

# Pytorch学习笔记

肖桐 PB18000037

## 1. Tensors的创建

### (1). `torch.empty()`

未初始化，tensor中的值由分配空间中原来的数据决定。如：

```
1 | x = torch.empty(5, 3)
```

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

创建一个所有元素都是在[0, 1]之间的随机数的tensor。如：

```
1 | x = torch.rand(5, 3)
```

### (3). `torch.zeros()`

创建一个所有元素都被初始化为0的tensor。如：

```
1 | x = torch.zeros(5, 3)
```

### (4). `torch.ones()`

创建一个所有元素都被初始化为1的tensor。如：

```
1 | x = torch.ones(5, 3)
```

### (5). `torch.tensor()` 或 `torch.Tensor()`

直接通过给定的数据创建tensor，方法的参数应该是一个list，或者numpy对象。如：

```
1 | x = torch.tensor([5.5, 3])
```

### (6). `.new_XXX()` 和 `.XXX_like()`

这两个方法是基于一个已经存在的tensor创建新的tensor。两个方法都会继承输入的tensor的数据类型，除非使用 `dtype` 属性明确指出新建tensor的数据类型。如：

```
1 | x = x.new_ones(5, 3, dtype = torch.double)
2 | x = torch.randn_like(x, dtype = torch.int32)
```

## 2. Tensors的属性查看

## (1). `.size()`

`.size()` 方法查看tensor的维度。如：

```
1 x = torch.tensor(5, 3)
2 print(x.size())
3
4 输出: torch.Size([5, 3])
```

## (2).

# 3. Tensors的操作

## (1). 基本运算（以加法为例）

加法可以有多种表示方法，如下面的两种操作结果都相同。（有无特殊情况使得这两种操作结果不同？）

```
1 x = torch.rand(5, 3)
2 y = torch.ones(5, 3)
3
4 a = x + y
5 b = torch.add(x, y) #也可以表示为 torch.add(x, y, out = b)
6
```

在操作方法后面加上一个 `_` 可以改变原tensor，如：

```
1 y.add_(x) #表示将y加上x，并保存在y中
```

同样的还有 `y.copy_(x)`，`y.t_(x)` 等等。

## (2). 索引操作

tensor的维度：最先表示的是最高的维度，如：

```
1 x = torch.tensor(2, 3, 4)
```

对于x，第三维长度为2，第二维长度为3，第一维长度为4。

torch支持所有numpy和python列表的索引操作。

## (3). reshape操作: `.view()`

可以通过使用 `.view()` 方法对tensor进行reshape，改变维度以及各个维度的长度。如：

```
1 x = torch.randn(4, 4)
2 y = x.view(16) #改为1维，第一维长度为16
3 z = x.view(-1, 8) #改为2维，第一维长度为8，-1表示根据其他维度长度确定，故这里第二维长度应该为2
```

## (4). `.item()`

`.item()` 方法可以将仅有一个元素的tensor转变为一个Python数字。如：

```
1 x = torch.tensor([-1.2])
2 print(x)      #输出-1.2
```

## (5). `.sum()`

`.sum()` 操作对tensor中的所有元素进行求和，得到一个只有一个元素的tensor，该元素即为原tensor所有元素求和得到的值。如：

```
1 x = torch.ones(2, 2)
2 y = x.sum()
3 print(y, type(y))
4
5 输出：
6 tensor(4.) <class 'torch.Tensor'>
```

## (5). `.mean()`

`.mean()` 方法对tensor中的所有元素进行求平均值，得到一个只有一个元素的tensor，该元素即为原tensor所有元素求和得到的值。

# 4. AutoGrad自动求导

## (1). 基本知识 or 前提知识

`torch.Tensor` is the central class of the package. If you set its attribute `.requires_grad` as `True`, it starts to track all operations on it. When you finish your computation you can call `.backward()` and have all the gradients computed automatically. The gradient for this tensor will be accumulated into `.grad` attribute.

