Pytorch: A quick Intro

Alexander Binder

University of Oslo (UiO)

January 22, 2022

UiO : Department of Informatics
University of Oslo

Outline

Why PyTorch?

- easy to debug in native python
- currently popular in research[1]
- nowadays pytorch and tensorflow are very similar https://towardsdatascience.com/pytorch-vs-tensorflow-in-2020-fe237862fae1
- automatic differentiation / autograd: computes derivatives of functions for you.
- for those who never heard of it: torchvision model zoo – many networks with pretrained weights ready to load

[1] Older frameworks?

Is there newer stuff than PyTorch?

There cant be nothing else beside PyTorch :P

See e.g. JAX:

https://github.com/google/jax

Key content I

- pytorch tensors: numpy with GPU transfer option
  - linear algebra similar to numpy
  - data is stored in `.data` field
- pytorch broadcasting rules
- torch.einsum for general tensor multiplications with summing
- **pytorch $\rightarrow$ math:** be able to write down mathematically what a certain pytorch operation does
- **math $\rightarrow$ pytorch:** be able to decide how math formula can be realized by which pytorch operations

Key content II

- pytorch autograd:
  - records graph of function computations
  - capable of computing gradient of weighted sum of Jacobi matrix
- when one needs to use only data or handle gradients, tensor have `.data` and `.grad.data` fields

- Installation https://pytorch.org/get-started/locally/
- quick intro: https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- cheat sheet: https://pytorch.org/tutorials/beginner/ptcheat.html

Pytorch modules

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.

Tensor mathematically:

- 1-tensor: a linear mapping $v_1 \mapsto L(v_1)$, representable as $L(v_1) = u \cdot v_1$ by a vector $u = (u_j)$
- 2-tensor: a bilinear mapping $v_1, v_2 \mapsto L(v_1, v_2)$, representable as $L(v_1, v_2) = v_1^t A v_2 = \sum_{ij} v_{1,i} v_{2,j} A_{ij}$ by a matrix $A = (A_{ij})$
- 3-tensor: a trilinear mapping $v_1, v_2, v_3 \mapsto L(v_1, v_2, v_3)$, representable as $L(v_1, v_2, v_3) = \sum_{ijk} v_{1,i} v_{2,j} v_{3,k} A_{ijk}$ by a 3-dim array $A = (A_{ij})$
- n-tensor ... n-linear mapping ... representable by a n-dim array $A = (A_{i_1 \cdots i_n})$
- n-tensors $\leftrightarrow$ n-dim arrays

Tensor?

Same as in physics lectures

- 1-tensor: object/array with 1-dimensional way to index it, vector
  $a[i] \leftrightarrow a_i$
- 2-tensor: object/array with 2-dimensional way to index it, matrix
  $a[i,k] \leftrightarrow a_{i,k}$
- 3-tensor: object/array with 3-dimensional way to index it
  $a[i,k,l] \leftrightarrow a_{i,k,l}$
- n-tensor: object/array with n-dimensional way to index it
- Tensor in pytorch:

  a representation of an numpy-array-like structure
  $A_i,\ A_{i,j,k},\ A_{i,j,k}$ or $A_{i_1,\ldots,i_n}$ with benefits (for storing computed gradients).

- with fixed values:

```
x= torch.zeros((5,1))
y= torch.ones((5))
z= torch.empty((3,2,3))
a = torch.empty((64,32,3,3)).fill_(32.) # initializes to some val
b= a.new_full((3,2),42.) # with same type and device as tensor a
c = torch.full((2, 3), 3.141592)
d = torch.randn((2, 3))
```

- from a saved tensor:
  https://pytorch.org/docs/stable/generated/torch.save.html
  https://pytorch.org/docs/stable/generated/torch.load.html

