Introduction to Deep Learning with NumPy and PyTorch 2/4

Part 2 - Automatic Differentiation and PyTorch - “a visit into the engine room”

1st Terascale Alliance Machine Learning School, DESY, Hamburg

Dirk Krücker
DESY - Hamburg, 23.10.2018

Content II

- Technicalities
  - BLAS etc.
  - What is PyTorch
  - Automatic differentiation
  - PyTorch as NumPy replacement
  - Cuda semantics
  - Autograd mechanics
- Tutorial IIa
  - PyTorch tensor basics
  - PyTorch Timing
  - Autograd example
- Tutorials IIb
  - A toy model I
  - A first Neural Network from NumPy to PyTorch

Needs for Deep Learning

Efficient Matrix Manipulation, Automatic differentiation and High Level Libraries

As we have seen Deep Learning means

- Efficient matrix, or higher rank tensor, manipulation
  - NumPy is the most common tool for scientific, numerical calculation (at least outside HEP)
  - Efficient low-level linear algebra system for matrix operation that allow to change between different hardware
    - BLAS: openBLAS or MKL and cuBLAS
    - **LAPACK** defines higher level operation linear systems, decompositions, eigenvalues etc.
      - systematic (cryptic) naming scheme that appears also in PyTorch
- Efficient differentiation for backpropagation
  - Different approaches, see later
- Productive interface to define models
  - Python

Needs for Deep Learning

Efficient Matrix Manipulation, Automatic differentiation and High Level Libraries

As we have seen Deep Learning means

- Efficient matrix, or higher rank tensor, manipulation
  - NumPy is the most common tool for scientific, numerical calculation (at least outside HEP)
  - Efficient low-level linear algebra system for matrix operation that allow to change between different hardware
    - BLAS: openBLAS or MKL and cuBLAS
    - **LAPACK** defines higher level operation linear systems, decompositions, eigenvalues etc.
      - systematic (cryptic) naming scheme that appears also in PyTorch
- Efficient differentiation for backpropagation
  - Different approaches, see later
- Productive interface to define models
  - Python

What is BLAS?

- **Basic Linear Algebra Subprograms** (BLAS)
  - a **set of low-level routines** for common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication
- Defines an interface to make libraries interchangeable
- BLAS implementations take advantage of special floating point hardware e.g. **SIMD** instructions.
  - SIMD (**Single instruction, multiple data**) vector register
  - All modern **CPUs** come with these abilities
    - SSE, AVX, AVX512, ...
  - And **GPUs**
    - Highly parallel - Latest NVidia 2080 Ti (Turing) 4352 cores
  - Rarely used directly in programming but by optimized libraries
    - openBLAS or **MKL** (Intel) or Apple's Accelerate framework
      - Multithreading!
    - **cuBLAS** (NVIDIA)

Needs for Deep Learning

Efficient Matrix Manipulation, Automatic differentiation and High Level Libraries

As we have seen Deep Learning means

- Efficient matrix, or higher rank tensor, manipulation
  - NumPy is the most common tool for scientific, numerical calculation (at least outside HEP)
  - Efficient low-level linear algebra system for matrix operation that allow to change between different hardware
    - BLAS: openBLAS or MKL and cuBLAS
    - **LAPACK** defines higher level operation linear systems, decompositions, eigenvalues etc.
      - systematic (cryptic) naming scheme that appears also in PyTorch
- Efficient differentiation for backpropagation
  - Different approaches, see later
- Productive interface to define models
  - Python

Something like NumPy with automatic differentiation and GPU support

PyTorch

Facebook's artificial intelligence research (open source) python library

- **PyTorch** is an open source machine learning library for Python
  - Released in 2016
    - We use the latest release 0.41 next will be 1.0
  - Written in Python, C++ and CUDA
    - CUDA is the NVidia language to program the graphics processing unit (GPU) for fast numerical

Main features

- Tensor computation (like NumPy) with GPU acceleration
- Deep Neural Networks built on a tape-based auto-differentiation

Modules

- **Autograd** - automatic diff
- **Optim** - optimization
- **nn** - neural networks

... we will cover all this later

PyTorch

Facebook's artificial intelligence research (open source) python library

- Release **PyTorch 0.41** is used here
  - next will be 1.0 - had been promised for summer 2018 ...
- Web: https://pytorch.org
- Tutorials: https://pytorch.org/tutorials
- Documentation: https://pytorch.org/docs/stable/index.html
- Advanced examples https://github.com/pytorch/examples

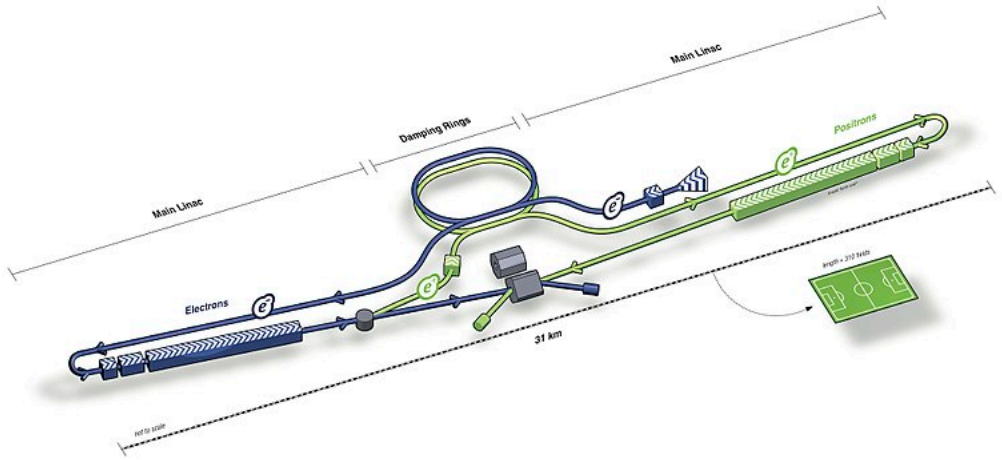
Why PyTorch?

