Lecture 4:
Introduction to PyTorch

David Völgyes
david.volgyes@ieee.org

February 5, 2020

About today

- You will get an introduction to PyTorch.
- PyTorch is a widely used deep learning framework, especially in academia.
- PyTorch version 1.0-1.4

Remark:

- There is a new PyTorch release in every 2-3 months.
    - 5 releases since last year
    - most likely at least two new will be released during the semester
- We use PyTorch version 1.x,
  but the syntax did not change in between 1.0 - 1.4 significantly.
- Some tutorials use Python 2. Stick to Python 3.6+.

Outline

- Deep learning frameworks
- PyTorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

A simplified workflow in supervised learning

- creating dataset
- creating a neural network (model)
- defining a loss function
- loading samples (data loader)
- predicting with the model
- comparison of the prediction and the target (loss)
- backpropagation: calculating gradients from the error
- updating the model (optimizer)
- checking the loss: if it is low enough, stop training

Readings

Highly recommended (by the end of the semester):

- Pytorch tutorials: https://pytorch.org/tutorials/
  - Deep Learning with PyTorch: A 60 Minute Blitz
    https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- Pytorch cheat sheet: https://pytorch.org/tutorials/beginner/ptcheat.html
- Broadcasting: https://pytorch.org/docs/stable/notes/broadcasting.html

Overwhelming, but good additional source for anything:

- Awesome PyTorch list: https://github.com/bharathgs/Awesome-PyTorch-list
  It is a collection of hundred of links, including tutorials, research papers, libraries, etc.

Note:

- Don’t get confused. A lot of the available code online is written in an older version of PyTorch (mostly in 0.3-0.4).

Progress

- **Deep learning frameworks**
- PyTorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

Why do we need Deep learning frameworks?

- **Speed:**
  - Fast GPU/CPU implementation of matrix multiplication, convolutions and backpropagation
- **Automatic differentiations:**
  - Pre-implementation of the most common functions and their gradients.
- **Reuse:**
  - Easy to reuse other people's models
- **Less error prone:**
  - The more code you write yourself, the more errors

Deep learning frameworks

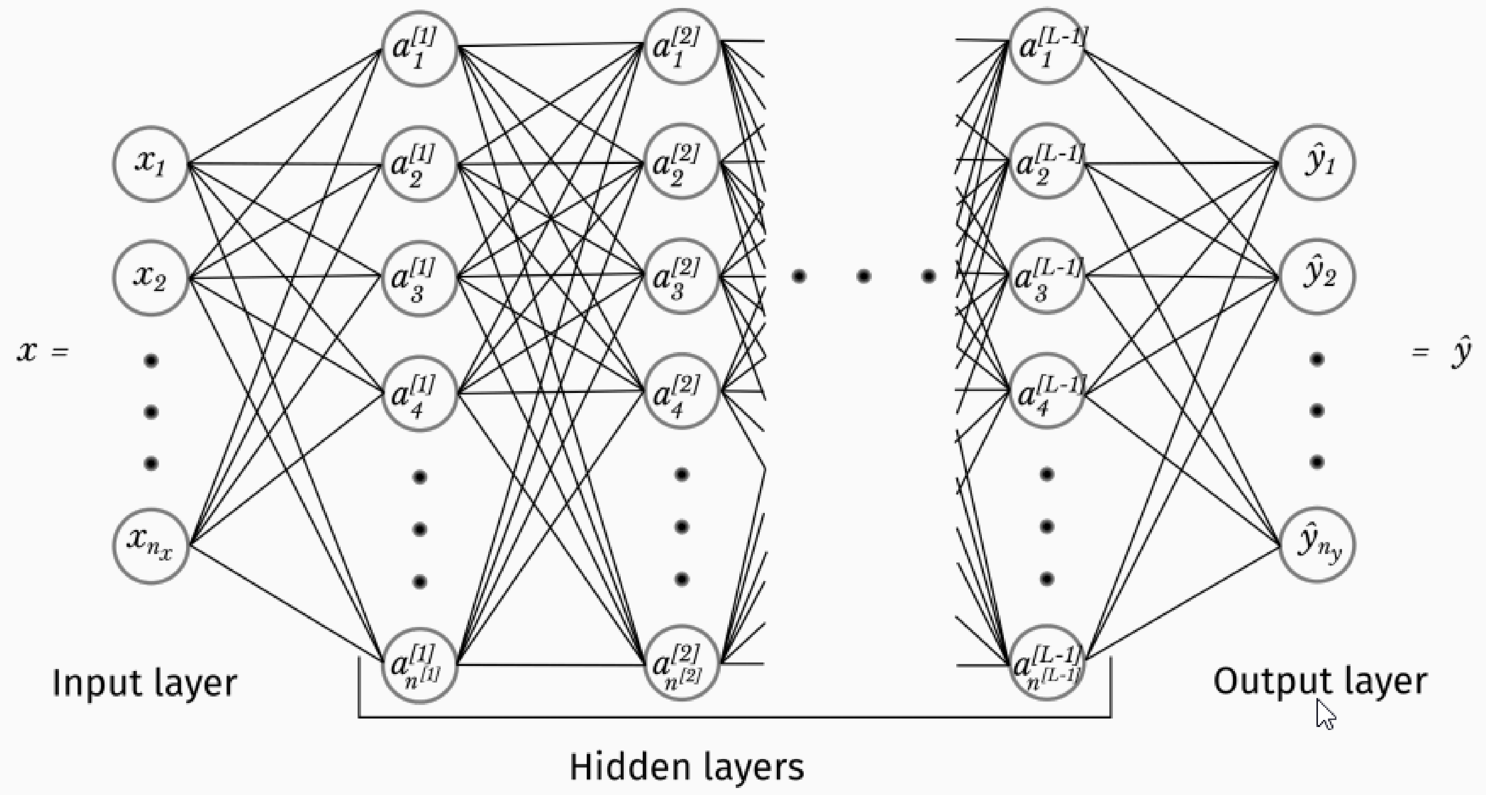
Deep learning frameworks does a lot of the complicated computation, remember last week.

$$z_k^{[L]} = \sum_{j=1}^{n^{[L-1]}} w_{jk}^{[L]} a_j^{[L-1]} + b_k^{[L]}$$

$$a_k^{[L]} = s(z_k^{[L]})$$
$$= \hat{y}_k$$

for

$$k = 1, \dots, n_y,$$
$$= 1, \dots, n^{[L]}.$$

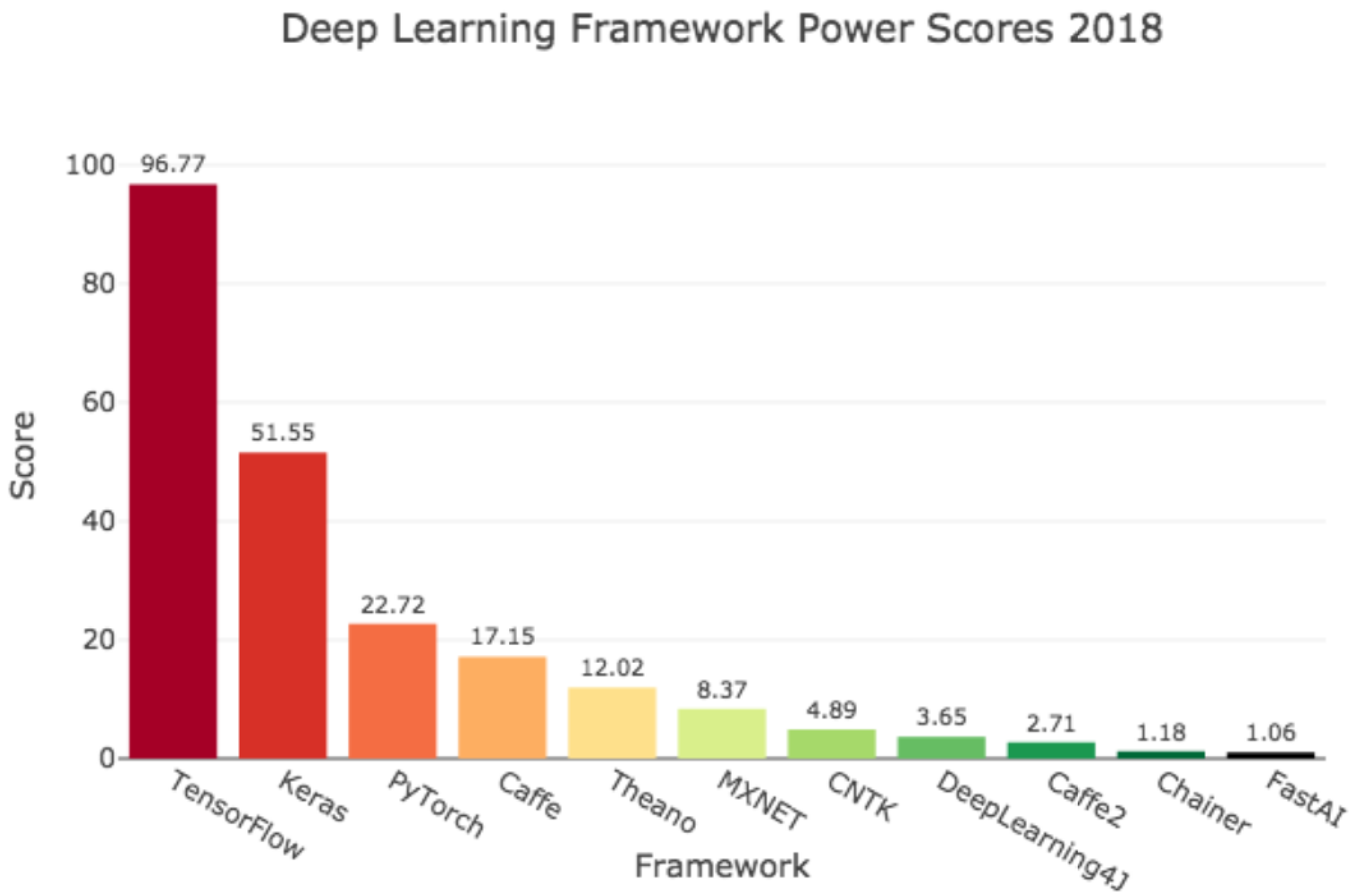


Major frameworks

- **pytorch** (developed by Facebook)
- Tensorflow (developed by Google)
- Caffe (developed by Facebook)
- MXNet
- MS Cognitive Toolkit (CNTK)
- Chainer

Remark: all of them are open source.