- from numpy:

```
a=np.random.normal(5,size=(2,3)).astype('float32')
x=torch.tensor(a) # this copies data
x2=torch.as_tensor(a) # this does NOT COPY data,
#and does nothing if its already a tensor with right type, etc.
x3=torch.from_numpy( a) # this does NOT COPY data
#when this can be inappropriate? not resizable tensor as limitati
```

- to numpy:

```
nparr = a.data.numpy() # a.numpy() works only if it has no grad f
```

```
x= torch.empty((2,3)) #empty tensor
```

A tensor has three important properties:

- its `.size()` or `.shape`
- the dtype: its type of numerical elements (most nns use `torch.float32`)
- device it is placed on (e.g. cpu, cuda:0, cuda:1)
- getting its size: output is a `torch.Size()` object.

```
print(x.size())
print(x.shape) # This is a {\tt torch.Size} class instance.
```

Use tuple or list to convert it:

```
xsize=tuple(x.size())
```

get its dtype:

```
print(x.dtype)
```

get its device placement

```
print(x.device) #is a {\tt torch.device} class instance
```

if you need strings, use `.__repr__()`.

Test for equality with

```
x.device==torch.device('cuda:0')
x.dtype==torch.float #rhs is a torch.dtype object
x.dtype.__repr__()=='torch.float32'
```

Important: you can print these anywhere in your execution code. no ugly fixed graph surprises.

current known dtypes and Tensor subclasses

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

current known devices


```
>>> a.to('blabladevice')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: Expected one of cpu, cuda, xpu, mkldnn, opengl, opencl, ideep, hip
, ve, ort, mlc, xla, lazy, vulkan, meta, hpu device type at start of device stri
ng: blabladevice
>>> 
```

https://pytorch.org/cppdocs/api/enum_namespacec10_1a815bc73d9ef8591e4a92a70311b71697.html

- ROCm for AMD
- XLA-compiler driven TPUs https://pytorch.org/xla/release/1.9/index.html (yup, 1.10 is the current PyTorch version). Can try those in google Colab: https://colab.research.google.com/notebooks/intro.ipynb
- Vulkan for Android devices

Type casting


```
print(a.dtype)
b= a.to(torch.float64) # see also the legacy method .type()
c= a.type(torch.DoubleTensor)
```

Device placement


```
print(a.device)
b= a.to('cuda:0')
```

Important knowledge: on multi-GPU clusters (and vanilla jobs) restrict yourself to one device, dont grab all GPUs!

```
CUDA_VISIBLE_DEVICES=2 python3 blablascript.py
```

This uses GPU 2 from the output of nvidia_smi

```
x=torch.ones((10))
y=x.view((2,5))
z=x.view((-1,5)) #-1 joker
```

- Be careful: Which elements ends up where in this case?

```
x=torch.ones((4,2,3))
y=x.view((-1,12))
```

Outline

Exercise will be on broadcasting. Its important for coding.

```
a= torch.full((2,3),3.)
b= torch.full((5,1,3),3.)
c= a+b
```

What will `c.shape` be ?

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

same holds for many binary operators like $+ - */$

$$
\begin{aligned}
a =& torch.ones((4)) \\
b =& torch.ones((1,4)) \\
& torch.add(a,b) && \rightarrow (1,4) \\
a =& torch.ones((4)) \\
b =& torch.ones((4,1)) \\
& torch.add(a,b) && \rightarrow (4,4)!!! \\
a =& torch.ones((3)) \\
b =& torch.ones((4,1)) \\
& torch.add(a,b) && \rightarrow (4,3) \\
a =& torch.ones((3)) \\
b =& torch.ones((1,4)) \\
& torch.add(a,b) && \rightarrow ERR
\end{aligned}
$$

- smaller tensor gets filled **from the left** with singleton dimensions until he has same dimensionality as larger tensor, as if `.unsqueeze(0)` would be applied again and again

1– the smaller tensor gets filled **from the left** with singleton dimensions until he has same dimensionality as larger tensor, as if `.unsqueeze(0)` would be applied again and again

2– then check whether they are compatible – they are incompatible if in one dimension both tensors have sizes > 1 which are not equal. if they are incompatible, you will get an error.