简而言之，将 `torch.Tensor` 类的 `.requires_grad` 属性设置为 `True` 之后，可以调用方法 `.backward()` 对其进行自动求梯度，求出的梯度存放在 `.grad` 属性中。(对 `requires_grad=True` 的 tensor 经"操作"产生的 tensor 的 `requires_grad` 属性是否也为 `True`？)

除了在新建 tensor 便对 `requires_grad` 属性进行设置之外，也可以通过方法 `.requires_grad_()` 进行设置。如：

```
1 a = torch.randn(2, 2)
2 a.requires_grad_(True) #如果不传递参数则默认为False
```

也可以通过调用 `.detach()` 和 `with torch.no_grad()` 方法取消自动求导。(有什么区别？)

`with torch.no_grad()` 使用：

```

1 print(x.requires_grad)
2 print((x ** 2).requires_grad)
3
4 with torch.no_grad():
5     print((x ** 2).requires_grad)
6
7 输出:
8 True
9 True
10 False

```

`.detach()` 使用: 该方法是复制一个 `.requires_grad = False` 但是内容完全相同的tensor:

```

1 print(x.requires_grad)
2 y = x.detach()
3 print(y.requires_grad)
4 print(x.eq(y).all())
5
6 输出:
7 True
8 False
9 tensor(True)

```

Pytorch中还有一个 `Function` 类对于自动求导的实现是十分重要的。

`Tensor` and `Function` are interconnected and build up an acyclic graph, that encodes a complete history of computation. Each tensor has a `.grad_fn` attribute that references a `Function` that has created the `Tensor` (except for Tensors created by the user - their `grad_fn` is `None`).

简而言之, 每个经过"操作"所创建的tensor会有一个 `grad_fn` 属性(或者说这些tensor的 `grad_fn` 属性不为 `None`), 而由用户直接创建的tensor的 `grad_fn` 属性为 `None`。如:

```

1 x = torch.ones(2, 2, requires_grad=True)
2 y = x + 2
3 print(x.grad_fn)
4 print(y.grad_fn)
5
6 输出:
7 None
8 <AddBackward0 object at 0x0000027DCB6B2880> #这是什么意思?

```

If you want to compute the derivatives, you can call `.backward()` on a `Tensor`. If `Tensor` is a scalar (i.e. it holds a one element data), you don't need to specify any arguments to `backward()`, however if it has more elements, you need to specify a `gradient` argument that is a tensor of matching shape.

即调用方法 `.backward()` 是需要向其中传递参数的, 所传的参数与tensor的维度有关。如果tensor为标量则不需要传参数。

## (2). 开始求梯度!

比如现在构造一个只有一个元素的tensor:

```

1 x = torch.ones(2, 2, requires_grad = True)
2 y = x + 2
3 z = y * y * 3
4 out = z.mean()

```

此时 `out` 即使只有一个元素的tensor。显然 `out` 是向量  $x$  的函数。

而且此时 `out` 为一个标量，故对 `out` 求梯度：

```

1 out.backward() #等价于 out.backward(torch.tensor(1.))
2 print(x.grad) #求出的梯度为什么在x的grad属性中？
3
4 输出：
5 tensor([[4.5, 4.5],
6         [4.5, 4.5]])

```

计算过程如下：

$$out = \frac{1}{4} \sum_i z_i, \text{ 而 } z_i = 3(x_i + 2)^2, \text{ 且 } z_i|_{x_i=1} = 27. \text{ 因此 } \frac{\partial out}{\partial x_i} = \frac{3}{2}(x_i + 2), \text{ 因此}$$

$$\left. \frac{\partial out}{\partial x_i} \right|_{x_i=1} = 4.5$$

一般地，如果有一个向量方程  $\vec{y} = f(\vec{x})$ ，则  $y$  关于  $x$  的梯度是一个雅各比矩阵 (*Jacobian Matrix*):

$$J = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_n} \end{pmatrix}$$

Generally speaking, `torch.autograd` is an engine for computing vector-Jacobian product. That is, given any vector  $v = (v_1, v_2, \dots, v_m)^T$ , compute the product  $v^T \cdot J$ . If  $v$  happens to be the gradient of a scalar function  $l = g(\vec{y})$ , that is,  $v = \left( \frac{\partial l}{\partial y_1} \cdots \frac{\partial l}{\partial y_m} \right)^T$ , then by the chain rule, the vector-Jacobian product would be the gradient of  $l$  with respect to  $\vec{x}$ :

$$J^T \cdot v = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \vdots \\ \frac{\partial l}{\partial x_n} \end{pmatrix}$$

(Note that  $v^T \cdot J$  gives a row vector which can be treated as a column vector by taking  $J^T \cdot v$ .)

当  $l$  不是标量时，需要调用 `.backward()` 方法，同时需要向该方法传入一个向量(tensor)。如：

```

1 x = torch.randn(3, requires_grad = True)
2 y = x * 2 #此时y为一个3维向量
3 v = torch.tensor([0.1, 1.0, 0.0001], dtype = torch.float)
4 #为什么是[0.1, 1.0, 0.0001]? 有什么含义? 怎么进行计算?
5 y.backward(v)
6 print(x.grad)

```

