pytorch—维度变换

改变张量形状

view和reshape：两个方法是一样的。view是0.3版本之前的，reshape方法是为了同步numpy开发的端口。

改变形状之后，要保证整个tensor的size不变。

b = a.view(4,784)、b.view(4,28,28,1)　这里理论上来说没问题，但是数据有一个维度丢失，我们需要拿到数据原先的维度信息。

```
In [2]:
import torch

In [3]:
#x相当于设置了四张图片，后三个参数是图片信息
a = torch.rand(4,1,28,28)

In [6]:
a.shape

torch.Size([4, 1, 28, 28])

In [8]:
a.view(4,28*28)

tensor([[0.1149, 0.4814, 0.4760,  ..., 0.8065, 0.9175, 0.3325],
        [0.5728, 0.6259, 0.4708,  ..., 0.2278, 0.3724, 0.3553],
        [0.5373, 0.7672, 0.5959,  ..., 0.9618, 0.0834, 0.2243],
        [0.4066, 0.2226, 0.5762,  ..., 0.6660, 0.6427, 0.9866]])

In [13]:
a.reshape(4*28,28).shape

torch.Size([112, 28])

In [14]:
b = a.view(4,784)

In [15]:
b.view(4,28,28,1)
```

没有将数据的size保持住，所以会报错

```
In [17]:
#没有将数据的size保持住，所以会报错
a.view(4,783)

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-17-a68cdf9a2a6b> in <module>
----> 1 a.view(4,783)

RuntimeError: shape '[4, 783]' is invalid for input of size 3136
```

维度增加、减少

unsqueeze

torch.unsqueeze(dim)：新插入一个维度，这个维度是自己定义的，数据的插入不会改变数据本身，不会增加也不会减少数据；只是说给数据增加了一个组别，这个组别的意义是我们自己去定义的。

dim是第一个维度处增加。范围和目标张量有关。[-a.dim()-1,a.dim()+1). 超过维度会报错。

里面传入参数为0和正数的时候是在这个数之前插入，如果是负数则是在这个维度之后进行数据的插入。

```
In [18]:
1 a.shape

torch.Size([4, 1, 28, 28])

In [19]:
1 a.unsqueeze(0).shape

torch.Size([1, 4, 1, 28, 28])

In [20]:
1 a.unsqueeze(-1).shape

torch.Size([4, 1, 28, 28, 1])

In [21]:
1 a.unsqueeze(4).shape

torch.Size([4, 1, 28, 28, 1])

In [22]:
1 a.unsqueeze(-4).shape

torch.Size([4, 1, 1, 28, 28])

In [23]:
1 a.unsqueeze(-5).shape

torch.Size([1, 4, 1, 28, 28])

In [24]:
1 a.unsqueeze(5).shape

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-24-b54eab361a50> in <module>
----> 1 a.unsqueeze(5).shape

IndexError: Dimension out of range (expected to be in range of [-5, 4], but got 5)
```

以下例子反应了数据本身不会改变，只是理解数据的方式变了。

```
In [26]:
1 a = torch.tensor([1.2,2.3])

In [29]:
1 a.shape

torch.Size([2])

In [27]:
1 a.unsqueeze(-1)

tensor([[1.2000],
        [2.3000]])

In [28]:
1 a.unsqueeze(0)

tensor([[1.2000, 2.3000]])
```