Flexibility and ease to debug

- The biggest difference between PyTorch and TensorFlow is that TensorFlow's computational graphs are static and PyTorch uses dynamic computational graphs
  - If we do our job, you will understand this at the end of this day(week)
- The 1.0 is not yet out but most of the things covered here will not differ

Automatic Differentiation

I worked on accelerator physics and ILC simulation a while ago



- I learned about automatic differentiation when I worked for the ILC.
- But the story starts even earlier SSC (the abandoned Superconducting Super Collider)
- "High-Order Description of Accelerators using Differential Algebra and First Applications to the SSC" SSC-N-571 (1988)

Automatic Differentiation

- is not **Symbolic Differentiation** (e.g. Mathematica or pen & paper)
- is not **Numerical Differentiation**

$$\lim_{\Delta x \to 0} \frac{f(x + \Delta x)}{\Delta x} \approx f'(x)$$

- It is about calculating exact derivatives numerically
- This is extremely useful for artificial neural networks

Derivatives as Algebra

Some interlude for the math nerds

- We are used to think about derivatives as functional mappings: **F->F'**
- It is possible to do the derivative "pointwise", e.g. **F(2) -> F'(2)**
- Let's define an Algebra $\boldsymbol{D}$ aka Vector space of pairs with a special multiplication.
  - We call this differential algebra or dual numbers
- As a proper group there is a **one** and **inverse** and
- we can define a **root** $\sqrt{(a_0, a_1)}$

- *Differential Algebra*
  $(a_0, a_1), (b_0, b_1), \ldots \in D$
- *Multiplication:*
  $(a_0, a_1) * (b_0, b_1) \boxed{:= (\mathbf{a_0 b_0},\ \mathbf{a_0 b_1} + \mathbf{a_1 b_0})}$
- *One element:*
  $$(a_0, a_1) * \mathbf{(1, 0)} = (a_0, a_1)$$
- *Inverse:*
  $$(a_0, a_1)^{-1} = (\frac{\mathbf{1}}{\boldsymbol{a_0}}, -\frac{\boldsymbol{a_1}}{\boldsymbol{a_0^2}})$$
- *Root:*
  $$\sqrt{(a_0, a_1)} = (\sqrt{\boldsymbol{a_0}}, \frac{\boldsymbol{a_1}}{\mathbf{2}\sqrt{\boldsymbol{a_0}}})$$

Derivatives as Algebra

Something for the math nerds

- Let's consider a simple example

  $\boldsymbol{f(x) = \frac{1+\sqrt{x}}{x}}$ with $f'(x) = \frac{1+\frac{1}{2}\sqrt{x}}{x^2}$

  at e.g. x=2 we have:

  $f(2) = \frac{1+\sqrt{2}}{2}$ and $f'(2) = \frac{1+\frac{1}{2}\sqrt{2}}{2}$

- Now we use our algebra to do the calculation at (2,1) applying the root and the inverse

$$f((2,1)) = \frac{1+\sqrt{(2,1)}}{(2,1)} = \frac{1+(\sqrt{2},,\frac{1}{2\sqrt{2}})}{(2,1)}$$

$$= \left((1,0) + \left(\sqrt{2},,\frac{1}{2\sqrt{2}}\right)\right) * \left(\frac{1}{2}, -\frac{1}{4}\right)$$

$$= \left(\frac{1+\sqrt{2}}{2}, \frac{1+\frac{1}{2}\sqrt{2}}{4}\right)$$

$f(2)$ $\qquad$ $f'(2)$

- *Differential Algebra*

  $(a_0, a_1), (b_0, b_1), \ldots \in D$

- *Multiplication:*

  $(a_0, a_1) * (b_0, b_1) := (\mathbf{a_0 b_0}, \mathbf{a_0 b_1 + a_1 b_0})$

- *One element:*

$$(a_0, a_1) * (\mathbf{1}, \mathbf{0}) = (a_0, a_1)$$

- *One element:*

$$(a_0, a_1)^{-1} = (\frac{\mathbf{1}}{\boldsymbol{a_0}}, -\frac{\boldsymbol{a_1}}{\boldsymbol{a_0^2}})$$

- *Root:*

$$\sqrt{(a_0, a_1)} = (\sqrt{\boldsymbol{a_0}}, \frac{\boldsymbol{a_1}}{\boldsymbol{2\sqrt{a_0}}})$$

Derivatives as Algebra

Something for the math nerds

- Let's consider a simple example

  $\boldsymbol{f(x) = \frac{1+\sqrt{x}}{x}}$ with $f'(x) = \frac{1+\frac{1}{2}\sqrt{x}}{x^2}$

  at e.g. x=2 we have:

  $f(2) = \frac{1+\sqrt{2}}{2}$ and $f(2) = \frac{1+\frac{1}{2}\sqrt{2}}{2}$

- Now we use our algebra to do the calculation at (2,1) applying the root and the inverse

$$f((2,1)) = \frac{1+\sqrt{(2,1)}}{(2,1)} = \frac{1+(\sqrt{2},,\frac{1}{2\sqrt{2}})}{(2,1)}$$

$$= \left((1,0) + \left(\sqrt{2},,\frac{1}{2\sqrt{2}}\right)\right) * \left(\frac{1}{2}, -\frac{1}{4}\right)$$

$$= \left(\frac{1+\sqrt{2}}{2}, \frac{1+\frac{1}{2}\sqrt{2}}{4}\right)$$

$f(2)$ $\qquad$ $f'(2)$

- We can calculate exact derivatives for a specific numerical value by following some "recipe" - do it yourself with pen and paper to get the idea of pointwise numerical derivatives
- Without ever calculating $f'(x)$ analytically
- The recipe can easily be implemented by a piece of e.g. C++ that implements a class
- This approach is called automatic differentiation by dual numbers or operator overloading
- Most ML libraries use another way:
- **Reverse accumulation using the computational graph**

Computation Graph

Derivative of a piece of code - The heart of modern deep learning

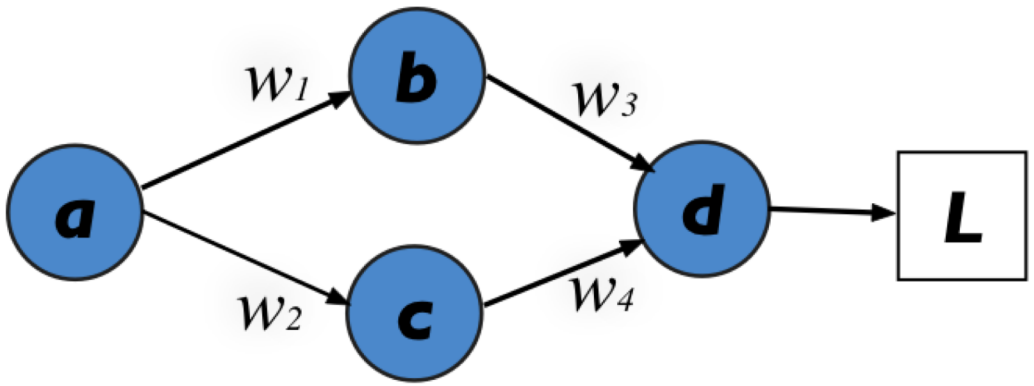
For the training of a neural network with potentially millions of weights, we need to

- Compute gradients of the loss function with respect to every weight and bias term
- Update these weights using gradient descent
- We certain will not be able to do this by hand

- Some **Python code** for a minimalistic network

```
b = w1 * a
c = w2 * a
d = (w3 * b) + (w4 * c)
L = f(d)
```

- The code defines a **computation graph**
- Starting at node ***a*** we work **forward** following the program code that calculates the output



Computation Graph

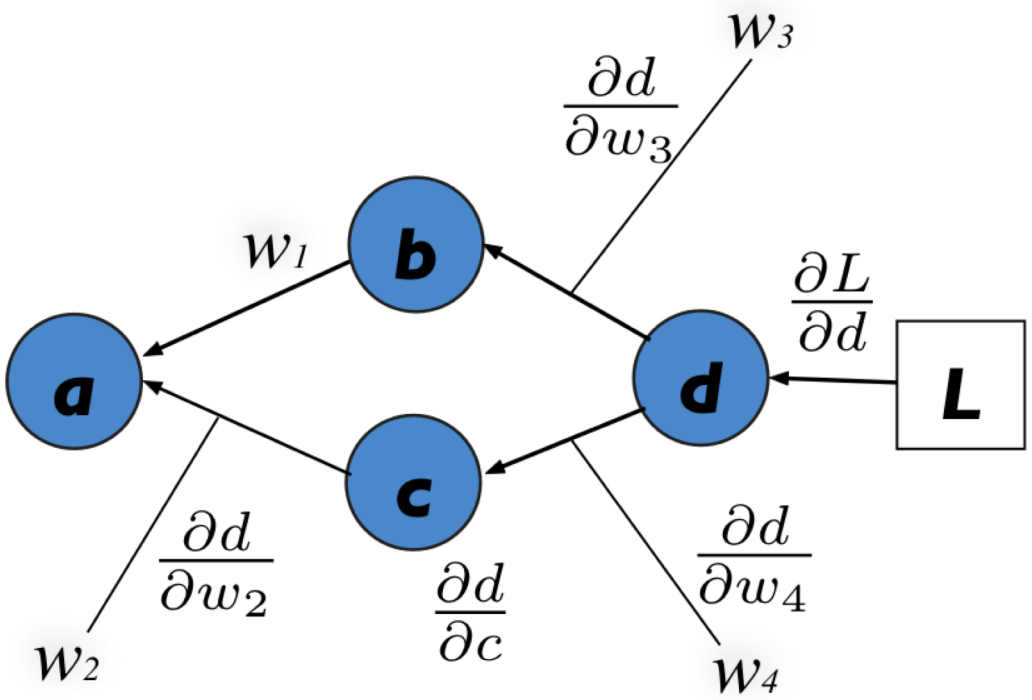
Derivative of a piece of code - The heart of modern deep learning