3– whenever a dimension with size 1 meets a dimension with a size $k > 1$, then the smaller vector is replicated/copied $k - 1$ times in this dimension until he reaches in this dimension size k and your actual operation is applied

Example:

start	after insert	after copying
(4,1)	(4,1)	(4,4)
(4)	(1,4)	(4,4)

Broadcasting

More examples:

start	after insert	after copying
(1,3)	(1,3)	(1,3)
(3)	(1,3)	(1,3)
start	**after insert**	**after copying**
(2,3)	(1,2,3)	(5,2,3)
(5,1,3)	(5,1,3)	(5,2,3)
start	**after insert**	**after copying**
(1,7)	(1,1,1,7)	(5,2,3,7)
(5,2,3,7)	(5,2,3,7)	(5,2,3,7)
start	**after insert**	**after copying**
(4,1)	(1,4,1)	ERR
(2,3,7)	(2,3,7)	ERR

if broadcasting is too ... , then apply `.unsqueeze(dim)` on your tensor, until both tensors have the same number of dimension axes. The only thing what is done then, is copying along $dim = 1$ axes.

Outline

`torch.mm(a,b)` dot product, not broadcasting. a, b must be 1-tensors

$$a.size() = (d),\ \ b.size() = (d)$$
$$torch.dot(a, b) = \sum_{d'} a_{d'} b_{d'} = \sum_{d'} a[d']b[d']$$
$$\rightarrow torch.dot(a, b).size() = ()$$

`torch.mm(A,B)` matrix multiplication, not broadcasting. A, B must be 2-tensors

$$A.size() = (i, k),\ \ B.size() = (k, l)$$
$$torch.mm(A, B)[i, l] = \sum_{k'} A_{i,k'} B_{k',l} = \sum_{k} A[i, k']B[k', l]$$
$$\rightarrow torch.mm(A, B).size() = (i, l)$$

`torch.bmm(A,B)` **batched** matrix multiplication, not broadcasting. A, B must be 3-tensors. multiplication along last dim of A and second dim of B.

$$A.size() = (b, i, k), \;\; B.size() = (b, k, l)$$

$$torch.bmm(A, B)[b, i, l] = \sum_{k'} A_{b,i,k'} B_{b,k',l} = \sum_{k} A[b, i, k'] B[b, k', l]$$

$$\rightarrow torch.bmm(A, B).size() = (b, i, l)$$

`torch.bmm(A,B)` performs for every index k a matrix multiplication between $A[k, :, :]$ and $B[k, :, :]$
– its a for loop over k of `torch.mm(A[k,:,:], B[k,:,:])`

Think: $torch.bmm(A, B)$ given a known shape of A puts what restrictions on B??

shapes dont fit?!

example: want to compute matrix vector product by `mm(.)` $(vA)_l = \sum_k^K v_k A_{k,l}$, $v.shape = (K)$.

v is 1-tensor, cannot use `torch.mm(`v, A`)`. add a dim in v:

$torch.mm(v.unsqueeze(0), A)$	$(1, K) \cdot (K, L) \rightarrow (1, L)$
$torch.mm(v.unsqueeze(0), A).squeeze(0)$	$(1, K) \cdot (K, L) \rightarrow (1, L) \rightarrow (L)$

`torch.squeeze(A,dim=2)` - remove singleton dim
$(a, b, 1, c) \rightarrow (a, b, c)$
`torch.unsqueeze(A,dim=1)` - insert singleton dim
$(a, b, c) \rightarrow (a, 1, b, c)$
`torch.unsqueeze(A,dim=0)` - insert singleton dim
$(a, b, c) \rightarrow (1, a, b, c)$

shapes dont fit?!

...