Popularity


Deep Learning Framework Power Scores 2018
Score
Framework
100
80
60
40
20
0
96.77
51.55
22.72
17.15
12.02
8.37
4.89
3.65
2.71
1.18
1.06
TensorFlow
Keras
PyTorch
Caffe
Theano
MXNET
CNTK
DeepLearning4J
Caffe2
Chainer
FastAI

Popularity, late 2019

- Industry is still dominated by Tensorflow / Keras
- But 70-75% of the academic papers are in PyTorch (last year: ~ 20%)
- Major projects change to PyTorch, e.g. OpenAI

Why PyTorch

- Python API
- Can use CPU, GPU (CUDA only)
- Supports common platforms:
  - Windows, iOS, Linux
- PyTorch is a thin framework which lets you work closely with programming the neural network
- Focus on the machine learn part not the framework itself
- Pythonic control flow
  - Flexible
  - Cleaner and more intuitive code
  - Easy to debug
- Python debugger
  - With PyTorch we can use the python debugger
  - It does not run all in a C++ environment abstracted way

Installing PyTorch

```
conda create -n IN5400 python=3.8 PyTorch torchvision cudatoolkit=10.1 jupyter ipython matplotlib scikit-learn -c PyTorch
```

```
#
```

Installing PyTorch

Without CUDA:

```
conda create -n IN5400 python=3.8 PyTorch torchvision cpuonly jupyter ipython matplotlib scikit-learn -c PyTorch
```

Installation instructions:
https://pytorch.org/get-started/locally/

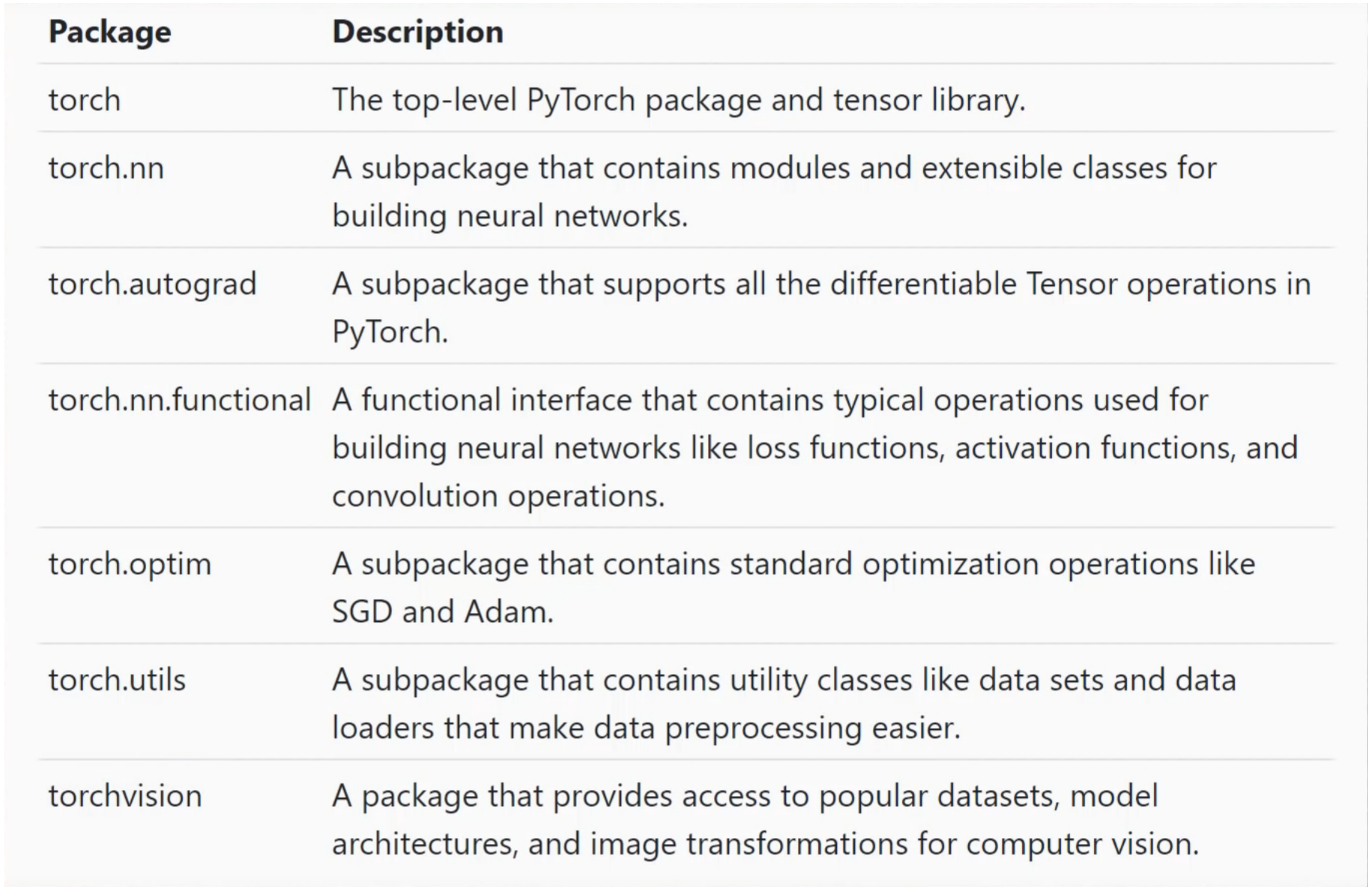
Older versions:
https://pytorch.org/get-started/previous-versions/

Remember: during the semester probably there will be at least two new releases!

Checking PyTorch installation

```
>>> import numpy as np
>>> import torch
>>> import sys
>>> import matplotlib
>>> print(f'Python  version: {sys.version}')
Python  version: 3.8.1 | packaged by conda-forge | (default, Jan 29 2020, 14:55:04) [GCC 7.3.0]

>>> print(f'Numpy   version: {np.version.version}')
Numpy   version: 1.17.5

>>> print(f'PyTorch version: {torch.version.__version__}')
PyTorch version: 1.4.0

>>> print(f'Matplotlib version: {matplotlib.__version__}')
Matplotlib version: 3.1.2

>>> print(f'GPU present: {torch.cuda.is_available()}')
GPU present: False
```

PyTorch packages



Package	Description
torch	The top-level PyTorch package and tensor library.
torch.nn	A subpackage that contains modules and extensible classes for building neural networks.
torch.autograd	A subpackage that supports all the differentiable Tensor operations in PyTorch.
torch.nn.functional	A functional interface that contains typical operations used for building neural networks like loss functions, activation functions, and convolution operations.
torch.optim	A subpackage that contains standard optimization operations like SGD and Adam.
torch.utils	A subpackage that contains utility classes like data sets and data loaders that make data preprocessing easier.
torchvision	A package that provides access to popular datasets, model architectures, and image transformations for computer vision.

Progress

- Deep learning frameworks
- PyTorch
  - **torch.tensor**
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

torch.Tensor class

- PyTorch ’s tensors are very similar to NumPy’s ndarrays
- but they have a device, 'cpu', 'cuda', or 'cuda:X'
- they might require gradients

```
>>> t = torch.tensor([1,2,3], device='cpu',
...         requires_grad=False,dtype=torch.float32)
>>> print(t.dtype)
torch.float32
>>> print(t.device)
cpu
>>> print(t.requires_grad)
False
>>> t2 = t.to(torch.device('cuda'))
>>> t3 = t.cuda()  # or you can use shorthand
>>> t4 = t.cpu()
```

See: https://pytorch.org/docs/stable/tensors.html

Pytorch 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
Boolean	torch.bool	torch.BoolTensor	torch.cuda.BoolTensor

Conversion in numpy and in PyTorch:

```
new_array  = old_array.astype(np.int8)  # numpy array
new_tensor = old_tensor.to(torch.int8)  # torch tensor
```

Remarks: Almost always torch.float32 or torch.int64 are used.
Half does not work on CPUs and on many GPUs (hardware limitation).