- Reading the computation graph **backwards** allow to apply the **chain rule** to compute all gradients

```
b = w1 * a
c = w2 * a
d = (w3 * b) + (w4 * c)
L = f(d)
```

- For example w_3: $$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial d}\frac{\partial d}{\partial w_3}$$

- E.g. w_2: $$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial w_2}$$



The graph saves the calculations flow and allow to apply the chain rule easily for all parameters

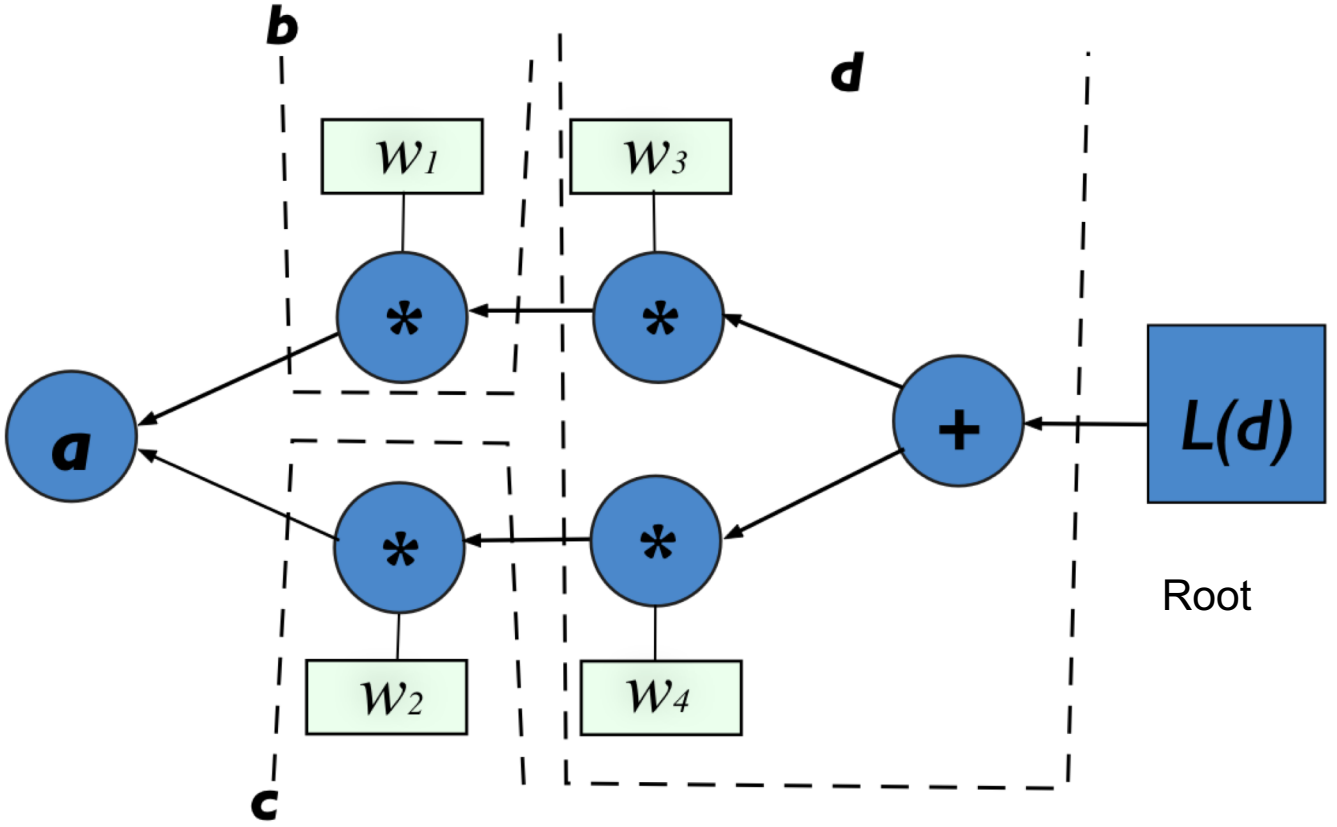
PyTorch and the Computation Graph

From the general to the concrete

PyTorch uses an preexisting automatic differentiation library called **Autograd**

- Autograd is a **reverse** automatic differentiation system
- Autograd records a **graph** recording all of the operations that created the data **as you execute** operations,
- Autograd creates a **directed acyclic graph** whose leaves are the input tensors and roots are the output tensors. By tracing this graph from roots to leaves, you can automatically compute the gradients using the chain rule.

- The computation graph is actually **the graph of operations** not just the nodes



