTorch is foremost a deep learning library

In deep learning, the information contained in objects of interest (e.g. images, texts, sounds) is converted to floating-point numbers (e.g. pixel values, token values, frequencies)

As this information is complex, multiple dimensions are required (e.g. 2 dimensions for the width & height of an image, plus 1 dimension for the 3 RGB colour channels)

Additionally, as various objects are grouped into batches to be processed together, batch size adds yet another dimension

Multidimensional arrays are thus particularly well suited for deep learning

## What is a PyTorch tensor?

Artificial neurons perform basic computations on these tensors

Their number however is huge & computing efficiency is paramount

GPUs are particularly well suited to perform many simple operations in parallel

The very popular NumPy library has at its core a mature multidimensional array object well integrated into the scientific Python ecosystem

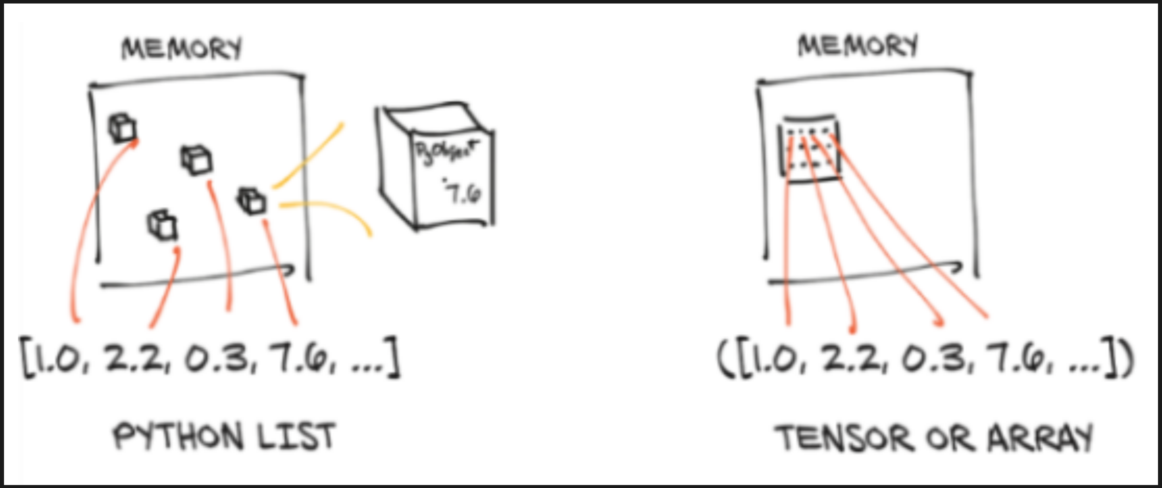
But the PyTorch tensor has additional efficiency characteristics ideal for machine learning & it can be converted to/from NumPy’s ndarray if needed

- What is a PyTorch tensor?
- Memory storage
- Data type (dtype)
- Basic operations
- Working with GPUs
- Working with NumPy ndarrays
- Linear algebra
- Distributed operations

## Efficient memory storage

In Python, collections (lists, tuples) are groupings of boxed Python objects

PyTorch tensors and NumPy ndarrays are made of unboxed C numeric types

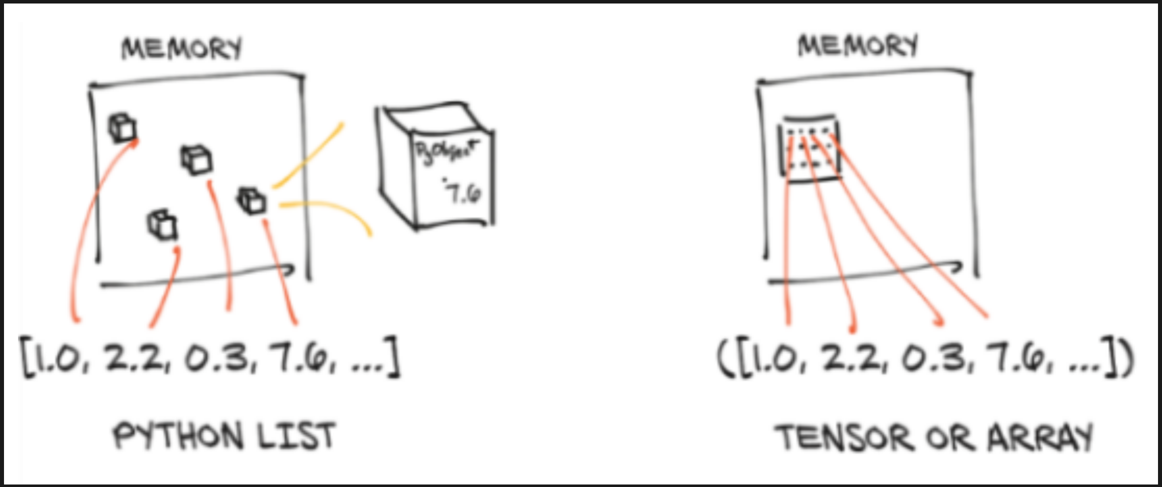


*Stevens, E., Antiga, L., & Viehmann, T. (2020). Deep learning with PyTorch. Manning Publications*