See: https://pytorch.org/docs/stable/tensors.html

Numpy-PyTorch functions

Creating arrays / tensor:

- **eye**: creating diagonal matrix / tensor
- **zeros**: creating tensor filled with zeros
- **ones**: creating tensor filled with ones
- **linspace**: creating linearly increasing values
- **arange**: linearly increasing integers

For instance:

```
>>> torch.eye(3, dtype=torch.double)
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]], dtype=torch.float64)
>>> torch.arange(6)
tensor([0, 1, 2, 3, 4, 5])
```

PyTorch functions, dimensionality

```
x.size()                      #* return tuple-like object of dimensions, old codes
x.shape                       #  return tuple-like object of dimensions, numpy style
x.ndim                        #  number of dimensions, also known as .dim()
x.view(a,b,...)               #* reshapes x into size (a,b,...)
x.view(-1,a)                  #* reshapes x into size (b,a) for some b
x.reshape(a,b,...)            # equivalent with .view()
x.transpose(a,b)              # swaps dimensions a and b
x.permute(*dims)              # permutes dimensions; missing in numpy
x.unsqueeze(dim)              # tensor with added axis; missing in numpy
x.unsqueeze(dim=2)            # (a,b,c) tensor -> (a,b,1,c) tensor; missing in numpy
torch.cat(tensor_seq, dim=0)  # concatenates tensors along dim
# For instance:
>>>t = torch.arange(6)
tensor([0, 1, 2, 3, 4, 5])
>>> t.reshape(2,3) # same as t.view(2,3 or t.view(2,-1)
tensor([[0, 1, 2],
       [3, 4, 5]])
>>> t.reshape(2,3).unsqueeze(1)
tensor([[[0, 1, 2]],
       [[3, 4, 5]]])
>>> t.reshape(2,3).unsqueeze(1).shape
torch.Size([2, 1, 3])
```

Indexing

Standard numpy indexing works:

```
>>> t = torch.arange(12).reshape(3,4)
tensor([[ 0,  1,  2,  3],
[ 4,  5,  6,  7],
[ 8,  9, 10, 11]])
>>> t[1,1:3]
tensor([5, 6])
>>> t[:,:] = 0 # fill everything with 0, a.k.a. t.fill_(0)
tensor([[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0]])
```

Broadcasting semantics

If two tensors x, y are "broadcastable", the resulting tensor size is calculated as follows:

- If the number of dimensions of x and y are not equal, prepend 1 to the dimensions of the tensor with fewer dimensions to make them equal length.
- Then, for each dimension size, the resulting dimension size is the max of the sizes of x and y along that dimension.

See: https://pytorch.org/docs/stable/notes/broadcasting.html

Broadcasting semantics example

```
# can line up trailing dimensions to make reading easier
>>> x=torch.empty(5,1,4,1)
>>> y=torch.empty(  3,1,1)
>>> (x+y).size()
torch.Size([5, 3, 4, 1])

# but not necessary:
>>> x=torch.empty(1)
>>> y=torch.empty(3,1,7)
>>> (x+y).size()
torch.Size([3, 1, 7])

>>> x=torch.empty(5,2,4,1)
>>> y=torch.empty(3,1,1)
>>> (x+y).size()
RuntimeError: The size of tensor a (2) must match
the size of tensor b (3) at non-singleton dimension 1
```

Be very careful, e.g. when you calculate loss!

Memory: Sharing vs Copying

Some operations share underlying memory, some create new tensors.

Copy Data:
torch.Tensor()
torch.tensor()
torch.clone()
type casting

Share Data
torch.as_tensor()
torch.from_numpy()
torch.view()
torch.reshape()

Most shape changing operators keep data.

Memory: Sharing vs Copying

How to test it?

- create a tensor
- copy/clone/view it
- modify an element
- compare the elements

```
>>> a = np.arange(6)                # [0,1,2,3,4,5]
>>> t = torch.from_numpy(a)
>>> t[2] = 11
>>> t
tensor([ 0,  1, 11,  3,  4,  5])
>>> a
array([ 0,  1, 11,  3,  4,  5])  # Changed the underlying numpy array too!
>>> b = a.copy()
>>> p = t.clone()
>>> t[0] = 7                        # a,t change, b, p remain intact.
```

Creating instances of torch.Tensor without data

```
>>> torch.eye(2)
tensor([[1., 0.],
        [0., 1.]])
>>> torch.zeros(2,2)
tensor([[0., 0.],
        [0., 0.]])
>>> torch.ones(2,2)
tensor([[1., 1.],
        [1., 1.]])

>>> torch.rand(2,2)
tensor([[0.6849, 0.1091],
        [0.4953, 0.8975]])

>>> torch.empty(2,2) # NEVER USE IT! Creates uninitialized tensor.
tensor([[-2.2112e-16,  3.0693e-41],
        [-3.0981e-16,  3.0693e-41]])

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

Interacting with numpy

```
>>> import imageio
>>> img = imageio.imread('example.png') #  reading data from disk
>>> t = torch.from_numpy(a)             #  input from numpy array
>>> out = model(t)                      #  processing
>>> result = out.numpy()                #  converting back to numpy

