50.039 – Theory and Practice of Deep Learning

Alex

Week 04: Pytorch part II – Autograd

[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 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

1 broadcasting

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

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

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

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

$$a = torch.ones((3))$$
$$b = torch.ones((1,4))$$
$$torch.add(a,b) \rightarrow ERR$$

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

4– your actual operation is applied

Examples:

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

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 ...

close to rituals that belong into a witches' place, 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.

2 Pytorch autograd

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

You can define a sequence of computations. see `autograf2.py, autograf3.py`

If tensors have the `requires_grad=True` flag set, then they are marked for tracking gradients along the computation sequence.

see `autograf2.py`:

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

see `autograf3.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:

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

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

then `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.

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
```