PyTorch and the Computation Graph

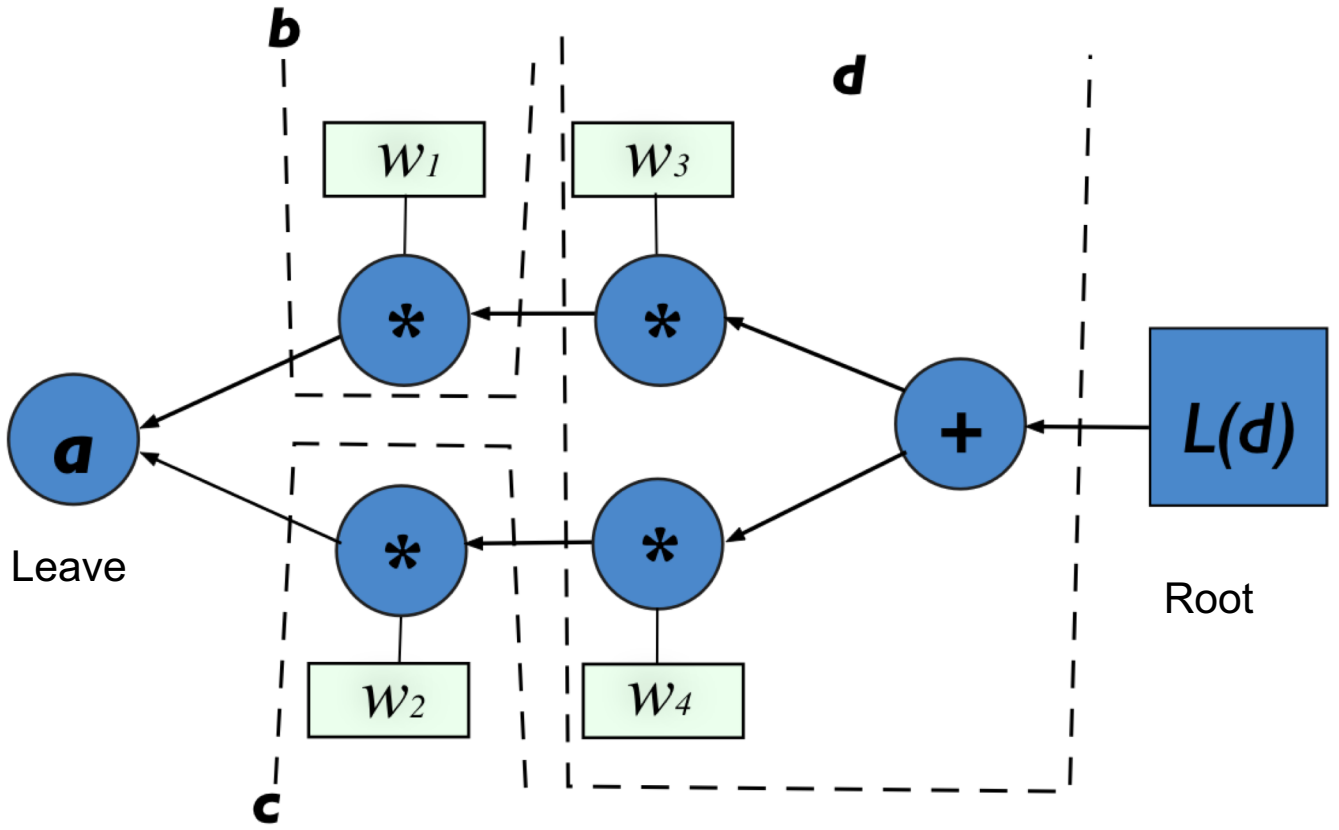
From the general to the concrete

- PyTorch creates this graph on-the-fly when you do you calculation in python

```
b = w1 * a
c = w2 * a
d = (w3 * b) + (w4 * c)
L = f(d)
L.backward()
```

The gradient is then derived when you have defined your calculation by a call to the backward() function of variable L assuming w1 is a PyTorch tensor

- The computation graph is actually **the graph of operations** not just the nodes



NumPy vs PyTorch

PyTorch as a NumPy Replacement

Explicit creation

PyTorch behaves in many respects like a `numpy.array`

- The main object is the `torch.tensor`
  - A tensor is a generalized matrix with arbitrary dimension
  - 0-d tensor is a scalar,
  - 1-d tensor is a vector
  - 2-d tensor is a matrix
  - 3-d tensor
  - 4-d etc
  - All by nested lists [[[...]]]

```
from torch import tensor
```

```
tensor(1.)
```

```
tensor(1.)
```

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

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

```
tensor([[1.1,2.2,3.3],[3.,2.,1]])
```

```
tensor([[1.1000, 2.2000, 3.3000],
        [3.0000, 2.0000, 1.0000]])
```

PyTorch as a NumPy Replacement

Functions to create tensors (some examples)

PyTorch behaves in many respects like a `numpy.array`

- Same-named functions to create tensors
- `torch.empty( ... )` uninitialized memory
- `torch.zeros( ... )`
- `torch.ones( ... )`
- `torch.zeros_like( tensor )`
- `torch.ones_like( tensor )`
- etc.

```
torch.empty(2,3)
```

```
tensor([[                                                0.0000,
                                                         0.0000,
         -1878420505050693180005333518516224000.0000],
        [                                                0.0000,
                                                         0.0000,
                                                         0.0000]])
```

```
torch.zeros(2,2)
```

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

```
torch.ones(5)
```

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

```
torch.zeros_like(M)
```

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

PyTorch as a NumPy Replacement

Functions to create tensors (some examples)

PyTorch behaves in many respects like a `numpy.array`

- Range of numbers
  `torch.arange(start=0,end,step=1,...`
- Normal distributed random matrices
  `torch.randn( ... )`
  or flat in [0,1)
  `torch.rand( ... )`
  and many other distributions and functions to reshuffle the data
- From existing NumPy array
  `torch.from_numpy( ... )`
  takes dtype from NumPy (NumPy default is **float64**!) and the memory stays connected!

```
torch.arange(0,10.)
```

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

```
torch.randn(2,3)
```

```
tensor([[ 0.1451,  1.6575,  0.8467],
        [ 0.2239,  1.4647, -0.0449]])
```