# tuples, lists, arrays, etc. can be converted automatically:
>>> t2 = torch.tensor(...)
```

Remarks:

- arrays / tensors must be on the same device.
- only detached arrays can be converted to numpy (see later)
- if data types are not the same, casting might be needed (v1.1 or older)
  E.g. adding an integer and a float tensor together.

Torch.tensor functionality

- Common tensor operations:
  - reshape
  - max/min
  - shape/size
  - etc
- Arithmetic operations
  - Abs / round / sqrt / pow /etc
- torch.tensor’s support broadcasting
- In-place operations

Torch.tensor summary

- Very similar to numpy (indexing, main functions)
- Every tensor has a device, a type, and a required_grad attribute
  Conversion and/or device transfer might be needed.
- In-place operations end in underscore, e.g. .fill_()
- Some operations create new tensors, some share data.
- Careful with the broadcasting semantics.

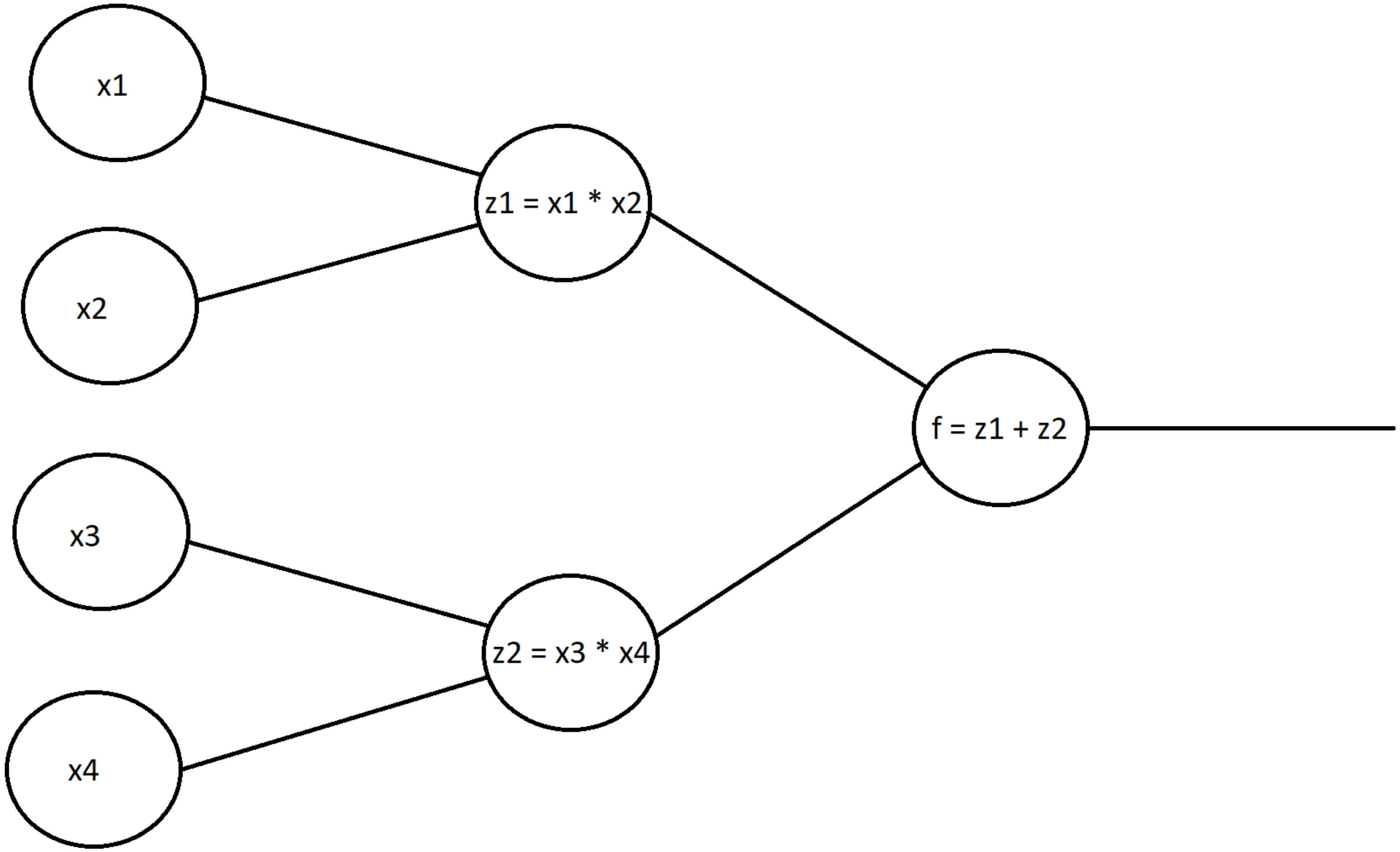
Remark: not just tensors are similar to ndarrays,
but torch functions are also similar to numpy functions.

Progress

- Deep learning frameworks
- PyTorch
  - torch.tensor
  - **Computational graph** (reminder from last week)
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