## Efficient memory storage

They are usually contiguous memory blocks, but the main difference is that they are unboxed: floats will thus take 4 or 8 bytes each

Boxed values take up more memory (memory for the pointer + memory for the primitive)



*Stevens, E., Antiga, L., & Viehmann, T. (2020). Deep learning with PyTorch. Manning Publications*

## Implementation

Under the hood, the values of a PyTorch tensor are stored as a `torch.Storage` instance which is a **one-dimensional array**

```
import torch
t = torch.arange(10.).view(2, 5); print(t) # Functions explained later
```

Output >>>

```
tensor([[ 0.,  1.,  2., 3.,  4.],
        [ 5.,  6.,  7.,  8.,  9.]])
```

## Implementation

```
storage = t.storage(); print(storage)
```

Output >>>

```
 0.0
 1.0
 2.0
 3.0
 4.0
 5.0
 6.0
 7.0
 8.0
 9.0
[torch.FloatStorage of size 10]
```

## Implementation

Storage can be indexed

```
storage[3]
```

Output >>>

```
3.0
```

## Implementation

```
storage[3] = 10.0; print(storage)
```

Output >>>

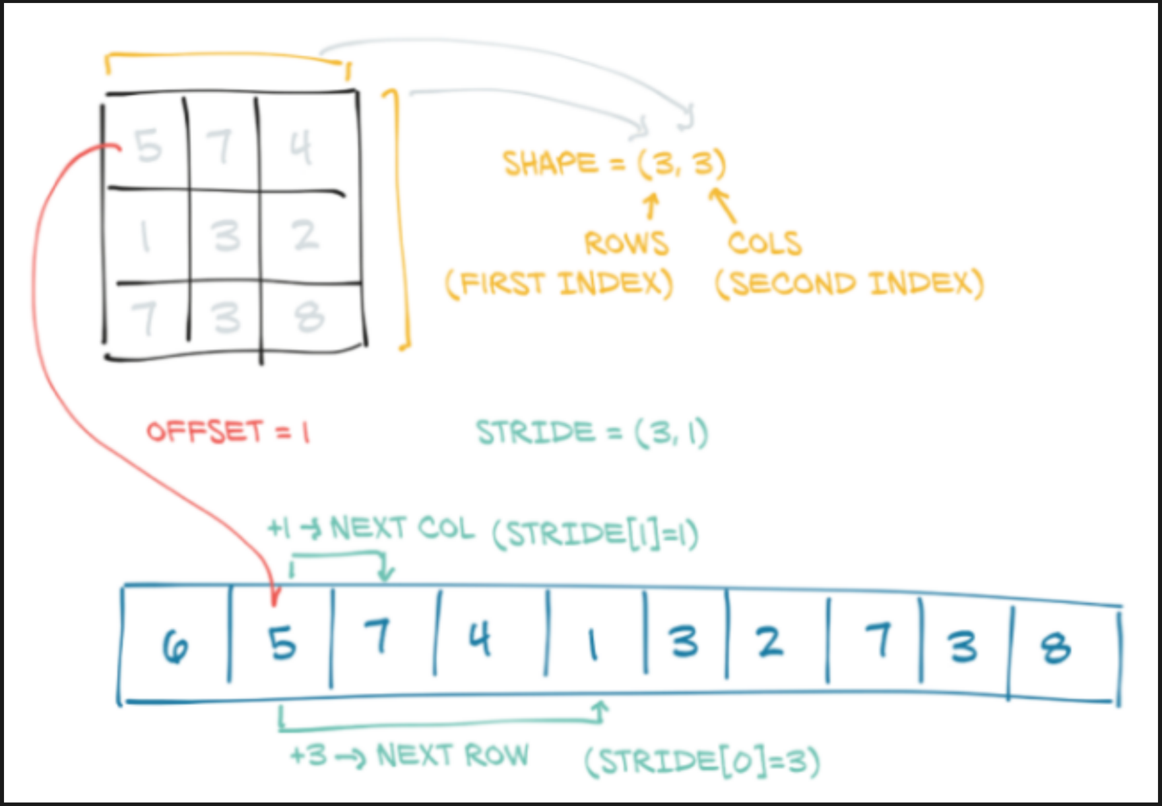
```
 0.0
 1.0
 2.0
 10.0
 4.0
 5.0
 6.0
 7.0
 8.0
 9.0
[torch.FloatStorage of size 10]
```

## Implementation

To view a multidimensional array from storage, we need **metadata** :

- the **size** (*shape* in NumPy) sets the number of elements in each dimension
- the **offset** indicates where the first element of the tensor is in the storage
- the **stride** establishes the increment between each element

# Storage metadata



*Stevens, E., Antiga, L., & Viehmann, T. (2020). Deep learning with PyTorch. Manning Publications*