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

```
tensor([0.5591, 0.9101])
```

```
a = np.array([2.,2.])
torch.from_numpy(a)
```

```
tensor([2., 2.], dtype=torch.float64)
```

```
b=torch.from_numpy(a)
b[0]=1
print a,b
```

```
[1. 2.] tensor([1., 2.], dtype=torch.float64)
```

PyTorch as a NumPy Replacement

Functions to create tensors (some examples)

PyTorch behaves in many respects like a `numpy.array`

- There is a `dtype` member telling you the numerical data type
  `M.dtype`
  torch default is **float32**
- There is a `shape` member keeping the dimensions of your tensor
  `M.shape`
- The memory can be reinterpreted
  `M.reshape( ... )` may be a copy
  `M.view( ... )` always a view
  `M.resize( ... )` does not change size but the shape
  `M.resize_( ... )` all `fun_` are in-place operation in pyTorch

```
M=tensor([[1.,2.,3.],[1.1,1.2,1.3]])
M.shape
```

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

```
M.reshape([1,6])
```

```
tensor([[1.0000, 2.0000, 3.0000, 1.1000, 1.2000, 1.3000]])
```

```
M.reshape([1,6]).squeeze()
```

```
tensor([1.0000, 2.0000, 3.0000, 1.1000, 1.2000, 1.3000])
```

```
M.dtype
```

```
torch.float32
```

PyTorch as a NumPy Replacement

Functions to create tensors (some examples)

PyTorch behaves in many respects like a `numpy.array` but not always. **Be aware of differences**!

- The memory can be reinterpreted
  - `M.reshape( ... )` may be a copy
  - `M.view( ... )` always a view
  - `M.resize( ... )` does not change - different to NumPy, instead use:
  - `M.resize_( ... )` all `fun_` are in-place operation in PyTorch

```
nT=np.array([[[111,112],[121,122]],[[211,212],[221,222]]])
print nT
T= tensor([[[111,112],[121,122]],[[211,212],[221,222]]])
print T
```

```
[[[111 112]
  [121 122]]

 [[211 212]
  [221 222]]]
tensor([[[111, 112],
         [121, 122]],

        [[211, 212],
         [221, 222]]])
```

```
nT.resize(8)
print nT
T.resize(8)
print T
```

```
[111 112 121 122 211 212 221 222]
tensor([[[111, 112],
         [121, 122]],

        [[211, 212],
         [221, 222]]])
```

```
T.resize_(8)
print T
```

```
tensor([111, 112, 121, 122, 211, 212, 221, 222])
```

PyTorch as a NumPy Replacement

Accessing elements, slicing and all that

PyTorch behaves in many respects like a `numpy.array`

- `M[1],M[1,2],M[1][2]`
- `M[:,1]` etc.

```
M=torch.arange(0,100).reshape([10,10])
print M
```

```
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, 24, 25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
        [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
        [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
        [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
        [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
        [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
        [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

```
M[1]
```

```
tensor([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

```
M[1,:]
```

```
tensor([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

```
M[:,1]
```

```
tensor([ 1, 11, 21, 31, 41, 51, 61, 71, 81, 91])
```

```
M[:,-2]
```

```
tensor([ 8, 18, 28, 38, 48, 58, 68, 78, 88, 98])
```

```
M[0:2,0:3]
```

```
tensor([[ 0,  1,  2],
        [10, 11, 12]])
```

```
M[0:2][0:3]
```

```
tensor([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

PyTorch as a NumPy Replacement

Broadcasting

```
A+1
```

```
tensor([[2, 3, 4],
        [5, 6, 7]])
```

PyTorch behaves in many respects like a `numpy.array`

- Operations act typically elementwise
- `A+A` coincide with standard matrix addition but
- `A*A` is elementwise

If the shape is different, from the NumPy manual:

The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is "broadcast" across the larger array so that they have compatible shapes.

- Nice feature to write calculations down quickly but also a source of confusion

The same applies to `torch.tensor`

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

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

```
A+A
```

```
tensor([[ 2,  4,  6],
        [ 8, 10, 12]])
```

```
A*A
```

```
tensor([[ 1,  4,  9],
        [16, 25, 36]])
```

```
b=tensor([[1],[2]])
b
```

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

```
A+b
```

```
tensor([[2, 3, 4],
        [6, 7, 8]])
```

PyTorch as a NumPy Replacement

Matrix operation, Linear Algebra and Axes/Dimensions

PyTorch behaves in many respects like a `numpy.array` but not always

- `A.transpose(dim0,dim1)` swapping 2 dimensions
  2-dim version: `A.t()` and `A.t_()`
  - but **no** numpy-style `nA.T`
- `A.sum()` sum of all elements
- `A.sum(0)` sum along **dimension**, same as NumPy **axis**
- `A.mm(A.t())` Matrix multiplication
- `torch.mv(A,b)` Matrix*vector multiplication
  most tensor methods tensor.fun(...)
  also exist as torch.fun(tensor,..)
- Standard linear algebra algorithms, e.g.:
  `u, s, v = torch.svd(a)`
  similar to: `np.linalg.svd(a)`
  but **no** np.linalg.cholesky() but `torch.potrf(a)` using LAPACK naming
- `torch.dot(a,b)` only for 1-d tensors
  differs to np.dot(a,b)