`torch.transpose(A,dim1,dim2)` swaps two dimensions

`torch.Tensor.permute(*dims)` permutes a set of dimensions rather than just swapping two

I have to implement:

$$T_{i,n,r,s} = \sum_{k,m,o} A_{i,k,m,n,o} B_{m,o,r} C_{k,m,r,s}$$

Outline

a general way to do all kinds of batched and non-batched tensor multiplications: torch.einsum

https://rockt.github.io/2018/04/30/einsum

rule:

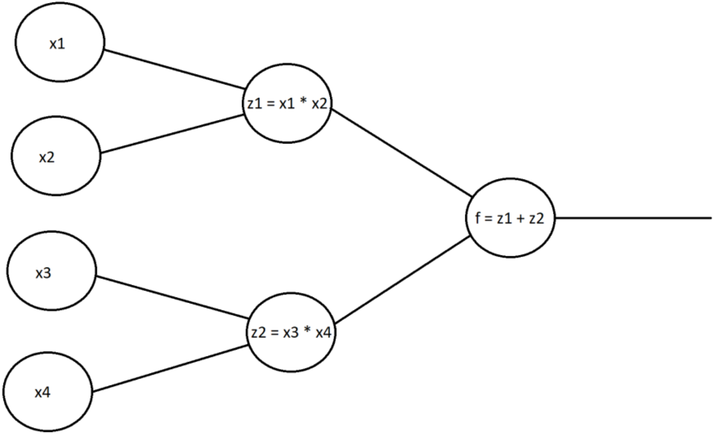
- left of $->$: all tensors separated by , which are to be multiplied and summed.
- **indices that have same name in multiple tensors, will get multiplied together**
- right of $->$ the result tensor with remaining indices.
- **All indices missing right of $->$ are summed out so that they vanish in the result.**

Outline

A directed-graph representation of computations done.

What is a computational graph?

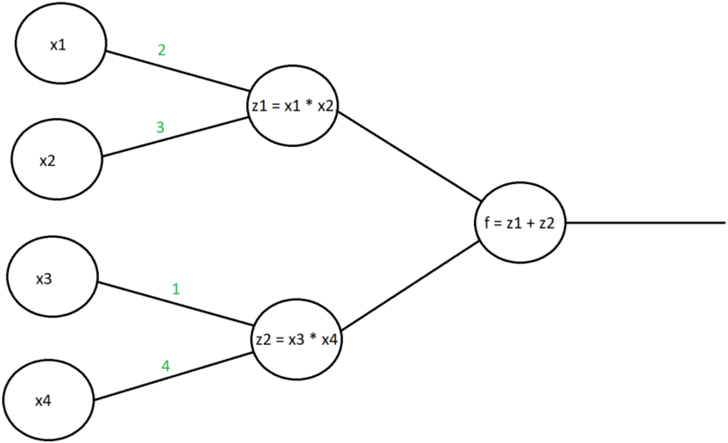
$f(\vec{x}) = x_1 * x_2 + x_3 * x_4 \qquad f(\vec{x}) = z_1 + z_2$



Forward pass: the actual computation

Forward propagation

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

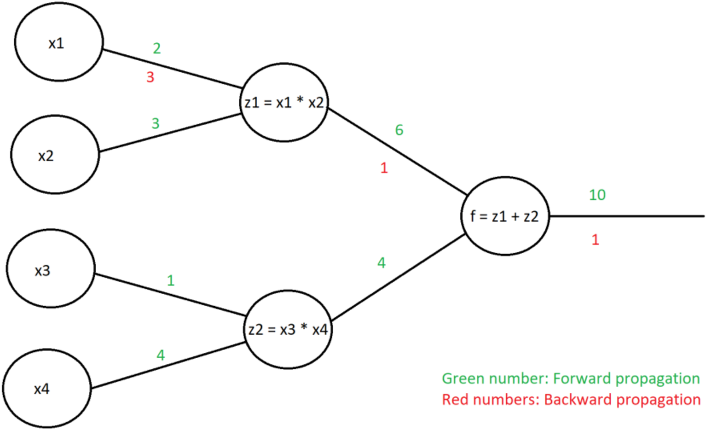


Backward pass: computing derivates

Backward propagation

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

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



What ? Automatic differentiation with respect to variables used in computations.
You can define a sequence of computations, then call `.backward()` or `torch.autograd.grad(...)`. see `autograf2.py`, `print_computationalgraph.py`

When ?