# Storage metadata

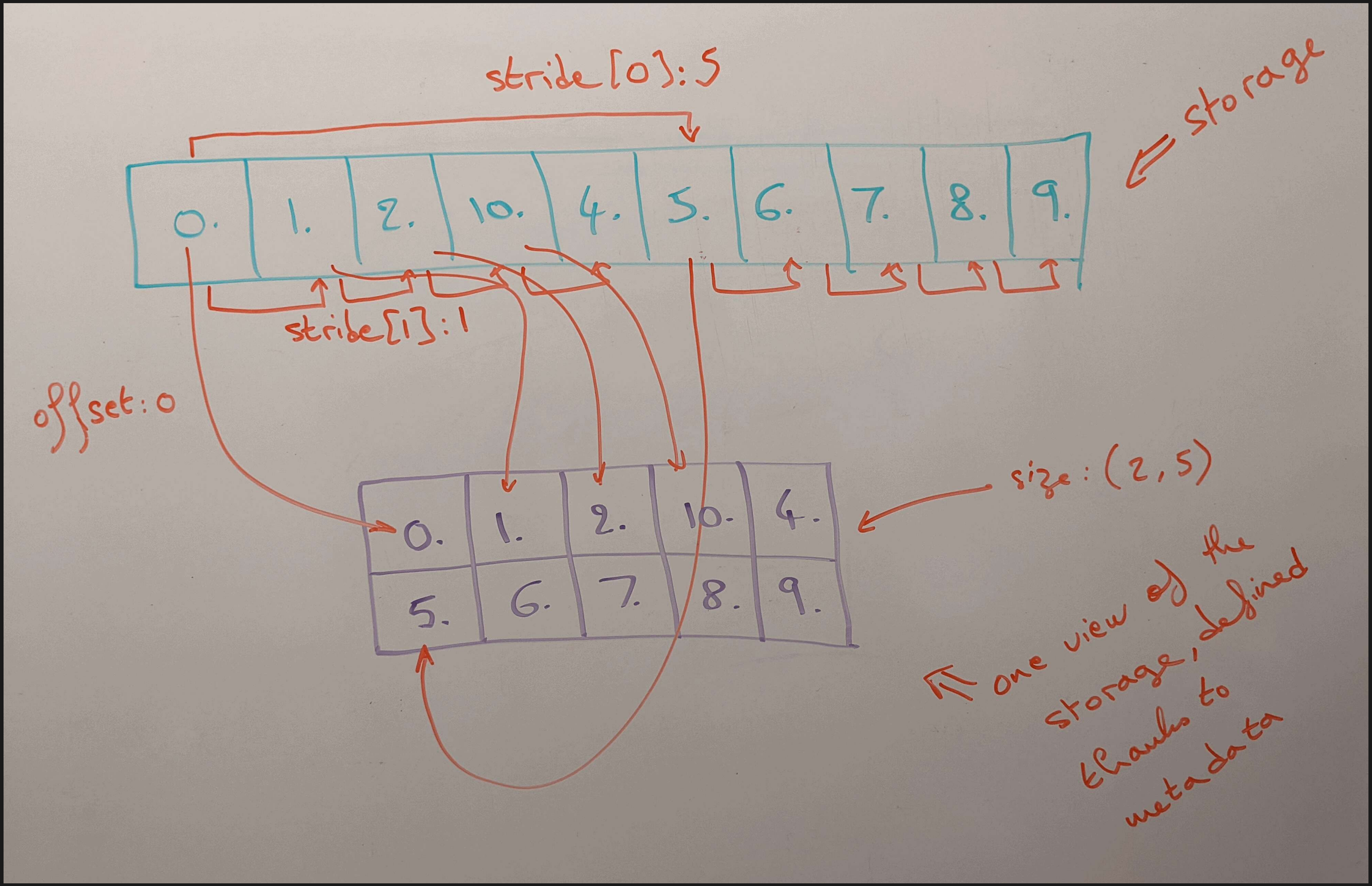
```
t.size()
t.storage_offset()
t.stride()
```