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

```
A.t()
```

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

```
A.sum()
```

```
tensor(21)
```

```
A.sum(0)
```

```
tensor([5, 7, 9])
```

```
A.sum(1)
```

```
tensor([ 6, 15])
```

```
A.mm(A.t())
```

```
tensor([[14, 32],
        [32, 77]])
```

```
b=tensor([2,2,2])
torch.mv(A,b)
```

```
tensor([12, 30])
```

PyTorch Specific Features

GPU support - CUDA semantics

PyTorch comes with genuine GPU support

- There are devices **cuda** and **cpu**
  - A PyTorch object can be sent to these devices if available
  - `A = A.cuda()`
  - `A = A.cpu()`
  - Or directly created on a certain device `torch.ones_like(A,divice=device)`
  - `A.numpy()` creates an error if the tensor is on the GPU
  - Generic form to write device independent code `A.to(device)`
  - A complete neural network model (derived from torch.nn.Module, see later) can also be moved to a GPU `model.cuda()`

```
A = torch.rand(1024,1024)
# check if GPU available
if torch.cuda.is_available():
    device = torch.device("cuda")
else: device = torch.device("cpu")
# move to GPU
A = A.cuda()
# directly create a tensor on GPU
B = torch.ones_like(A, device=device)
# move to GPU
A = A.to(device)
#A.numpy() # error
A=A.cpu()
A.numpy()
```

PyTorch Specific Features

Autograd mechanics

- One of the most important feature is the automatic differentiation by `torch.autograd`
- We have heard about the computation graph
- PyTorch allows to calculate derivatives for tensors (before PyTorch 0.4 this functionality was provide by a different object `variable` - be aware when you google)
- On creation or later you can tell PyTorch that you will need a derivative
  - Named argument `requires_grad=True` `False` on default
- When an tensor with `requires_grad=True` is used in a calculation the resulting tensor gets the attribute `grad_fn` set to a pointer of the applied operation
  - This way PyTorch builds up a linked list of operation to calculate the derivatives

```
a=torch.ones(2,2,requires_grad=True)
a
```

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
```

```
b=a+1
b
```

```
tensor([[2., 2.],
        [2., 2.]], grad_fn=<AddBackward>)
```

```
c=torch.tanh(b)
c
```

```
tensor([[0.9640, 0.9640],
        [0.9640, 0.9640]], grad_fn=<TanhBackward>)
```

```
print c.grad_fn
print c.grad_fn.next_functions
```

```
<TanhBackward object at 0x1108eccd0>
((<AddBackward object at 0x1108ec1d0>, 0L),)
```

PyTorch Specific Features

Autograd mechanics

$$L = \sum_{i=0}^{4} a_i^2$$

$$\frac{dL}{d\vec{a}} = grad(L) = 2\vec{a}$$

- At the end of the necessary calculations you can call `backward()` on the variable of interest, for example a calculated loss.

```
a=torch.tensor([1.,2.,3.,4.,5.],requires_grad=True)
L=torch.sum(a*a)
L.backward()
print L
print a.grad
```

```
tensor(55., grad_fn=<SumBackward0>)
tensor([ 2.,  4.,  6.,  8., 10.])
```

- The graph is erased after backward() is called if not explicitly asked to be retained

PyTorch Specific Features

Autograd mechanics

- Not all operations should be recorded. PyTorch needs a way to do operation that are not added to the computation graph, i.e.
  - `detach()` to create a copy that is not tracked
  - The data member `tensor.data` or `tensor.grad.data`
  - A with block
    ```
    with torch.no_grad():
        ....do something...
    ```

```
a=torch.tensor([1.,2.,3.,4.,5.],requires_grad=True)
b=a+1
c=a.detach()+1
print b
print c
```

```
tensor([2., 3., 4., 5., 6.], grad_fn=<AddBackward>)
tensor([2., 3., 4., 5., 6.])
```

```
c=a.data+1
a=torch.tensor([1.,2.,3.,4.,5.],requires_grad=True)
b=a+1
c=a.data+1
print b
print c
```

```
tensor([2., 3., 4., 5., 6.], grad_fn=<AddBackward>)
tensor([2., 3., 4., 5., 6.])
```

```
a=torch.tensor([1.,2.,3.,4.,5.],requires_grad=True)
b=a+1
with torch.no_grad():
    c=a+1
print b
print c
```