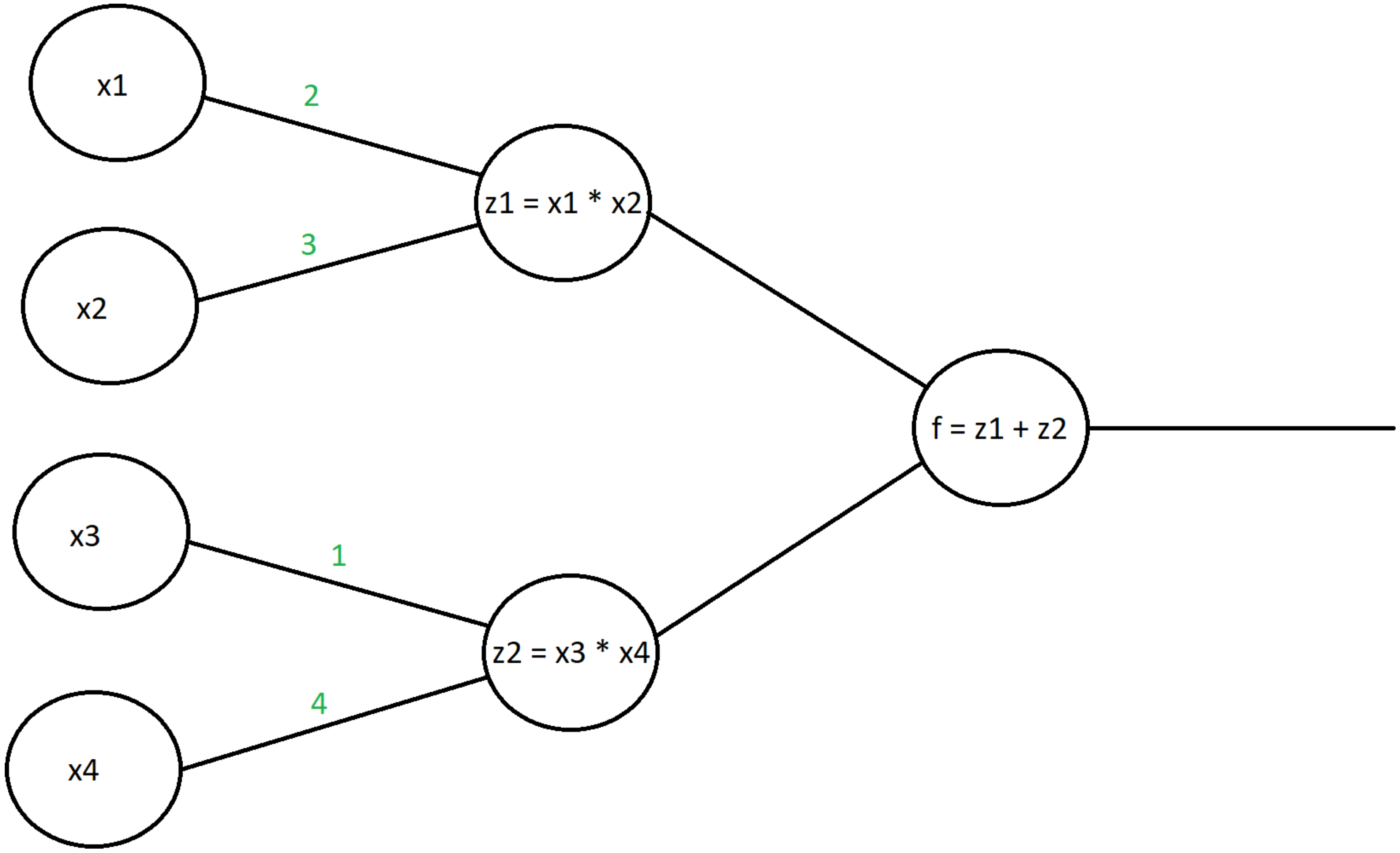
What is a computational graph?

$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$ $\qquad f(\vec{x}) = z_1 + z_2$



Forward propagation

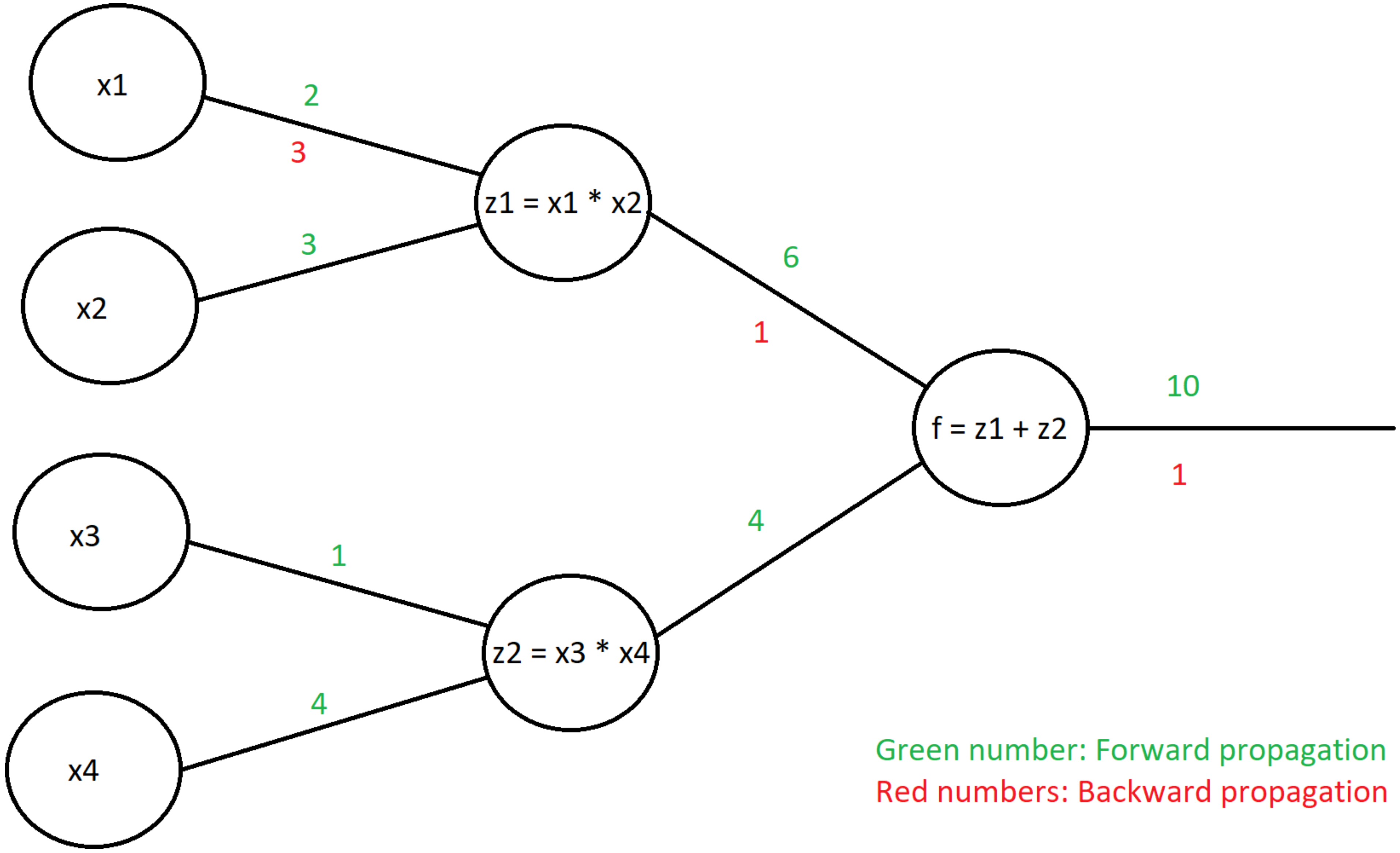
$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$ $\quad f(\vec{x}) = z_1 + z_2$



Backward propagation

What if we want to get the derivative of *f* with respect to the *x1*?

$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$ $\quad f(\vec{x}) = z_1 + z_2$ $\quad \frac{\partial f(\vec{x})}{\partial x_1} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x_1} = x_2$

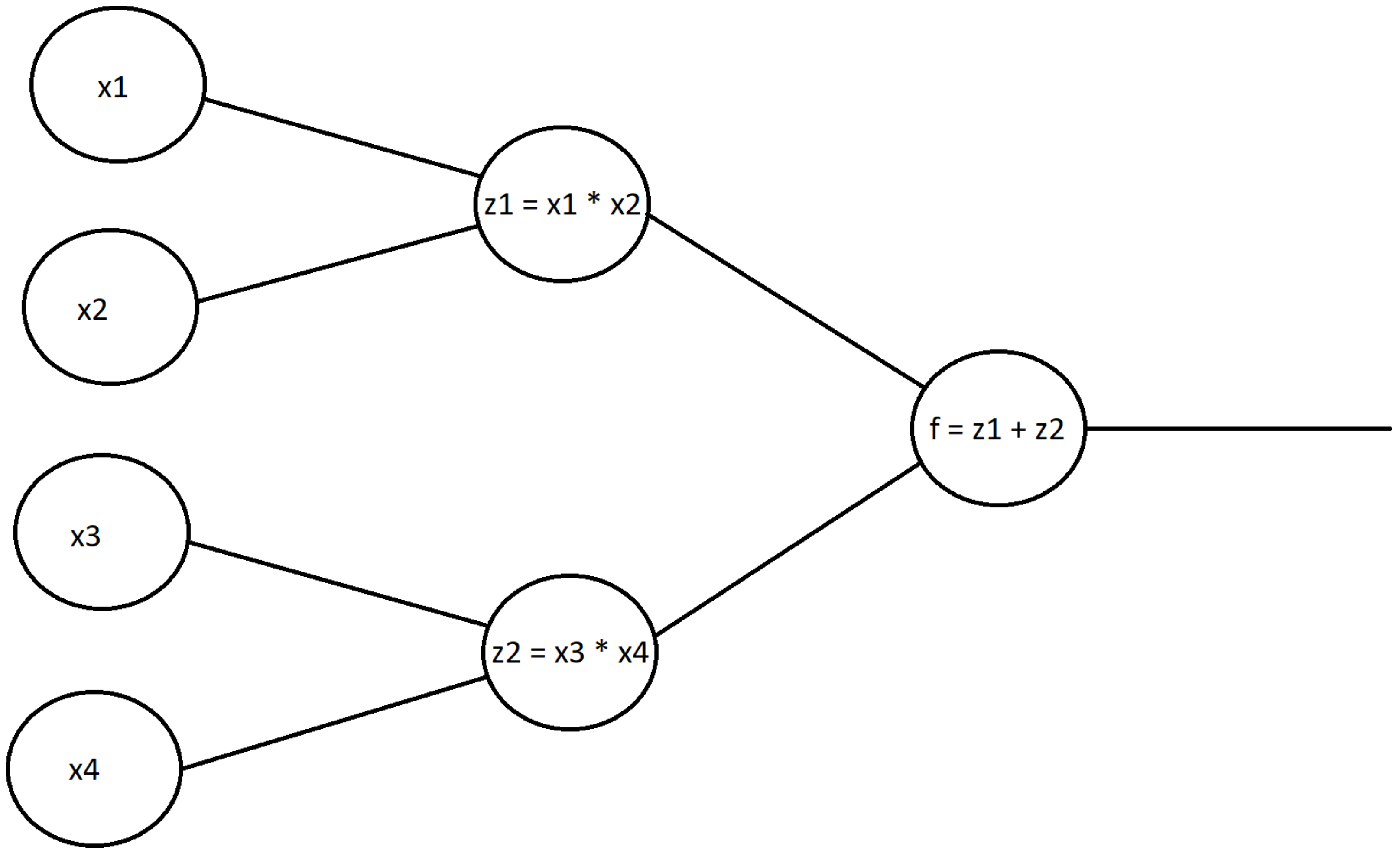

x1
2
3
x2
3
z1 = x1 * x2
6
1
x3
1
z2 = x3 * x4
4
x4
4
f = z1 + z2
10
1
Green number: Forward propagation
Red numbers: Backward propagation

Progress

- Deep learning frameworks
- PyTorch
  - torch.tensor
  - Computational graph
  - **Automatic differentiation (torch.autograd)**
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

Autograd

- Autograd - Automatic differentiation for all operations on Tensors
  - Static computational graph (TensorFlow 1.0)
  - Dynamic computational graph (PyTorch)
- The backward graph is defined by the forward run!

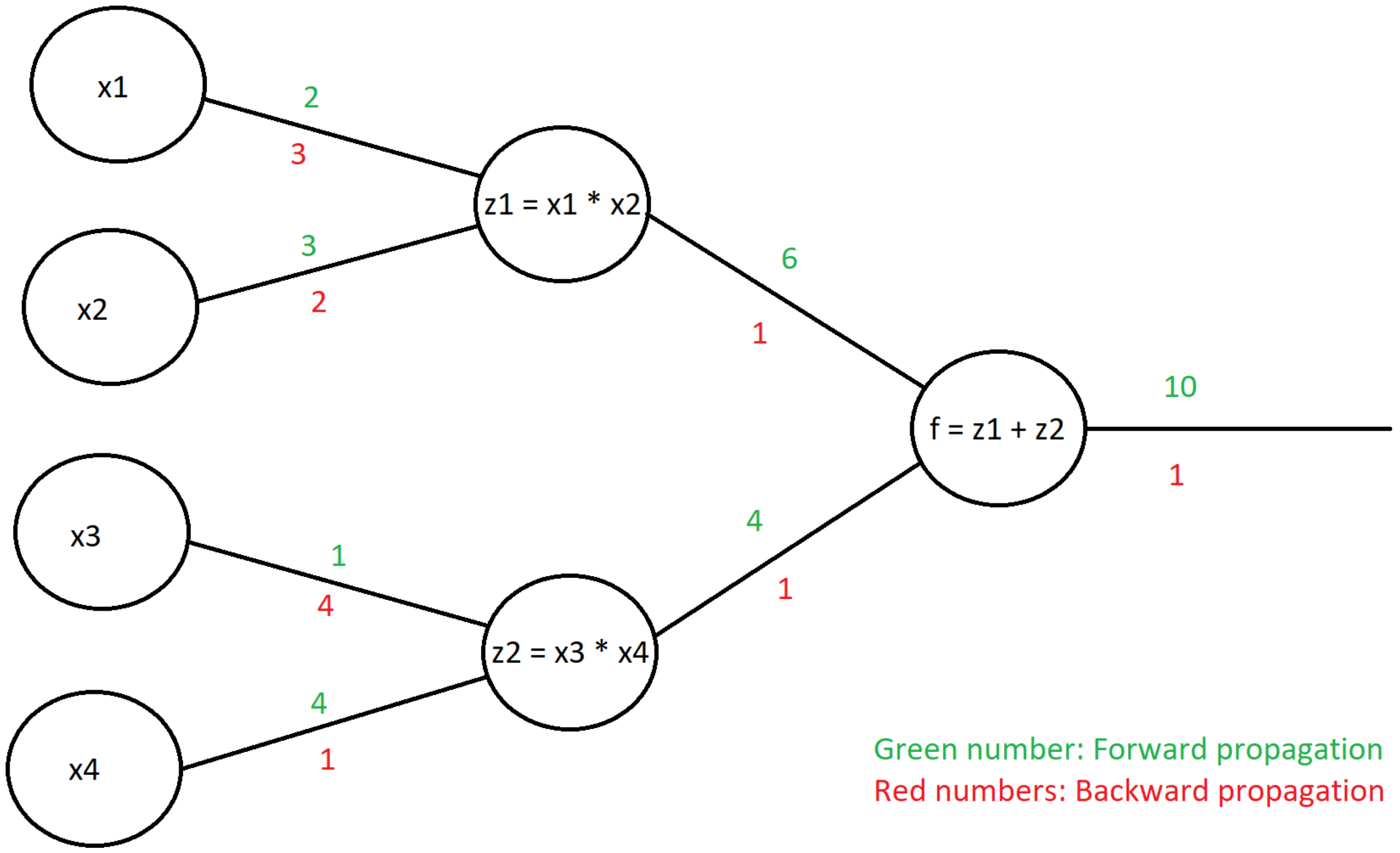


Example 1 (autograd)