Output >>>

```
torch.Size([2, 5])
0
(5, 1)
```

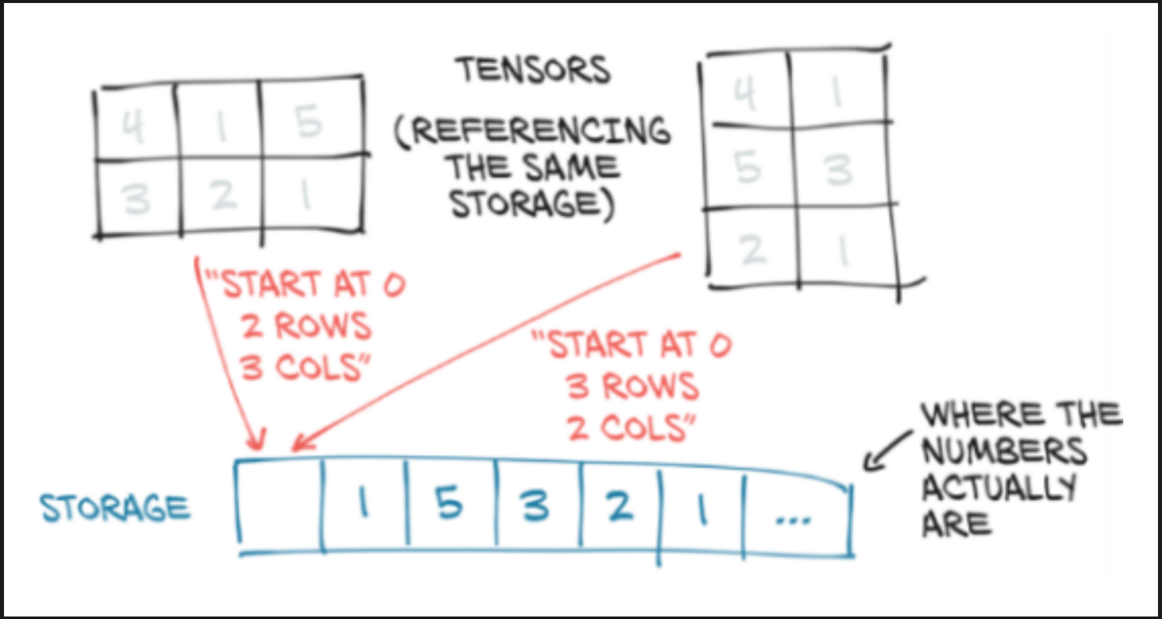
```
size: (2, 5)
offset: 0
stride: (5, 1)
```

# Storage metadata



## Sharing storage

Multiple tensors can use the same storage, saving a lot of memory since the metadata is a lot lighter than a whole new array



TENSORS
(REFERENCING THE SAME STORAGE)
4 1 5
3 2 1
4 1
5 3
2 1
"START AT 0 2 ROWS 3 COLS"
"START AT 0 3 ROWS 2 COLS"
STORAGE
4 1 5 3 2 1 ...
WHERE THE NUMBERS ACTUALLY ARE

*Stevens, E., Antiga, L., & Viehmann, T. (2020). Deep learning with PyTorch. Manning Publications*

## Transposing in 2 dimensions

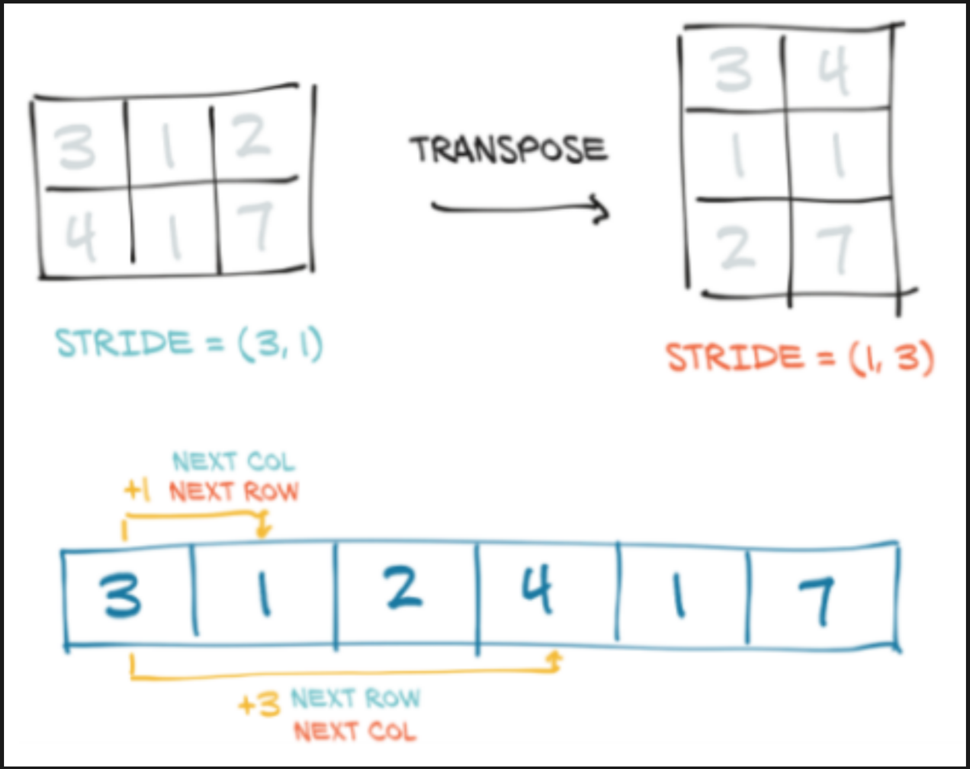
```
t = torch.tensor([[3, 1, 2], [4, 1, 7]]); print(t)
t.size()
t.t()
t.t().size()
```

Output >>>

```
tensor([[3, 1, 2],
        [4, 1, 7]])
torch.Size([2, 3])
tensor([[3, 4],
        [1, 1],
        [2, 7]])
torch.Size([3, 2])
```

## Transposing in 2 dimensions

= flipping the stride elements around



*Stevens, E., Antiga, L., & Viehmann, T. (2020). Deep learning with PyTorch. Manning Publications*

