2.2 数据操作

在深度学习中，我们通常会频繁地对数据进行操作。作为动手学深度学习的基础，本节将介绍如何对内存中的数据进行操作。

在PyTorch中， `torch.Tensor` 是存储和变换数据的主要工具。如果你之前用过NumPy，你会发现 `Tensor` 和NumPy的多维数组非常类似。然而， `Tensor` 提供GPU计算和自动求梯度等更多功能，这些使 `Tensor` 更加适合深度学习。

> **"tensor"这个单词一般可译作“张量”，张量可以看作是一个多维数组。标量可以看作是0维张量，向量可以看作1维张量，矩阵可以看作是二维张量。**

2.2.1 创建 `Tensor`

我们先介绍 `Tensor` 的最基本功能，即 `Tensor` 的创建。

首先导入PyTorch：

```
import torch
```

然后我们创建一个5x3的未初始化的 `Tensor` ：

```
x = torch.empty(5, 3)
print(x)
```

输出：

```
tensor([[ 0.0000e+00,  1.5846e+29,  0.0000e+00],
        [ 1.5846e+29,  5.6052e-45,  0.0000e+00],
        [ 0.0000e+00,  0.0000e+00,  0.0000e+00],
        [ 0.0000e+00,  0.0000e+00,  0.0000e+00],
        [ 0.0000e+00,  1.5846e+29, -2.4336e+02]])
```

创建一个5x3的随机初始化的 `Tensor` ：

```
x = torch.rand(5, 3)
print(x)
```

输出：

```
tensor([[0.4963, 0.7682, 0.0885],
        [0.1320, 0.3074, 0.6341],
        [0.4901, 0.8964, 0.4556],
        [0.6323, 0.3489, 0.4017],
        [0.0223, 0.1689, 0.2939]])
```

创建一个5x3的long型全0的 `Tensor` ：

```
x = torch.zeros(5, 3, dtype=torch.long)
print(x)
```

输出：

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

还可以直接根据数据创建:

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

输出：

```
tensor([5.5000, 3.0000])
```

还可以通过现有的 `Tensor` 来创建，此方法会默认重用输入 `Tensor` 的一些属性，例如数据类型，除非自定义数据类型。

```
x = x.new_ones(5, 3, dtype=torch.float64)  # 返回的tensor默认具有相同的torch.dtype和to
print(x)

x = torch.randn_like(x, dtype=torch.float) # 指定新的数据类型
print(x)
```

输出：

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)
tensor([[ 0.6035,  0.8110, -0.0451],
        [ 0.8797,  1.0482, -0.0445],
        [-0.7229,  2.8663, -0.5655],
        [ 0.1604, -0.0254,  1.0739],
        [ 2.2628, -0.9175, -0.2251]])
```

我们可以通过 `shape` 或者 `size()` 来获取 `Tensor` 的形状:

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

输出：

```
torch.Size([5, 3])
torch.Size([5, 3])
```

> **注意：返回的torch.Size其实就是一个tuple, 支持所有tuple的操作。**

还有很多函数可以创建 `Tensor` ，去翻翻官方API就知道了，下表给了一些常用的作参考。

函数	功能
Tensor(*sizes)	基础构造函
tensor(data,)	类似np.ar的构函
ones(*sizes)	全1Ten
zeros(*sizes)	全0Ten
eye(*sizes)	对角为1，他为
arange(s,e,step)	从s到e，步为st
linspace(s,e,steps)	从s到e，均切分steps
rand/randn(*sizes)	均匀/准分
normal(mean,std)/uniform(from,to)	正态布/均分布
randperm(m)	随机列

这些创建方法都可以在创建的时候指定数据类型dtype和存放device(cpu/gpu)。

2.2.2 操作

本小节介绍 `Tensor` 的各种操作。

算术操作

在PyTorch中，同一种操作可能有很多种形式，下面用加法作为例子。

- **加法形式一**

```
y = torch.rand(5, 3)
print(x + y)
```

- **加法形式二**

```
print(torch.add(x, y))
```

还可指定输出：

```
result = torch.empty(5, 3)
torch.add(x, y, out=result)
print(result)
```

- **加法形式三、inplace**

```
# adds x to y
y.add_(x)
print(y)
```

> **注：PyTorch操作inplace版本都有后缀 `_` ，例如 `x.copy_(y)`, `x.t_()`**

以上几种形式的输出均为：