```
>>> import torch
>>> from torch import autograd
>>> x1 = torch.tensor(2, requires_grad=True, dtype=torch.float32)
>>> x2 = torch.tensor(3, requires_grad=True, dtype=torch.float32)
>>> x3 = torch.tensor(1, requires_grad=True, dtype=torch.float32)
>>> x4 = torch.tensor(4, requires_grad=True, dtype=torch.float32)
>>> # Forward propagation
>>> z1 = x1 * x2
>>> z2 = x3 * x4
>>> f = z1 + z2
>>> df_dx = grad(outputs=f, inputs = [x1, x2, x3, x4])
>>> df_dx
(tensor(3.), tensor(2.), tensor(4.), tensor(1.))
```

Example 1 (autograd)

```
>>> df_dx = grad(outputs=f, inputs = [x1, x2, x3, x4])
>>> df_dx
(tensor(3.), tensor(2.), tensor(4.), tensor(1.))
```


x1
2
3
x2
3
2
z1 = x1 * x2
6
1
x3
1
4
x4
4
1
z2 = x3 * x4
4
1
f = z1 + z2
10
1
Green number: Forward propagation
Red numbers: Backward propagation

Leaf tensor

A «leaf tensor» is a tensor you created directly, not as the result of an operation.

```
>>> x = torch.tensor(2)  # x is a leaf tensor
>>> y = x + 1            # y is not a leaf tensor
```

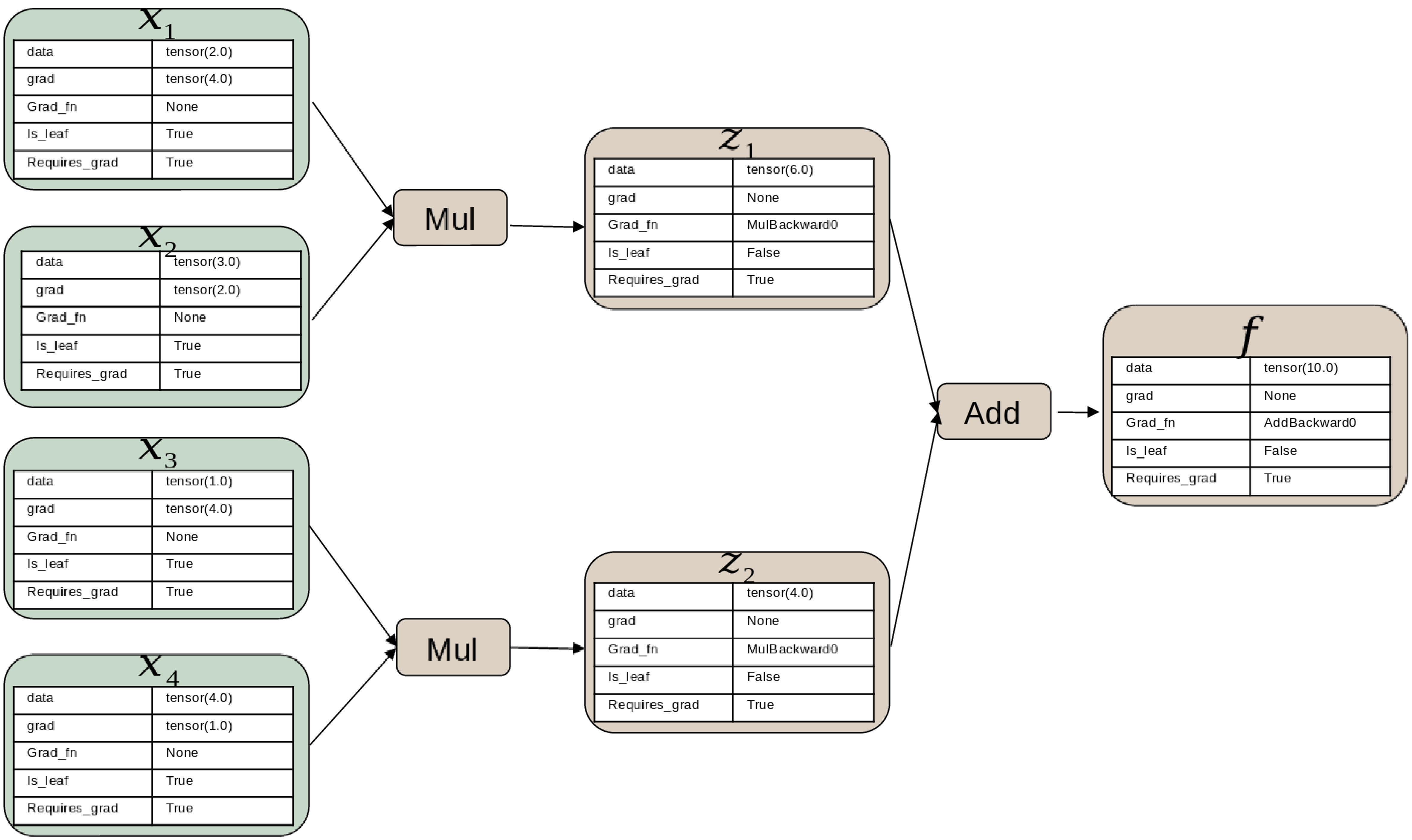
Remember the computation graph:
x1, x2, x3, x4 are the leaf tensors.

Autograd

The need for specifying all tensors is inconvenient.

```
>>> import torch
>>> from torch import autograd
>>> x1 = torch.tensor(2, requires_grad=True, dtype=torch.float32)
>>> x2 = torch.tensor(3, requires_grad=True, dtype=torch.float32)
>>> x3 = torch.tensor(1, requires_grad=True, dtype=torch.float32)
>>> x4 = torch.tensor(4, requires_grad=True, dtype=torch.float32)
>>> # Forward propagation
>>> z1 = x1 * x2
>>> z2 = x3 * x4
>>> f = z1 + z2
>>> #  df_dx = grad(outputs=f, inputs = [x1, x2, x3, x4]) # inconvenient
>>> f.backward()                                         # that is better!
>>> print(f" f's derivative w.r.t. x1 is {x.grad}")
tensor(3.)
```

Chain rule is applied back to all the leaf tensors with *requires_grad=True* attribute.



x_1

data	tensor(2.0)
grad	tensor(4.0)
Grad_fn	None
Is_leaf	True
Requires_grad	True

x_2

data	tensor(3.0)
grad	tensor(2.0)
Grad_fn	None
Is_leaf	True
Requires_grad	True

x_3

data	tensor(1.0)
grad	tensor(4.0)
Grad_fn	None
Is_leaf	True
Requires_grad	True

x_4

data	tensor(4.0)
grad	tensor(1.0)
Grad_fn	None
Is_leaf	True
Requires_grad	True

z_1

data	tensor(6.0)
grad	None
Grad_fn	MulBackward0
Is_leaf	False
Requires_grad	True

z_2

data	tensor(4.0)
grad	None
Grad_fn	MulBackward0
Is_leaf	False
Requires_grad	True

f

data	tensor(10.0)
grad	None
Grad_fn	AddBackward0
Is_leaf	False
Requires_grad	True

Context managers, decorators

- We can locally disable/enable gradient calculation with
  - torch.no_grad()
  - torch.enable_grad()
- or using the @torch.no_grad @torch.enable_grad decorators

```
>>> x = torch.tensor([1], requires_grad=True)
>>> with torch.no_grad():
...   y = x * 2
>>> y.requires_grad
False

>>> with torch.no_grad():
...   with torch.enable_grad():
...     y = x * 2
>>> y.requires_grad
True
```

Note: Use «torch.no_grad()» during inference

Autograd in depth (optional)

https://www.youtube.com/watch?v=MswxJw-8PvE

https://pytorch.org/docs/stable/autograd.html

Example 2 - Solving a linear problem

Generating data:

```
>>> a_ref = -1.5
>>> b_ref = 8
>>> noise = 0.2 * np.random.randn(50)
>>> x = np.linspace(1, 4, 50)
>>> y = a_ref * x + b_ref + noise
```

Defining loss function:

```
>>> def MSE_loss(prediction, target):
...     return (prediction-target).pow(2).mean()
```

Example 2 - Solving a linear problem

Data as torch tensors and the unknown variables:

```
xx = torch.tensor(x, dtype = torch.float32)
yy = torch.tensor(y, dtype = torch.float32)

a = torch.tensors(0, requires_grad = True, dtype=torch.float32)
b = torch.tensors(5, requires_grad = True, dtype=torch.float32)
```

Example 2 - Solving a linear problem

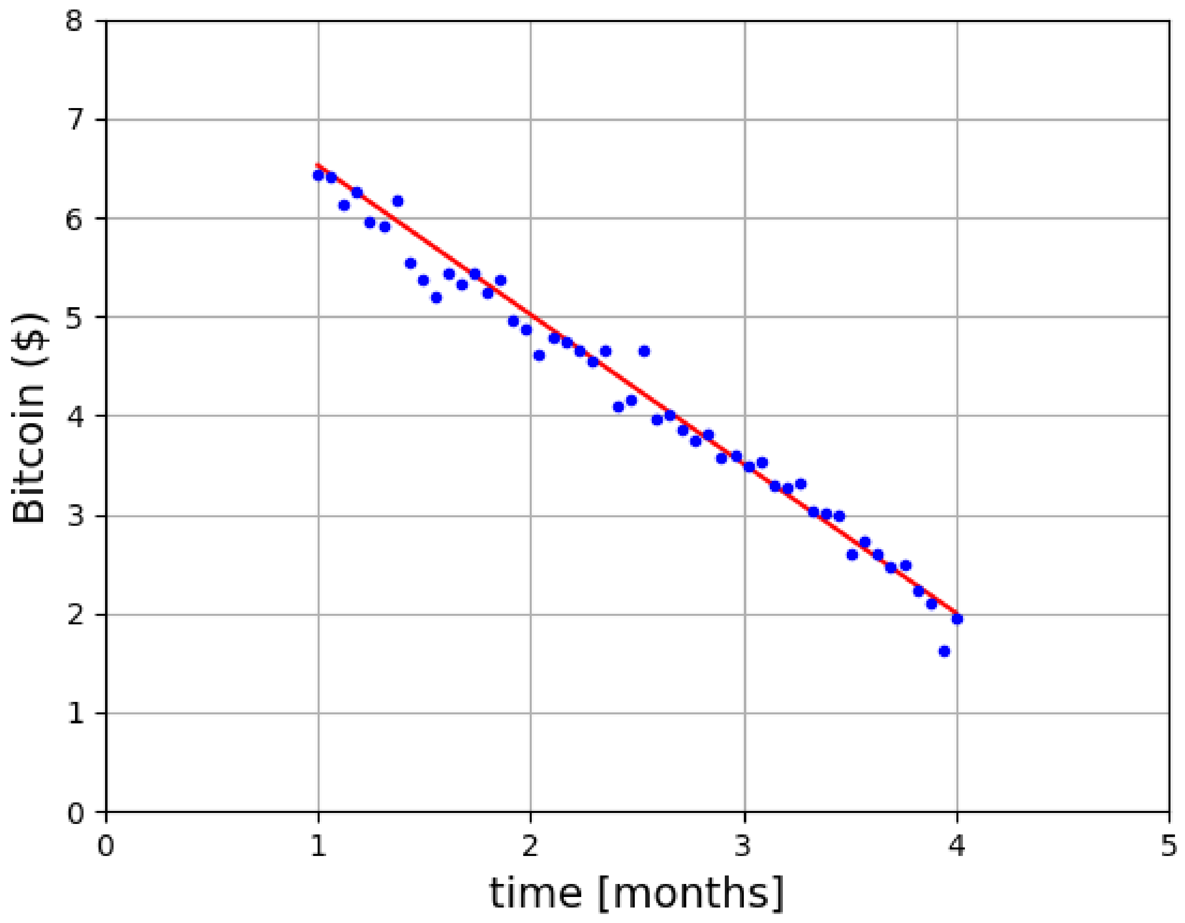
Training loop:

```python
number_of_epochs = 1000
learning_rate = 0.01
for iteration in range(number_of_epochs):
    y_pred = a * xx + b
    loss = MSE_loss(y_pred, yy)
    loss.backward()
    with torch.no_grad():
        a = a - learning_rate * a.grad
        b = b - learning_rate * b.grad
    a.requires_grad = True
    b.requires_grad = True
print(a)
print(b)
```

Example 2 - Solving a linear problem

Result:

```
tensor(-1.5061, requires_grad=True)
tensor(8.0354, requires_grad=True)
```



Other useful torch.tensor functions

If you want to detach a tensor from the graph, you can use « **detach()** »
If you want to get a python number from a tensor, you can use « **item()** »
But if you just take an element, it still will be part of the computational graph!