## Transposing in higher dimension tensors

`torch.t()` is a shorthand for `torch.transpose(0, 1)`:

```
torch.equal(t.t(), t.transpose(0, 1))
```

Output >>>

```
True
```

While `torch.t()` only works for 2D tensors, `torch.transpose()` can be used to transpose 2 dimensions in tensors of any number of dimensions

## Transposing in higher dimension tensors

```
t = torch.zeros(1, 2, 3); print(t)

t.size()
```

Output >>>

```
tensor([[[0., 0., 0.],
         [0., 0., 0.]]])

torch.Size([1, 2, 3])
```

## Transposing in higher dimension tensors

```
t.transpose(0, 1)


t.transpose(0, 1).size()
```

Output >>>

```
tensor([[[0., 0., 0.]],
        [[0., 0., 0.]]])


torch.Size([2, 1, 3])
```

## Transposing in higher dimension tensors

```
t.transpose(0, 2)


t.transpose(0, 2).size()
```

Output >>>

```
tensor([[[0.],
         [0.]],
        [[0.],
         [0.]],
        [[0.],
         [0.]]])


torch.Size([3, 2, 1])
```

## Transposing in higher dimension tensors

```
t.transpose(1, 2)


t.transpose(1, 2).size()
```

Output >>>

```
tensor([[[0., 0.],
         [0., 0.],
         [0., 0.]]])


torch.Size([1, 3, 2])
```

- What is a PyTorch tensor?
- Memory storage
- Data type (dtype)
- Basic operations
- Working with GPUs
- Working with NumPy ndarrays
- Linear algebra
- Distributed operations

## Default dtype

Since PyTorch tensors were built with utmost efficiency in mind for neural networks, the default data type is 32-bit floating points

This is sufficient for accuracy and much faster than 64-bit floating points

> Note that, by contrast, NumPy ndarrays use 64-bit as their default

# List of PyTorch tensor dtypes

| | |
|---|---|
| torch.float16 / torch.half | 16-bit / half-precision floating-point |
| torch.float32 / torch.float | 32-bit / single-precision floating-point |
| torch.float64 / torch.double | 64-bit / double-precision floating-point |
| torch.uint8 | unsigned 8-bit integers |
| torch.int8 | signed 8-bit integers |
| torch.int16 / torch.short | signed 16-bit integers |
| torch.int32 / torch.int | signed 32-bit integers |
| torch.int64 / torch.long | signed 64-bit integers |
| torch.bool | boolean |

## Checking & changing dtype

```
t = torch.rand(2, 3); print(t)
t.dtype   # remember that the default dtype for PyTorch tensors is float32
t2 = t.type(torch.float64); print(t2) # if dtype ≠ default, it is printed
t2.dtype
```

Output >>>

```
tensor([[0.8130, 0.3757, 0.7682],
        [0.3482, 0.0516, 0.3772]])
torch.float32
tensor([[0.8130, 0.3757, 0.7682],
        [0.3482, 0.0516, 0.3772]], dtype=torch.float64)
torch.float64
```

- What is a PyTorch tensor?
- Memory storage
- Data type (dtype)
- **Basic operations**
- Working with GPUs
- Working with NumPy ndarrays
- Linear algebra
- Distributed operations

# Creating tensors

| | |
|---|---|
| `torch.tensor`: | Input individual values |
| `torch.arange`: | Similar to `range` but creates a 1D tensor |
| `torch.linspace`: | 1D linear scale tensor |
| `torch.logspace`: | 1D log scale tensor |
| `torch.rand`: | Random numbers from a uniform distribution on `[0, 1)` |
| `torch.randn`: | Numbers from the standard normal distribution |
| `torch.randperm`: | Random permutation of integers |
| `torch.empty`: | Uninitialized tensor |
| `torch.zeros`: | Tensor filled with `0` |
| `torch.ones`: | Tensor filled with `1` |
| `torch.eye`: | Identity matrix |

## Creating tensors

```
torch.manual_seed(0)  # If you want to reproduce the result
torch.rand(1)

torch.manual_seed(0)  # Run before each operation to get the same result
torch.rand(1).item()  # Extract the value from a tensor
```

Output >>>

```
tensor([0.4963])

0.49625658988952637
```

## Creating tensors

```
torch.rand(1)
torch.rand(1, 1)
torch.rand(1, 1, 1)
torch.rand(1, 1, 1, 1)
```

Output >>>

```
tensor([0.6984])
tensor([[0.5675]])
tensor([[[0.8352]]])
tensor([[[[0.2056]]]])
```

## Creating tensors

```
torch.rand(2)
torch.rand(2, 2, 2, 2)
```

Output >>>

```
tensor([0.5932, 0.1123])
tensor([[[[0.1147, 0.3168],
          [0.6965, 0.9143]],
         [[0.9351, 0.9412],
          [0.5995, 0.0652]]],
        [[[0.5460, 0.1872],
          [0.0340, 0.9442]],
         [[0.8802, 0.0012],
          [0.5936, 0.4158]]]])
```