```
tensor([2., 3., 4., 5., 6.], grad_fn=<AddBackward>)
tensor([2., 3., 4., 5., 6.])
```

Please work through the following notebooks:

- tensor_tutorial_basics.ipynb
- tensor_tutorial_timing.ipynb
- autograd_tutorial.ipynb

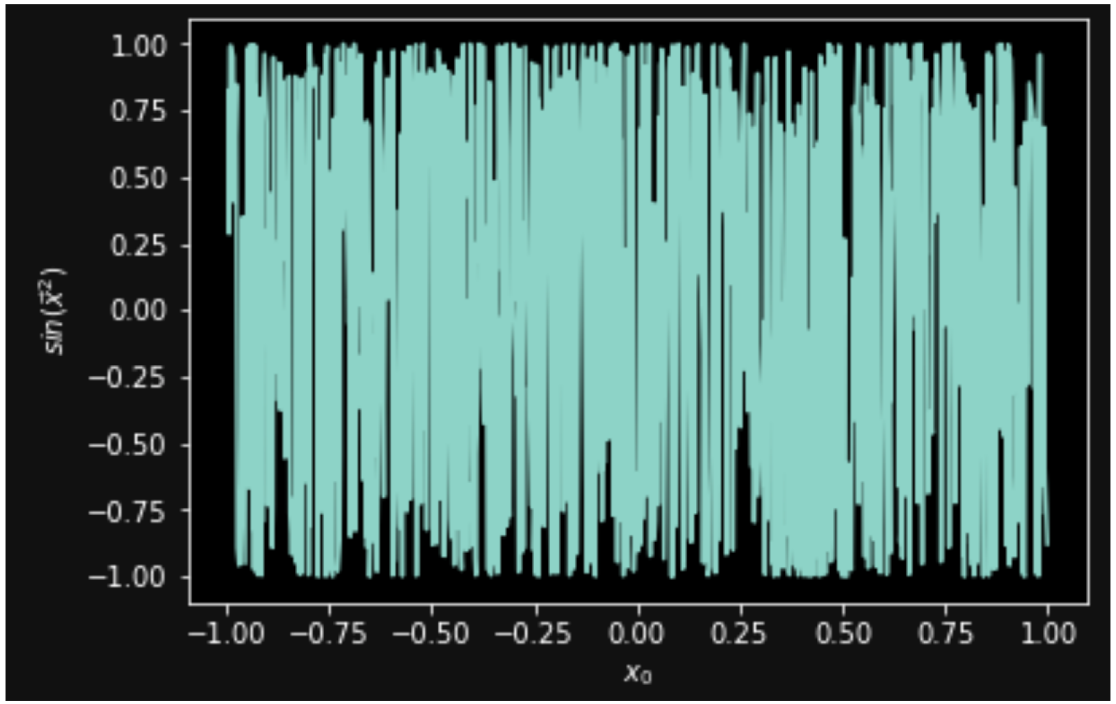
A first Neural Network

A simple model to create some toy data

- After you have got familiar with PyTorch we build and train our first Neural Network
- For this we need some toy data
  - A sine over a parabola in a hundred dimensional space:

$$y = \sin\left(\sum_{i=1}^{100} x_i^2\right)$$

  - The positions $\boldsymbol{x_i}$ are randomly picked from $\boldsymbol{x_i} \in [-1, 1]$
  - The model is fully deterministic but the relation between a single $\boldsymbol{x_i}$ and $\boldsymbol{y}$ looks random.
    The neural net must learn the $\boldsymbol{\sin(|\vec{x}|^2)}$
    from the data.



A first Neural Network

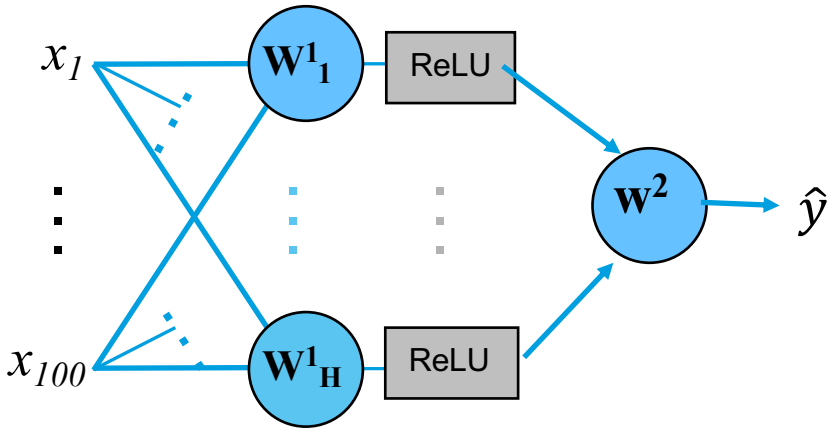
2 layers with backpropagation and gradient descent

$$\underset{D_{out}=1}{\hat{\mathbf{y}}} = \underset{D_{out}\times H}{\mathbf{W}_2} \; F_{ReLU}\Big(\underset{H\times D_{in}=100}{\mathbf{W}_1} \; \mathbf{x}\Big)$$

- 2 layers with backpropagation and gradient descent
- Regression with MSE loss

We build the simple network with different tools

- NumPy
- PyTorch
- PyTorch with Autograd
- PyTorch with high level API



Please work through the following notebook:

- aFirstNN_numpy_to_pyTorch.ipynb

A *personal* Introduction to Deep Learning

- There a millions of tutorials out there
- Here, I try to give you my personal view which is based on my experience as physicist with experience in HEP (and medical imaging)
- Examples and explanations from HEP
  - You may know about optimization from Mechanics (Lagrangian)
  - You may know about statistics from fits and limit setting
  - You may know about linear algebra from detector alignment
  - You know what a tensor is
  - You know a lot about computing
- In short: you know (almost) everything you need to know about Deep Learning
  (if needed we will refresh your memory 😊)

- Neural Networks
  - A little bit of history of studying the brain
  - Neurons as computation units
  - A geometrical view
  - Getting deeper

- Training

loss

regression

classification

methods

  - Stochastic
    - Adam
  - Solving
- Numerical
  - BLAS
  - GPU
- Auto differentiation

Summary

Now we forget 1/2 of what we have learnt

- There a efficient library that sum up all these thing
- Doing Deep Learning means
  - Picking a library of your choice -> PyTorch
  - Define a topology
  - Use a reasonable loss function
  - Take an optimizer
  - And play

Thank you

Contact

DESY. Deutsches
Elektronen-Synchrotron

www.desy.de

Dirk Krücker
CMS
dirk.kruecker@desy.de