- If tensors are leaf tensors and have the `requires_grad=True` flag set, then they are marked for tracking operations along the computation sequence for later gradient computation.
- leaf tensor: not created as the result of an operation but defined by you as an input.

https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py

...

if `e` is a tensor with 1 element, then `e.backward()` computes the gradient of `e` with respect to all its inputs that were involved in computing `e`.

see `print_computationalgraph.py`: the whole backward graph if e is a tensor of $n \geq 2$ elements, then the gradient of e is a matrix, the Jacobi-matrix. Example for 3 elements:

$$e = (e_1, e_2, e_3)$$

$$de/dx = \begin{pmatrix} \frac{de_1}{dx_1} & \frac{de_2}{dx_1} & \frac{de_3}{dx_1} \\ \frac{de_1}{dx_2} & \frac{de_2}{dx_2} & \frac{de_3}{dx_2} \\ \vdots & \vdots & \vdots \\ \frac{de_1}{dx_8} & \frac{de_2}{dx_8} & \frac{de_3}{dx_8} \\ \vdots & \vdots & \vdots \\ \frac{de_1}{dx_D} & \frac{de_2}{dx_D} & \frac{de_3}{dx_D} \end{pmatrix}$$

if e is a tensor of $n \geq 2$ elements, then the gradient of e is a matrix, the Jacobi-matrix.
In this case: (for an example where e has 3 elements)
`e.backward(torch.tensor([-5,2,6]) )` computes the D-dim weighted gradient vector

$$\frac{de_1}{dx} * (-5) + \frac{de_2}{dx} * 2 + \frac{de_3}{dx} * 6$$
$$= \begin{pmatrix} \frac{de_1}{dx_1} * (-5) + \frac{de_2}{dx_1} * 2 + \frac{de_3}{dx_1} * 6 \\ \frac{de_1}{dx_2} * (-5) + \frac{de_2}{dx_2} * 2 + \frac{de_3}{dx_2} * 6 \\ \vdots \\ \frac{de_1}{dx_8} * (-5) + \frac{de_2}{dx_8} * 2 + \frac{de_3}{dx_8} * 6 \\ \vdots \\ \frac{de_1}{dx_D} * (-5) + \frac{de_2}{dx_D} * 2 + \frac{de_3}{dx_D} * 6 \end{pmatrix}$$

This is an inner product between the jacobi matrix and a vector that has as many elements as e in the forward pass.

Autograd

- Autograd tracks the graph of computations
- Tracked computations will be used to compute a gradient automatically
- use `with torch.no_grad():` environment to **not** record computations for gradient calculations for some larger block of code that is reused – use case: everything outside of handling training data, e.g. computing validation or test scores.[a]
- outlook / out of class: for GAN-training `sometensor.detach()` prevents the gradient flowing from `sometensor` to all those parts used to compute sometensor.

[a] Why you dont want to track gradient computations in this case?

Note: If you have a tensor with attached gradient, then the `.data` stores the tensor values, and `.grad.data` the gradient values

```
vals=x.data.numpy() #exports function values to numpy
g_vals=x.grad.data.numpy() #exports gradient values to numpy
```

Others

- useful stuff: standard operations like mean or max, `torch.random`, `torch.nn.functional`
- Things behaving unexplainably weird? check your version: `print(torch.__version__)`

The end

Questions?!