50.021 – AI

Alex

Week 02: Pytorch

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

Key content

- pytorch tensors: numpy with GPU transfer option
  - linear algebra similar to numpy
  - torch.einsum for general tensor multiplications with summing
  - data is stored in `.data` field
- pytorch broadcasting rules
- when one needs to use only data or handle gradients, tensor have `.data` and `.grad.data` fields

1 Pytorch tensor basics

`https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py`

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 in pytorch:

a representation of an numpy-array-like structure A_i or $A_{i,j,k}$ or $A_{i,j,k}$ or $A_{i,j,k,l}$ with possibly more than 2 indices with benefits (for storing computed gradients).

1.1 init tensor as zeros, ones, constants

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

A tensor has three important properties:

- its .size()
- the dtype: its numerical type (most nns use `torch.float32`)
- device it is placed on (cpu, cuda:0, cuda:1)

getting its size: output is a `torch.Size()` object.

```
print(x.size())
```

Use `list` or `tuple` to get a list/tuple from that.

```
xs=tuple(x.size())
print(type(xs))
print(xs)
print(xs[0])
```

get its dtype:

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

get its device placement

```
print(x.device)
```

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.

```
x= torch.zeros((5,1))
y= torch.ones((5))
z= torch.empty((3,2,3))
a= torch.new_full((3,2),42.) # tensor with a value
```

`https://pytorch.org/docs/stable/tensors.html` – dtypes and tensor types. dtype is the type of one element. tensor type depends on dtype and device placement

1.2 tensor from numpy

```
a=np.random.normal(5,size=(2,3),dtype=np.float32)
x=torch.tensor( a) % this copies data
x2=torch.from_numpy( a) % this does NOT COPY data - when this can be inappropriate?
```

1.3 tensor to numpy

```
t=x.data.numpy() #x.numpy() works only if x has no gradient attached )later(
```

1.4 change shape: reshape a tensor

```
a.view(...)
```

```
x=torch.ones((10))
x=x.view((-1,5))
```

usually changing the number of

1.5 change device: move tensor to gpu / cpu device

```
device=torch.device('cuda:0')
xg=x.to(device)
xc=x.to(torch.device('cpu'))
```

1.6 change tensor dtype

print your type

```
print(x.type())
```

cast tensor

```
x=x.type(torch.FloatTensor)
x=x.type_as(a) # a is another tensor
```

1.7 create tensor of same type and device as another tensor

This is useful when one uses `torch.nn.DataParallel` for multi-GPU computations – then one does not want to instantiate a tensor explicitly on a fixed GPU like `cuda:0`.

```
b=a.new_ones((3,2))
```

1.8 other useful stuff

```
res=torch.where(x>5,x,y)
```

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

Debugging in pytorch

Most of my own programming errors come from above three properties: try to compute results

- with incompatible shapes.
- from two tensors with incompatible numerical type (integer and float, float32 and double),
- with incompatible devices (one on cpu, other on GPU),

The good news: in pytorch you can print `.size()` anywhere in the running code, also its dtype and its device placement

debugging advice: RTFM and print the shapes.

1.9 linear algebra: sum, inner product, matrix product

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

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

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

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

`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.

$$\begin{aligned} A.size() &= (b,i,k) \\ B.size() &= (b,k,l) \\ torch.bmm(A,B)[b,i,l] &= \sum_{k'} A_{b,i,k'} B_{b,k',l} = \sum_{k} A[b,i,k']B[b,k',l] \rightarrow torch.bmm(A,B).size() = (b,i,l) \end{aligned}$$

`torch.matmul(A,B)` - matrix multiplication with broadcasting - that is only 1 dimension is summed out `https://pytorch.org/docs/stable/torch.html#`

`torch.matmul` This function is a bit tricky, performs broadcasting of shapes (`https://pytorch.org/docs/stable/notes/broadcasting.html`), then multiplies along the last dimension of A, and over the second last dimension of B – `torch.einsum` is more clear here.

Broadcasting warning

You cannot avoid getting to know the broadcasting rules (quiz?).

many simple functions like $+$, $*$ do broadcasting by default.

`torch.tensordot(A,B, dims=(list1,list2) )` - general tensor contractions with broadcasting – that is multiple dimensions can be summed out `https://pytorch.org/docs/stable/torch.html#torch.tensordot`

torch.einsum

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.

1.10 linear algebra: shapes dont fit?!

`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 $(a, b, c) \rightarrow (1, a, b, c)$

example: want to compute with mm matrix vector product $(vA)_l = \sum_k v_k A_{k,l}$. v is 1-tensor, so cannot use `torch.mm(`v, A`)`. So add a singleton dimension in v:

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

`torch.transpose(A,dim1,dim2)` swaps two dimensions `torch.Tensor.permute(*dims)` permutes a set of dimensions rather than just swapping two

2 broadcasting

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

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

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

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

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

- 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

$$\begin{aligned} a &= torch.ones((2,5)) \\ b &= torch.ones((5)) \\ torch.add(a,b) &\to (2,5) \\ \to a &= torch.ones((2,5)) \text{ ok as is} \\ \to b &= torch.ones((1,5)) \text{ copy 1x until } (2,5) \end{aligned}$$

$$\begin{aligned} a &= torch.ones((3)) \\ b &= torch.ones((4,1)) \\ torch.add(a,b) &\to (4,3) \\ \to a &= torch.ones((1,3)) \text{ copy 3x until } (4,3) \\ \to b &= torch.ones((4,1)) \text{ copy 2x until } (4,3) \end{aligned}$$

- 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