```
>>> x=torch.tensor([2.5,3.5], requires_grad=True)
tensor([2.5000, 3.5000], requires_grad=True)
>>> x.detach()
tensor([2.5000, 3.5000])
>>> x[0]     # still part of the graph!
tensor(2.5000, grad_fn=<SelectBackward>)
>>> x[0].item()
2.5

>>> #  a frequent line when you go back to numpy:
>>> x.detach().cpu().numpy()
array([2.5, 3.5], dtype=float32)
```

Remember our example workflow

- creating dataset
- creating a neural network (model)
- defining a loss function
- loading samples (data loader)
- predicting with the model
- comparison of the prediction and the target (loss)
- backpropagation: calculating gradients from the error
- updating the model (optimizer)
- checking the loss: if low enough, stop training

Progress

- Deep learning frameworks
- PyTorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - **Data loading and preprocessing (torch.utils)**
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

Data loading and preprocessing

- The «torch.utils.data» package have two useful classes for loading and preprocessing data:
  - torch.utils.data.Dataset
  - torch.utils.data.DataLoader
- For more information visit:
- https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

torch.utils.data.Dataset

Typical structure of the Dataset class

```
import torch
class ExampleDataset(torch.utils.data.Dataset):
    def __init__(self,params, *args,**kwargs):
        super().__init__(*args,**kwargs)
        # do initalization based on the params,
        # e.g. load images, etc.
        self.data = ...

    def __getitem__(self, idx):
        # return sample indexed with 'idx'
        # must be tensor or dictionary of tensors!
        self.data[idx]

    def __len__(self):
        # return the number of samples
        return self.data.shape[0]
```

torch.utils.data.Dataset: regression

```
import torch
class LinearRegressionDataset(torch.utils.data.Dataset):

    def __init__(self,N = 50, m = -3, b = 2, *args,**kwargs):
        # N: number of samples, e.g. 50
        # m: slope
        # b: offset
        super().__init__(*args,**kwargs)

        self.x = torch.rand(N)
        self.noise = torch.rand(N)*0.2
        self.m = m
        self.b = b

    def __getitem__(self, idx):
        y = self.x[idx] * self.m + self.b + self.noise[idx]
        return {'input': self.x[idx], 'target': y}

    def __len__(self):
        return len(self.x)
```

torch.utils.data.Dataset: images

```
import torch
import imageio
class ImageDataset(torch.utils.data.Dataset):
    def __init__(self, root, N, *args,**kwargs):
        super().__init__(*args,**kwargs)

        self.input, self.target = [], []
        for i in range(N):
            t = imageio.imread(f'{root}/train_{i}.png')
            t = torch.from_numpy(t).permute(2,0,1)
            l = imageio.imread(f'target_{i}.png')
            l = torch.from_numpy(l).permute(2,0,1)
            self.input.append(t)
            self.target.append(l)

    def __getitem__(self, idx):
        return {'input': self.input[idx], 'target': self.target[idx]}

    def __len__(self):
        return len(self.input)
```

torch.utils.data.Dataset

```python
import torch
import ImageDataset

datapath = 'data_directory'
myImageDataset = ImageDataset(dataPath, 50)
# iterating through the samples
for sample in myImageDataset:
    input = sample['input'].cpu()  # or .cuda()
    target = sample['target'].cpu()  # or .to(device)
    ....
```

Never ever use .cuda() in the dataset or data loaders!

torch.utils.data.DataLoader

```
import torch
import ImageDataset
datapath = 'data_directory'
myImageDataset = ImageDataset(dataPath, 50)
# iterating through the samples
train_loader = DataLoader(dataset=myImageDataset, batch_size=32,
                          shuffle=False, num_workers=2)
for sample in train_loader:
    ...
```

- «DataLoader» is used to:
  - Batching the dataset
  - Shuffling the dataset
  - Utilizing multiple CPU cores/ threads

Data augmentetion

(forward reference, not for today)

- modifying the dataset for better training
  (more robust, etc.)
- data set can have a a *transform* parameter

Details here:
https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

Data augmentetion

(forward reference, not for today)

```python
import torch
import imageio
class ImageDataset(torch.utils.data.Dataset):
    def __init__(self, root, N, transform = None, *args,**kwargs):
        super().__init__(*args,**kwargs)
        self.transform = transform
        ...
    def __getitem__(self, idx):
        sample = {'input': self.input[idx], 'target': self.target[idx]}
        if self.transform:
            sample = self.transform(sample)
        return sample

    def __len__(self):
        return len(self.input)
```

Data augmentetion

(forward reference, not for today)

```
class ImageDataset(torch.utils.data.Dataset):
    def __init__(self, root, N, transform = None, *args,**kwargs):
        super().__init__(*args,**kwargs)
        self.transform = transform
        ...
    def __getitem__(self, idx):
        sample = {'input': self.input[idx], 'target': self.target[idx]}
        if self.transform:
            sample = self.transform(sample)
        return sample

    def __len__(self):
        return len(self.input)
```

Data transformations

(forward reference, not for today)

```
import torchvision.transforms as T
composed = transforms.Compose([T.Rescale(256),
                               T.RandomCrop(224),
                               T.ToTensor()]
                               )
...
dataset = Mydataset(..., transform = composed)

# another version, needs different dataset
dataset = Mydataset(..., transform = {'input'  : composed,
                                      'target' : None} )
```

Progress

- Deep learning frameworks
- PyTorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - **Useful functions (torch.nn.functional)**
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

torch.nn.functional

- The «torch.nn.functional» package is the functional interface for Pytorch features.
- Most feature exist both as a function and as a class.
- Structural parts, or objects with internal state usually used as objects
- Stateless or simple expressions are usually used in functional form.
- Activation functions, losses, convolutions, etc. It is a huge module.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
x = torch.rand(2,2)
y = F.relu(x)
relu = nn.ReLU()  # creating the object first
z = relu(x)       # then using it
y == z            # they should be the same

# Similarly:
mseloss = nn.MSELoss()

F.mseloss(...) == mseloss(...)
```

Progress

- Deep learning frameworks
- PyTorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - **Creating the model (torch.nn)**
  - Optimizers (torch.optim)
  - Save/load models
- Miscellaneous

Creating the model

A model is of a nn.Module class type. A model can contain other models. E.g. we can create the class “Model” based on the stacking nn.Modules of type nn.Linear()

The nn.Module’s weights as called “Parameters”, and are similar to tensors with “requires_grad=True”.

A nn.Module consists of an initialization of the Parameters and a forward function.

```python
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        # structure definition and initialization

    def forward(self, x):
        # actual forward propagation
        result = processing(x)
        return result
```

Creating the model

```
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        # let's assume 28x28 input images, e.g. MNIST characters
        self.fc1 = nn.Linear(in_features = 28 * 28, out_features = 128, bias=True)
        self.fc2 = nn.Linear(in_features = 128, out_features = 64, bias=True)
        self.fc3 = nn.Linear(in_features = 64, out_features = 10, bias=True)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Creating the model, alternative way

```
class Model2(nn.Module):
    def __init__(self):
        super().__init__()
        # let's assume 28x28 input images, e.g. MNIST characters
        self.fc1 = nn.Linear(in_features = 28 * 28, out_features = 128, bias=True)
        self.activation1 = nn.ReLU()
        self.fc2 = nn.Linear(in_features = 128, out_features = 64, bias=True)
        self.activation2 = nn.ReLU()
        self.fc3 = nn.Linear(in_features = 64, out_features = 10, bias=True)
        self.activation3 = nn.ReLU()

    def forward(self, x):
        x = self.activation1(self.fc1(x))
        x = self.activation2(self.fc2(x))
        x = self.activation3(self.fc3(x))
        return x
```

What is the difference?

nn.Module’s member functions

Access information of a model:

```
>>> model = Model()
>>> model.eval() # see below
>>> list(model.children())
[Linear(in_features=784, out_features=128, bias=True),
Linear(in_features=128, out_features=64, bias=True),
Linear(in_features=64, out_features=10, bias=True)]
```

- Children: the parameters and modules / layers defined in the constructor.
- Parts defined in the *forward* method will not be listed.
- Forward is called many times, expensive objects should not be recreated.

Some layers as e.g. "dropout" and "batch_norm" should operate differently during training and evaluation of the model. We can set the model in different state by the *.train()* and *.eval()* functions.

Model parameters

```
>>> for key, value in model.state_dict().items():
...     print(f'layer = {key:10s} | feature shape = {value.shape}')

layer = fc1.weight | feature shape = torch.Size([128, 784])
layer = fc1.bias   | feature shape = torch.Size([128])
layer = fc2.weight | feature shape = torch.Size([64, 128])
layer = fc2.bias   | feature shape = torch.Size([64])
layer = fc3.weight | feature shape = torch.Size([10, 64])
layer = fc3.bias   | feature shape = torch.Size([10])
```

The .state_dict() contains all the trainable parameters of the model,
this is used for optimization and saving/restoring the model. (See later.)

Advanced examples

(Not part of the mandatory curriculum for today)

Naive implementation of ReLU:

```
class ReLU(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        x = x.clone()
        x[x<0] = 0    # implements max(x,0)
        return x
```

You can implement any activation function, any transformation, and autograd tracks everything.

Advanced examples, part 2

(Not part of the mandatory curriculum for today)

Skip connections and residual connections:

```python
class SkipResBlock(nn.Module):
    def __init__(self):
        super().__init__()
        self.convolution = nn.ConvX(...)
        self.convolution2 = nn.ConvX(...)

    def forward(self, x):
        y = self.convolution(x) + x # residual block
        y = F.relu(y)
        z = torch.cat(y,x, dim=...) # skip connection
        z = self.convolution2(z)
        return F.relu(z)
```

Careful with the dimensions.

Current state

The workflow looks like this so far:

```python
device = torch.device('cpu')
dataset = CustomDataset()
dataloader = DataLoader(dataset, ...)
model = MyModel()
model.to(device)
for i in range(epochs):
    training_loss = 0
    for sample in dataloader:
        input = sample['input'].to(device)
        target = sample['target'].to(device)
        prediction = model(input)
        loss = loss_function(prediction, target)
        training_loss += loss.item()
        loss.backward()
        # updating the model
    print(f'Current training loss: {training_loss}')
    # validation loop
    ...
# saving the model
```

Progress

- Deep learning frameworks
- PyTorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - **Optimizers (torch.optim)**
  - Save/load models
- Miscellaneous

Defining an optimizer

Using PyTorch’s optimizers is easy!

```
import torch
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)
...
for sample in dataloader:
    input = sample['input'].to(device)
    target = sample['target'].to(device)
    prediction = model(input)
    loss = loss_fn(prediction, target)

    optimizer.zero_grad() # clears the gradients
    loss.backward()
    optimizer.step()      # performs the optimization
```

Accumulating gradients

If we don't clear the gradients, they sum up.
This is often source of bugs, but
this can be exploited for larger effective batch sizes:

```
import torch
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)

optimizer.zero_grad()
for idx, sample in enumerate(dataloader):
    input = sample['input'].to(device)
    target = sample['target'].to(device)

    prediction = model(input)
    loss = loss_fn(prediction, target)

    loss.backward()
    if idx % 10 = 9:
        optimizer.step()
        optimizer.zero_grad()
```

Progress

- Deep learning frameworks
- PyTorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - **Save/load models**
- Miscellaneous

Save/load models

Saving and loading can easily be don using “torch.save” and “torch.load”
PyTorch uses “pickling” to serialize the data.

```
>>> state = {'model_state' : model.state_dict(),
             'optimizer': optimizer.state_dict)}
>>> torch.save(state, 'state.pt')
```

Restoring state:

```
>>> model = Model()
>>> optimizer = optim.SGD(model_parameters(), lr=0.01)
>>> checkpoint = torch.load('state.pt')
>>> model.load_state_dict(checkpoint['model_state'])
>>> optimizer.load_state_dict(checkpoint['optimizer_state'])
```

All the pieces together, part 1

```python
import json
config = json.load(open('config.cfg'))
device = torch.device(config['device'])
training_data = CustomDataset(..., **config['train'])
validation_data = CustomDataset(..., **config['valid'])
train_loader = DataLoader(training_data, **config['loader'])
validation_loader = DataLoader(validation_data, **config['loader'])
model = MyModel(**config['model'])
model.to(device)
optimizer = Optimizer(model.parameters(), **config['optimizer'])
for i in range(config['epochs']):
    model.train()
    for sample in train_loader:
        optimizer.zero_grad()
        input, target = sample['input'].to(device), sample['target'].to(device)
        prediction = model(input)
        loss = loss_function(prediction, target)
        print(f'Current training loss: {loss.item()}')
        loss.backward()
        optimizer.step()
```

All the pieces together, part 2

```
    # validation loop
    model.eval()
    validation_loss = 0
    for sample in validation_loader:
        input, target = sample['input'].to(device), sample['target'].to(device)
        prediction = model(input)
        loss = loss_function(prediction, target)
        validation_loss += loss.item()
    print(f'Current validation loss: {validation_loss}')
    if validation_loss < config['loss_threshold']: # or other condition
        break
full_state = {'model_state' : model.state_dict(), 'optimizer': optimizer.state_dict()}
torch.save(full_state, 'parameters.pt')
```

Reproducibility

Sometimes it is hard to reproduce bugs because of the randomness
in the training. The solution is using fixed random seeds.
For debugging purposes, you should start your codes with these lines:

```
import numpy as np
np.random.seed(42)                                  # your favourite integer

import torch
torch.manual_seed(42)                             # your favourite integer
torch.backends.cudnn.deterministic = True    # disable optimizations
torch.backends.cudnn.benchmark = False
```

But remove them when you are done with debugging,
otherwise all the models will be the same!
See: https://pytorch.org/docs/stable/notes/randomness.html

Progress

- Deep learning frameworks
- PyTorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - Useful functions (torch.nn.functional)
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- **Miscellaneous**

Pretrained models

(see later in the semester)

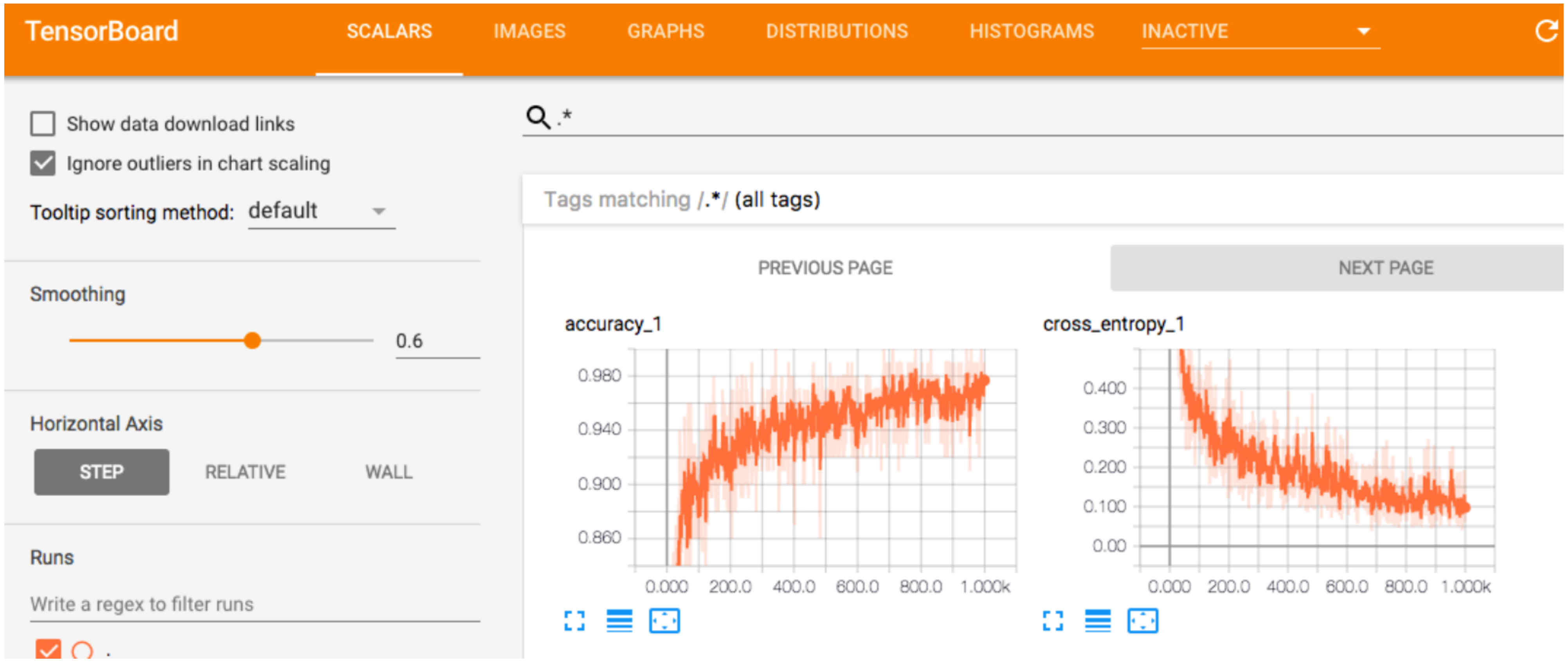
- There are several open model repositories with pretrained weights
- They can be used for transfer learning
- Or could be used as a building block for your own model, etc.

```
from torchvision.models.resnet import vgg16
model = vgg16(pretrained=True)
```

See https://pytorch.org/docs/stable/torchvision/models.html

Visualization

Tensorboard:



Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.6
Horizontal Axis
Runs
.*
Tags matching /.*/ (all tags)
accuracy_1
cross_entropy_1

See: https://pytorch.org/tutorials/intermediate/tensorboard_tutorial.html
Alternatives: MLFlow (mlflow.org), Weights and Biases (wandb.com), etc.

Building your own deep learning rig

- For the course: no need to invest
- For fun: there is no upper limit

https://favouriteblog.com/best-gpu-for-deep-learning/

Summary

- *Deep learning frameworks*
- PyTorch
  - torch.tensor
  - Computational graph
  - Automatic differentiation (torch.autograd)
  - Data loading and preprocessing (torch.utils)
  - *Useful functions (torch.nn.functional)*
  - Creating the model (torch.nn)
  - Optimizers (torch.optim)
  - Save/load models
- *Miscellaneous*

Check out the tutorials / examples !