# Creating tensors

```
torch.rand(2)
torch.rand(3)
torch.rand(1, 1)
torch.rand(1, 1, 1)
torch.rand(2, 6)
```

Output >>>

```
tensor([0.7682, 0.0885])
tensor([0.1320, 0.3074, 0.6341])
tensor([[0.4901]])
tensor([[[0.8964]]])
tensor([[0.4556, 0.6323, 0.3489, 0.4017, 0.0223, 0.1689],
        [0.2939, 0.5185, 0.6977, 0.8000, 0.1610, 0.2823]])
```

## Creating tensors

```python
torch.rand(2, 4, dtype=torch.float64)  # You can set dtype
torch.ones(2, 1, 4, 5)
```

Output >>>

```
tensor([[0.6650, 0.7849, 0.2104, 0.6767],
        [0.1097, 0.5238, 0.2260, 0.5582]], dtype=torch.float64)
tensor([[[[1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1.]]],
        [[[1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1.]]]])
```

## Creating tensors

```
t = torch.rand(2, 3); print(t)
torch.zeros_like(t)                 # matches the size of t
torch.ones_like(t)
torch.randn_like(t)
```

Output >>>

```
tensor([[0.4051, 0.6394, 0.0871],
        [0.4509, 0.5255, 0.5057]])
tensor([[0., 0., 0.],
        [0., 0., 0.]])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[-0.3088, -0.0104,  1.0461],
        [ 0.9233,  0.0236, -2.1217]])
```

## Creating tensors

```
torch.arange(2, 10, 4)    # from 2 to 10 in increments of 4
torch.linspace(2, 10, 4)  # 4 elements from 2 to 10 on the linear scale
torch.logspace(2, 10, 4)  # Same on the log scale
torch.randperm(4)
torch.eye(3)
```

Output >>>

```
tensor([2, 6])
tensor([2.0000,  4.6667,  7.3333, 10.0000])
tensor([1.0000e+02, 4.6416e+04, 2.1544e+07, 1.0000e+10])
tensor([1, 3, 2, 0])
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

## Tensor indexing

```
x = torch.rand(3, 4)
x[:]                    # With a range, the comma is implicit: same as x[:, ]
x[:, 2]
x[1, :]
x[2, 3]
```

Output >>>

```
tensor([[0.6575, 0.4017, 0.7391, 0.6268],
        [0.2835, 0.0993, 0.7707, 0.1996],
        [0.4447, 0.5684, 0.2090, 0.7724]])
tensor([0.7391, 0.7707, 0.2090])
tensor([0.2835, 0.0993, 0.7707, 0.1996])
tensor(0.7724)
```

# Tensor indexing

```
x[-1:]         # Last element (implicit comma, so all columns)
x[-1]          # No range, no implicit comma: we are indexing
# from a list of tensors, so the result is a one dimensional tensor
# (Each dimension is a list of tensors of the previous dimension)
x[-1].size()  # Same number of dimensions than x (2 dimensions)
x[-1:].size() # We dropped one dimension
```

Output >>>

```
tensor([[0.8168, 0.0879, 0.2642, 0.3777]])
tensor([0.8168, 0.0879, 0.2642, 0.3777])

torch.Size([4])
torch.Size([1, 4])
```

## Tensor indexing

```
x[0:1]     # Python ranges are inclusive to the left, not the right
x[:-1]     # From start to one before last (and implicit comma)
x[0:3:2]   # From 0th (included) to 3rd (excluded) in increment of 2
```

Output >>>

```
tensor([[0.5873, 0.0225, 0.7234, 0.4538]])
tensor([[0.5873, 0.0225, 0.7234, 0.4538],
        [0.9525, 0.0111, 0.6421, 0.4647]])
tensor([[0.5873, 0.0225, 0.7234, 0.4538],
        [0.8168, 0.0879, 0.2642, 0.3777]])
```

## Tensor indexing

```
x[None]          # Adds a dimension of size one as the 1st dimension
x.size()
x[None].size()
```

Output >>>

```
tensor([[[0.5873, 0.0225, 0.7234, 0.4538],
         [0.9525, 0.0111, 0.6421, 0.4647],
         [0.8168, 0.0879, 0.2642, 0.3777]]])
torch.Size([3, 4])
torch.Size([1, 3, 4])
```

## Tensor information

```python
t = torch.rand(2, 3); print(t)
t.size()
t.dim()
t.numel()
```

Output >>>

```
tensor([[0.5885, 0.7005, 0.1048],
        [0.1115, 0.7526, 0.0658]])
torch.Size([2, 3])
2
6
```

## Vectorized operations

Since PyTorch tensors are homogeneous (i.e. made of a single data type), as with NumPy's ndarrays , operations are vectorized & thus staggeringly fast

## Simple mathematical operations

```
t1 = torch.arange(1, 5).view(2, 2); print(t1)
t2 = torch.tensor([[1, 1], [0, 0]]); print(t2)
t1 + t2 # Operation performed between elements at corresponding locations
t1 + 1  # Operation applied to each element of the tensor
```

Output >>>

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

## Reduction