```
tensor([[ 1.3967,  1.0892,  0.4369],
        [ 1.6995,  2.0453,  0.6539],
        [-0.1553,  3.7016, -0.3599],
        [ 0.7536,  0.0870,  1.2274],
        [ 2.5046, -0.1913,  0.4760]])
```

索引

我们还可以使用类似NumPy的索引操作来访问 `Tensor` 的一部分，需要注意的是：**索引出来的结果与原数据共享内存，也即修改一个，另一个会跟着修改。**

```
y = x[0, :]
y += 1
print(y)
print(x[0, :]) # 源tensor也被改了
```

输出：

```
tensor([1.6035, 1.8110, 0.9549])
tensor([1.6035, 1.8110, 0.9549])
```

除了常用的索引选择数据之外，PyTorch还提供了一些高级的选择函数:

函数	功能
index_select(input, dim, index)	在指定维度dim上选取，比如选取某些行、某些列
masked_select(input, mask)	例子如上，a[a>0]，使用ByteTensor进行选取
nonzero(input)	非0元素的下标
gather(input, dim, index)	根据index，在dim维度上选取数据，输出的size与index一样

这里不详细介绍，用到了再查官方文档。

改变形状

用 `view()` 来改变 `Tensor` 的形状：

```
y = x.view(15)
z = x.view(-1, 5)  # -1所指的维度可以根据其他维度的值推出来
print(x.size(), y.size(), z.size())
```

输出：

```
torch.Size([5, 3]) torch.Size([15]) torch.Size([3, 5])
```

注意 `view()` 返回的新 `Tensor` 与源 `Tensor` 虽然可能有不同的 `size` ，但是是共享 `data` 的，也即更改其中的一个，另外一个也会跟着改变。(顾名思义，view仅仅是改变了对这个张量的观察角度，内部数据并未改变)

```
x += 1
print(x)
print(y) # 也加了1
```

输出：

```
tensor([[1.6035, 1.8110, 0.9549],
        [1.8797, 2.0482, 0.9555],
        [0.2771, 3.8663, 0.4345],
        [1.1604, 0.9746, 2.0739],
        [3.2628, 0.0825, 0.7749]])
tensor([1.6035, 1.8110, 0.9549, 1.8797, 2.0482, 0.9555, 0.2771, 3.8663, 0.4345,
        1.1604, 0.9746, 2.0739, 3.2628, 0.0825, 0.7749])
```

所以如果我们想返回一个真正新的副本（即不共享data内存）该怎么办呢？Pytorch还提供了一个 `reshape()` 可以改变形状，但是此函数并不能保证返回的是其拷贝，所以不推荐使用。推荐先用 `clone` 创造一个副本然后再使用 `view` 。**参考此处**

```
x_cp = x.clone().view(15)
x -= 1
print(x)
print(x_cp)
```

输出:

```
tensor([[ 0.6035,  0.8110, -0.0451],
        [ 0.8797,  1.0482, -0.0445],
        [-0.7229,  2.8663, -0.5655],
        [ 0.1604, -0.0254,  1.0739],
        [ 2.2628, -0.9175, -0.2251]])
tensor([1.6035, 1.8110, 0.9549, 1.8797, 2.0482, 0.9555, 0.2771, 3.8663, 0.4345,
        1.1604, 0.9746, 2.0739, 3.2628, 0.0825, 0.7749])
```

> **使用 `clone` 还有一个好处是会被记录在计算图中，即梯度回传到副本时也会传到源 `Tensor` 。**

另外一个常用的函数就是 `item()` ，它可以将一个标量 `Tensor` 转换成一个Python number:

```
x = torch.randn(1)
print(x)
print(x.item())
```

输出:

```
tensor([2.3466])
2.3466382026672363
```

线性代数

另外，PyTorch还支持一些线性函数，这里提一下，免得用起来的时候自己造轮子，具体用法参考官方文档。如下表所示:

函数
trace

函数
diag
triu/tril
mm/bmm
addmm/addbmm/addmv/addr/baddbmm..
t
dot/cross
inverse
svd

PyTorch中的 `Tensor` 支持超过一百种操作，包括转置、索引、切片、数学运算、线性代数、随机数等等，可参考**官方文档**。

2.2.3 广播机制

前面我们看到如何对两个形状相同的 `Tensor` 做按元素运算。当对两个形状不同的 `Tensor` 按元素运算时，可能会触发广播（broadcasting）机制：先适当复制元素使这两个 `Tensor` 形状相同后再按元素运算。

例如：