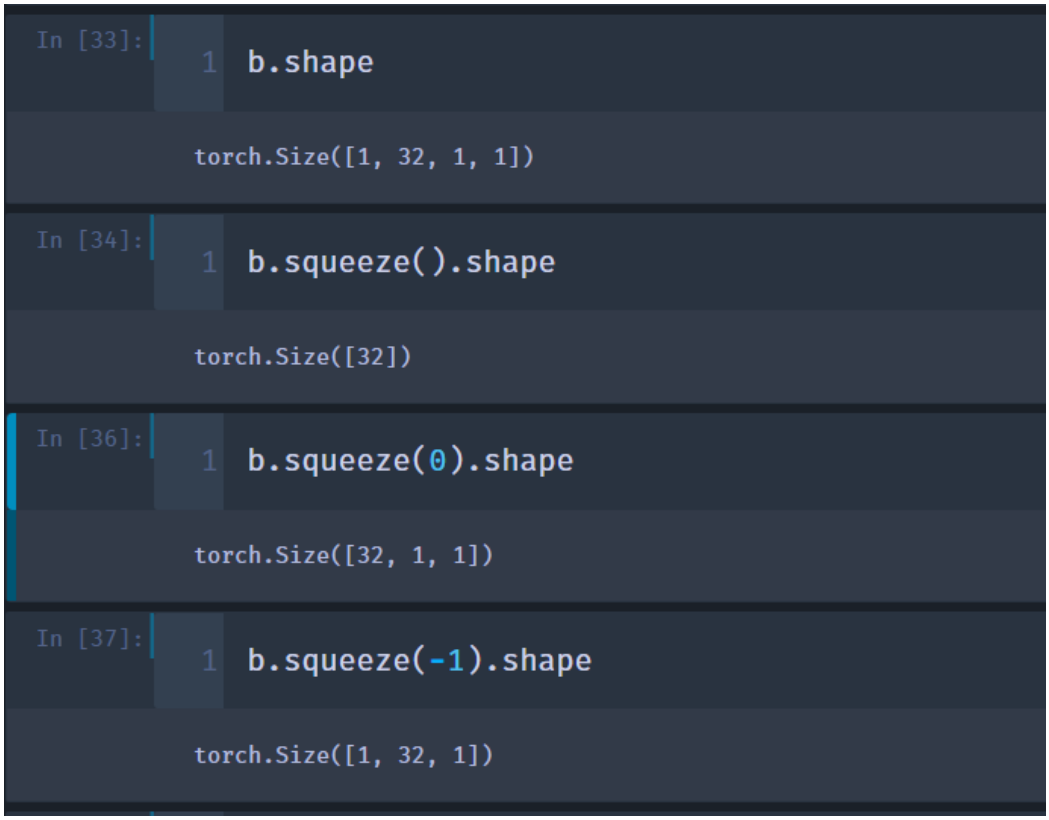
案例：

通过维度变换之后可以进行其他操作。

```
In [30]:
1 b = torch.rand(32)

In [31]:
1 f = torch.rand(4,32,14,14)

In [32]:
1 b = b.unsqueeze(1).unsqueeze(2).unsqueeze(0)

In [33]:
1 b.shape

torch.Size([1, 32, 1, 1])
```

squeeze

squeeze(input,dim) ：如果不给参数，那么就会将能挤压的全部进行挤压。如果给出了具体的数字，那么只会在相应的维度进行挤压。



维度扩展

维度扩展是指将那个维度上的shape改变掉。

expand

只是改变了我们的理解方式，并没有改变数据，只是一种view。不会主动复制数据，只会在有需要的时候复制数据。实现速度快，节约内存，推荐使用。

仅仅限于原来维度上的数据是1，变为之后的n才是可行的。如果原来的数据不为1，那么就不能进行操作。

如果输入的参数是-1，那么系统会默认给参数，这个用于有时候懒得去计算，让系统自己添加。这里有个bug，如果输入的参数是其他负数，生成的结果也是该负数，但是这是没有意义的。

```
In [39]:
1 b.shape

torch.Size([1, 32, 1, 1])

In [40]:
1 b.expand(4,32,14,14).shape

torch.Size([4, 32, 14, 14])

In [46]:
1 b.expand(-1,32,-1,-1).shape

torch.Size([1, 32, 1, 1])

In [47]:
1 b.expand(-1,32,-1,-4).shape

torch.Size([1, 32, 1, -4])
```

repeat

实实在在的增加了数据。将原张量横向、纵向地复制。

```
In [48]:
1 b.shape

torch.Size([1, 32, 1, 1])

In [50]:
1 #里面的参数是每个维度重复的次数
2 b.repeat(4,32,1,1).shape

torch.Size([4, 1024, 1, 1])

In [51]:
1 b.repeat(4,1,1,1).shape

torch.Size([4, 32, 1, 1])

In [52]:
1 b.repeat(4,1,32,32).shape

torch.Size([4, 32, 32, 32])
```

转置

.t

.t ：对矩阵进转置操作。只能对2D的矩阵进行操作，对于高维的矩阵不能进行该操作。

transpose

transpose(input, dim0, dim1) ： input是张量，后面两个参数是需要调换位置的维度。

调换维度之后会打乱原先的顺序，要是数据连续，需要使用contiguous方法来使数据连续。

维度可以进行交换，但是我们需要人为记住并跟踪维度信息。否则就会出现数据的污染。

```
In [64]:
a = torch.rand(4,3,32,32)

In [65]:
a.shape

torch.Size([4, 3, 32, 32])

In [76]:
a1 = a.transpose(1,3).contiguous().view(4,3*32*32).view(4,3,32,32)

In [77]:
a2 = a.transpose(1,3).contiguous().view(4,3*32*32).view(4,32,32,3).transpose(1,3)

In [78]:
a1.shape,a2.shape

(torch.Size([4, 3, 32, 32]), torch.Size([4, 3, 32, 32]))

In [79]:
torch.all(torch.eq(a,a1))

tensor(False)

In [80]:
torch.all(torch.eq(a,a2))

tensor(True)
```

permute

transpose可以进行任意两个维度的变化。但是当我们需要变化的维度较多的时候，那么我们可以使用permute方法。

同样，permute也会涉及到将内存打乱的问题，因此，我们有时候也会需要使用contiguous函数保持内存的连续。

```
In [85]:
b = torch.rand(4,3,28,32)
#我们希望得到（4,28,32,3）这样的样式

In [87]:
#第一种方式
b.transpose(1,3).shape

torch.Size([4, 32, 28, 3])

In [90]:
b.transpose(1,3).transpose(1,2).shape

torch.Size([4, 28, 32, 3])

In [91]:
#第二种方式
b.permute(0,2,3,1).shape

torch.Size([4, 28, 32, 3])
```

broadcasting

官方文档：https://pytorch.org/docs/stable/notes/broadcasting.html#broadcasting-semantics

能够扩展维度。从末尾开始连续找第一个不相同的维度容量开始扩充，直到扩充成维度相同。

需要满足两个条件：

- 每个tensor至少是一维的
- 两个tensor的维数从后往前，对应的位置要么是相等的，要么其中一个是1，或者不存在

如果`x`和`y`是broadcastable的，那么结果的tensor的size按照如下的规则计算

- 如果两者的维度不一样，那么就自动增加1维（也就是unsqueeze）
- 对于结果的每个维度，它取x和y在那一维上的最大值

利用broadcast可以省略内存，减少内存消耗。