```python
t = torch.ones(2, 3, 4); print(t)
t.sum()   # Reduction over all entries
```

Output >>>

```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],
        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
tensor(24.)
```

> Other reduction functions (e.g. mean) behave the same way

## Reduction

```
# Reduction over a specific dimension
t.sum(0)
t.sum(1)
t.sum(2)
```

Output >>>

```
tensor([[2., 2., 2., 2.],
        [2., 2., 2., 2.],
        [2., 2., 2., 2.]])
tensor([[3., 3., 3., 3.],
        [3., 3., 3., 3.]])
tensor([[4., 4., 4.],
        [4., 4., 4.]])
```

## Reduction

```
# Reduction over multiple dimensions
t.sum((0, 1))
t.sum((0, 2))
t.sum((1, 2))
```

Output >>>

```
tensor([6., 6., 6., 6.])
tensor([8., 8., 8.])
tensor([12., 12.])
```

## In-place operations

With operators post-fixed with `_`:

```
t1 = torch.tensor([1, 2]); print(t1)
t2 = torch.tensor([1, 1]); print(t2)
t1.add_(t2); print(t1)  # Same as t1 = t1 + t2 or t1 = t1.add(t2)
t1.zero_(); print(t1)
```

Output >>>

```
tensor([1, 2])
tensor([1, 1])
tensor([2, 3])
tensor([0, 0])
```

## Tensor views

```python
t = torch.tensor([[1, 2, 3], [4, 5, 6]]); print(t)
t.size()
t.view(6)
t.view(3, 2)
t.view(3, -1) # Same: with -1, the size is inferred from other dimensions
```

Output >>>

```
tensor([[1, 2, 3],
        [4, 5, 6]])
torch.Size([2, 3])
tensor([1, 2, 3, 4, 5, 6])
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

## *Note the difference*

```
t1 = torch.tensor([[1, 2, 3], [4, 5, 6]]); print(t1)
t2 = t1.t(); print(t2)
t3 = t1.view(3, 2); print(t3)
```

Output >>>

```
tensor([[1, 2, 3],
        [4, 5, 6]])
tensor([[1, 4],
        [2, 5],
        [3, 6]])
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

## Logical operations

```
t1 = torch.randperm(5); print(t1)
t2 = torch.randperm(5); print(t2)
t1 > 3                                  # Test each element
t1 < t2                                 # Test corresponding pairs of elements
```

Output >>>

```
tensor([4, 1, 0, 2, 3])
tensor([0, 4, 2, 1, 3])
tensor([ True, False, False, False, False])
tensor([False,  True,  True, False, False])
```

- What is a PyTorch tensor?
- Memory storage
- Data type (dtype)
- Basic operations
- Working with GPUs
- Working with NumPy ndarrays
- Linear algebra
- Distributed operations

## Device attribute

Tensor data can be placed in the memory of various processor types:

- the RAM of CPU
- the RAM of a GPU with CUDA support
- the RAM of a GPU with AMD's ROCm support
- the RAM of an XLA device (e.g. Cloud TPU ) with the torch_xla package

## Device attribute

The values for the device attributes are:

- CPU: `'cpu'`
- GPU (CUDA and AMD’s ROCm): `'cuda'`
- XLA: `xm.xla_device()`

This last option requires to load the torch_xla package first:

```python
import torch_xla
import torch_xla.core.xla_model as xm
```

## Creating a tensor on a specific device

```
t_gpu = torch.rand(2, 3, device='cuda')
```

## Copying a tensor to a specific device

```
t_cpu = t_gpu.to(device='cpu')     # Move to CPU
t_gpu = t_cpu.to(device='cuda')    # Move to GPU
t_gpu = t_cpu.to(device='cuda:0')  # Move to a specific GPU
t_gpu = t_cpu.to(device='cuda:1')  # Move to a specific GPU
```

Or the equivalent short forms:

```
t_cpu = t_gpu.cpu()                # Move to CPU
t_gpu = t_cpu.cuda()               # Move to GPU
t_gpu = t_cpu.cuda(0)              # Move to a specific GPU
t_gpu = t_cpu.cuda(1)              # Move to a specific GPU
```

- What is a PyTorch tensor?
- Memory storage
- Data type (dtype)
- Basic operations
- Working with GPUs
- **Working with NumPy ndarrays**
- Linear algebra
- Distributed operations

## Working with NumPy

PyTorch tensors can be converted to NumPy ndarrays and vice-versa in a very efficient manner as both objects will share the same memory

```
t = torch.rand(2, 3)
t_np = t.numpy()
t2 = torch.from_numpy(t_np)
torch.equal(t, t2)
```

> Remember that PyTorch tensors use 32-bit floating points by default
> While NumPy uses 64-bit by default
> In neural networks, 32-bit is what you want to use

> Note that NumPy arrays only work on CPU, so to convert a tensor allocated to the GPU, the content will be copied to the CPU first

- What is a PyTorch tensor?
- Memory storage
- Data type (dtype)
- Basic operations
- Working with GPUs
- Working with NumPy ndarrays
- **Linear algebra**
- Distributed operations

## torch.linalg **module**

- All functions from **numpy.linalg** implemented
  (with accelerator & automatic differentiation support)

- Some additional functions

> Requires torch >= 1.9
> Linear algebra support less developed before introduction of this module

## System of linear equations solver

Let’s have a look at an extremely basic example:

```
2x + 3y - z = 5
x - 2y + 8z = 21
6x + y - 3z = -1
```

We are looking for the values of `x`, `y`, and `z` that would satisfy this system

## System of linear equations solver

We create a 2D tensor `A` of size `(3, 3)` with the coefficients of the equations & a 1D tensor `b` of size `3` with the right hand sides values of the equations

```
A = torch.tensor([[2., 3., -1.], [1., -2., 8.], [6., 1., -3.]]); print(A)
b = torch.tensor([5., 21., -1.]); print(b)
```

Output >>>

```
tensor([[ 2.,  3., -1.],
        [ 1., -2.,  8.],
        [ 6.,  1., -3.]])
tensor([ 5., 21., -1.])
```

## System of linear equations solver

Solving this system is as simple as running the `torch.linalg.solve` function:

```
x = torch.linalg.solve(A, b); print(x)
```

Output >>>

```
tensor([1., 2., 3.])
```

Our solution is:

$$\begin{aligned} x &= 1 \\ y &= 2 \\ z &= 3 \end{aligned}$$

## Verify our result

```
torch.allclose(A @ x, b)
```

Output >>>

```
True
```

## System of linear equations solver

Here is another simple example:

```
# Create a square normal random matrix
A = torch.randn(4, 4); print(A)
# Create a tensor of right hand side values
b = torch.randn(4); print(b)

# Solve the system
x = torch.linalg.solve(A, b); print(x)

# Verify
torch.allclose(A @ x, b)
```

## System of linear equations solver

Output >>>

```
tensor([[ 1.5091,  2.0820,  1.7067,  2.3804], # A (coefficients)
        [-1.1256, -0.3170, -1.0925, -0.0852],
        [ 0.3276, -0.7607, -1.5991,  0.0185],
        [-0.7504,  0.1854,  0.6211,  0.6382]])

tensor([-1.0886, -0.2666,  0.1894, -0.2190])  # b (right hand side values)

tensor([ 0.1992, -0.7011,  0.2541, -0.1526])  # x (our solution)

True                                          # Verification
```

## With 2 multidimensional tensors

```python
A = torch.randn(2, 3, 3)                # Must be batches of square matrices
B = torch.randn(2, 3, 5)                # Dimensions must be compatible
X = torch.linalg.solve(A, B); print(X)
torch.allclose(A @ X, B)
```

Output >>>

```
tensor([[[-0.0545, -0.1012,  0.7863, -0.0806, -0.0191],
         [-0.9846, -0.0137, -1.7521, -0.4579, -0.8178],
         [-1.9142, -0.6225, -1.9239, -0.6972,  0.7011]],
        [[ 3.2094,  0.3432, -1.6604, -0.7885,  0.0088],
         [ 7.9852,  1.4605, -1.7037, -0.7713,  2.7319],
         [-4.1979,  0.0849,  1.0864,  0.3098, -1.0347]]])
True
```

## Matrix inversions

> It is faster and more numerically stable to solve a system of linear equations directly than to compute the inverse matrix first

**Limit matrix inversions to situations where it is truly necessary**

## Matrix inversions

```
A = torch.rand(3, 3, 3)
A_inv = torch.linalg.inv(A)
A @ A_inv
```

Output >>>

```
tensor([[[ 1.0000e+00, -6.0486e-07,  1.3859e-06],
         [ 5.5627e-08,  1.0000e+00,  1.0795e-06],
         [-1.4133e-07,  7.9992e-08,  1.0000e+00]],
        [[ 1.0000e+00, -1.3301e-07, -1.3090e-07],
         [ 4.2567e-08,  1.0000e+00, -5.0706e-09],
         [ 3.2889e-08, -2.6357e-09,  1.0000e+00]],
        [[ 1.0000e+00,  4.3329e-08, -3.6741e-09],
         [-7.4627e-08,  1.0000e+00,  1.4579e-07],
         [-6.3580e-08,  8.2354e-08,  1.0000e+00]]])
```

## Other linear algebra functions

torch.linalg contains many more functions, including for instance:

- torch.tensordot which generalizes matrix products
- torch.linalg.tensorsolve which computes the solution `X` to the system `torch.tensordot(A, X) = B`
- torch.linalg.eigvals which computes the eigenvalues of a square matrix

- What is a PyTorch tensor?
- Memory storage
- Data type (dtype)
- Basic operations
- Working with GPUs
- Working with NumPy ndarrays
- Linear algebra
- Distributed operations

## Parallel tensor operations

PyTorch already allows for distributed training of ML models

The implementation of distributed tensor operations—for instance for linear algebra—is in the work through the use of a ShardedTensor primitive that can be sharded across nodes

See also this issue for more comments about upcoming developments on (among other things) tensor sharding

Questions?