```
x = torch.arange(1, 3).view(1, 2)
print(x)
y = torch.arange(1, 4).view(3, 1)
print(y)
print(x + y)
```

输出：

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

由于 `x` 和 `y` 分别是1行2列和3行1列的矩阵，如果要计算 `x + y` ，那么 `x` 中第一行的2个元素被广播（复制）到了第二行和第三行，而 `y` 中第一列的3个元素被广播（复制）到了第二列。如此，就可以对2个3行2列的矩阵按元素相加。

2.2.4 运算的内存开销

前面说了，索引操作是不会开辟新内存的，而像 `y = x + y` 这样的运算是会新开内存的，然后将 `y` 指向新内存。为了演示这一点，我们可以使用Python自带的 `id` 函数：如果两个实例的ID一致，那么它们所对应的内存地址相同；反之则不同。

```
x = torch.tensor([1, 2])
y = torch.tensor([3, 4])
id_before = id(y)
y = y + x
print(id(y) == id_before) # False
```

如果想指定结果到原来的 `y` 的内存，我们可以使用前面介绍的索引来进行替换操作。在下面的例子中，我们把 `x + y` 的结果通过 `[:]` 写进 `y` 对应的内存中。

```
x = torch.tensor([1, 2])
y = torch.tensor([3, 4])
id_before = id(y)
y[:] = y + x
print(id(y) == id_before) # True
```

我们还可以使用运算符全名函数中的 `out` 参数或者自加运算符 `+=` (也即 `add_()`)达到上述效果，例如 `torch.add(x, y, out=y)` 和 `y += x` (`y.add_(x)`)。

```
x = torch.tensor([1, 2])
y = torch.tensor([3, 4])
id_before = id(y)
torch.add(x, y, out=y) # y += x, y.add_(x)
print(id(y) == id_before) # True
```

> **注：虽然 `view` 返回的 `Tensor` 与源 `Tensor` 是共享 `data` 的，但是依然是一个新的 `Tensor` （因为 `Tensor` 除了包含 `data` 外还有一些其他属性），二者id（内存地址）并不一致。**

2.2.5 `Tensor` 和NumPy相互转换

我们很容易用 `numpy()` 和 `from_numpy()` 将 `Tensor` 和NumPy中的数组相互转换。但是需要注意的一点是： **这两个函数所产生的的 `Tensor` 和NumPy中的数组共享相同的内存（所以他们之间的转换很快），改变其中一个时另一个也会改变！！！**

> **还有一个常用的将NumPy中的array转换成 `Tensor` 的方法就是 `torch.tensor()` ，需要注意的是，此方法总是会进行数据拷贝（就会消耗更多的时间和空间），所以返回的 `Tensor` 和原来的数据不再共享内存。**

`Tensor` 转NumPy

使用 `numpy()` 将 `Tensor` 转换成NumPy数组:

```
a = torch.ones(5)
b = a.numpy()
print(a, b)

a += 1
print(a, b)
b += 1
print(a, b)
```

输出：

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

NumPy数组转 Tensor

使用 `from_numpy()` 将NumPy数组转换成 `Tensor` ：

```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
print(a, b)

a += 1
print(a, b)
b += 1
print(a, b)
```

输出：

```
[1. 1. 1. 1. 1.] tensor([1., 1., 1., 1., 1.], dtype=torch.float64)
[2. 2. 2. 2. 2.] tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
[3. 3. 3. 3. 3.] tensor([3., 3., 3., 3., 3.], dtype=torch.float64)
```

所有在CPU上的 `Tensor` （除了 `CharTensor` ）都支持与NumPy数组相互转换。

此外上面提到还有一个常用的方法就是直接用 `torch.tensor()` 将NumPy数组转换成 `Tensor` ，需要注意的是该方法总是会进行数据拷贝，返回的 `Tensor` 和原来的数据不再共享内存。

```
c = torch.tensor(a)
a += 1
print(a, c)
```

输出

```
[4. 4. 4. 4. 4.] tensor([3., 3., 3., 3., 3.], dtype=torch.float64)
```

2.2.6 `Tensor` on GPU

用方法 `to()` 可以将 `Tensor` 在CPU和GPU（需要硬件支持）之间相互移动。

```
# 以下代码只有在PyTorch GPU版本上才会执行
if torch.cuda.is_available():
    device = torch.device("cuda")          # GPU
    y = torch.ones_like(x, device=device)  # 直接创建一个在GPU上的Tensor
    x = x.to(device)                       # 等价于 .to("cuda")
    z = x + y
    print(z)
    print(z.to("cpu", torch.double))       # to()还可以同时更改数据类型
```