# **Pytorch教程三-张量的基本操作**

张量是Torch中最为基本的操作原子，基本上所有的运算和操作都是基于torch的，因此关于tensor的操作也就十分的多，比如获取改编张量的形状，获取或改变张量中的元素，张量的四则运算等等。本次文章不穷举所有的操作，主要先讲述一些基本的操作之后，在举例几个我们常用的操作，Let’s Go!

我们先创建一个张量：  
a = torch.tensor([[1,2,3],[4,5,6],[7,8,9]],dtype=torch.float)  
b = torch.ones\_like(a)  
print(a)  
print(b)

# 输出

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

[4., 5., 6.],

[7., 8., 9.]])

tensor([[1., 1., 1.],

[1., 1., 1.],

[1., 1., 1.]])

**一、形状相关的操作**

**（1）shape/resize**

获取和改变张量的形状的的需求在实际的深度学习项目中经常使用，Torch提供了针对不同张量提供了不同的函数和方法用于获取和改变张量的形状。我们来看一下以下代码的输出：

print(a.shape)  
print(a.shape[0])  
print(a.resize(1,9))  
print(a.resize(9,1))  
print(a)

print(a.numpy())

# 输出

torch.Size([3, 3])

3

tensor([[1., 2., 3., 4., 5., 6., 7., 8., 9.]])

tensor([[1.],

[2.],

[3.],

[4.],

[5.],

[6.],

[7.],

[8.],

[9.]])

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

[4., 5., 6.],

[7., 8., 9.]])

[[1. 2. 3.]

[4. 5. 6.]

[7. 8. 9.]]

可以知道，使用torch.shape即可获取到张量的形状，通过向resize()方法传递一个数组就可以改变张量的形状，不改变原始张量，另外我们可以使用numpy()可以将tensor转成numpy数据。

**（2）resize\_()/reshape()/resize\_as()/dim()**

但是下划线的**resize\_()**方法是改变原始张量的形状的，其他下划线的方法也会改变原始张量如**cos\_(),sin\_()**等等，也就是说在PyTorch中以“\_”结尾的函数的效果是对tensot进行原地的操作（改变），函数没有返回值，大家稍微注意一下就行。

除了resize()方法，reshape()方法也是可以的：

c = torch.reshape(a,shape=(1,9))  
print(c)  
print(a)

# 输出

tensor([[1., 2., 3., 4., 5., 6., 7., 8., 9.]])

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

[4., 5., 6.],

[7., 8., 9.]])

另外，使用**a.resize\_as(b)**可以将张量a的形状设置成与张量B一致的形状（a的形状来源于b）。使用**dim()**函数可以获取张量的维度，比如**a.dim()**的结果为2。

**（3）view()/permute()/flatten()**

View()函数的功能与numpy中的reshape()函数是有相似之处的。其作用是将tensor中的数据按照行顺序转变为一维的张量：

print(a.view(1,9))  
print(a.view(3,3))

# 输出

tensor([[1., 2., 3., 4., 5., 6., 7., 8., 9.]])

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

[4., 5., 6.],

[7., 8., 9.]])

可以看出view(3,3)的顺序是按照行顺序进行的，那么想按照列的顺序进行填充则只能使用permute()函数了：

print(a.permute(1,0))

# 输出

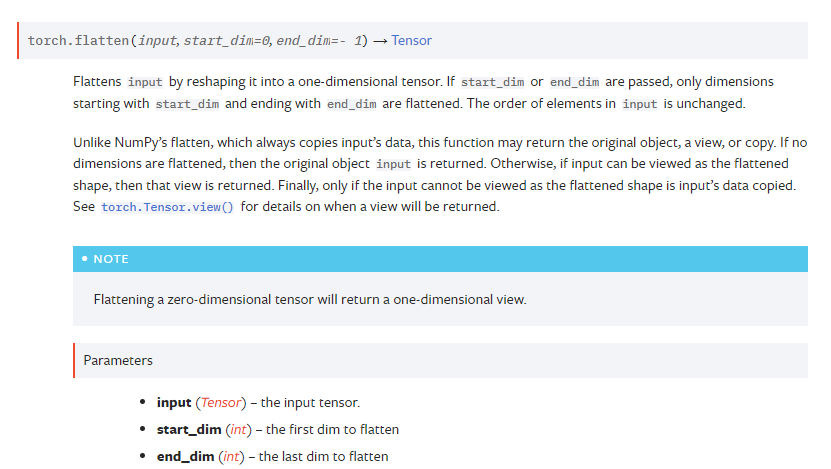
tensor([[1., 4., 7.],

[2., 5., 8.],

[3., 6., 9.]])

可以看出这个这好比给张量进行了转置的操作，参数(1,0)就是互换张量的两个维度。

flatten()顾名思义也是打平张量的意思，函数原型和介绍如下所示：



这个flatten()与numpy的flatten还是有一些区别的。

import numpy as np  
t = np.arange(18)  
t = np.reshape(t, (3,3,2))  
t = torch.from\_numpy(t)  
print(t)  
print(torch.flatten(t))  
print(torch.flatten(t,start\_dim=0, end\_dim=1))  
print(torch.flatten(t,start\_dim=1, end\_dim=2))

# 输出

tensor([[[ 0, 1],

[ 2, 3],

[ 4, 5]],

[[ 6, 7],

[ 8, 9],

[10, 11]],

[[12, 13],

[14, 15],

[16, 17]]], dtype=torch.int32)

tensor([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17],

dtype=torch.int32)

tensor([[ 0, 1],

[ 2, 3],

[ 4, 5],

[ 6, 7],

[ 8, 9],

[10, 11],

[12, 13],

[14, 15],

[16, 17]], dtype=torch.int32)

tensor([[ 0, 1, 2, 3, 4, 5],

[ 6, 7, 8, 9, 10, 11],

[12, 13, 14, 15, 16, 17]], dtype=torch.int32)

可以看出，使用flatten()函数也能进行平铺我们的张量。

**（4）unsqueeze()/squeeze()**

张量的维度可以是多维的，在实际的编码中我们需要给已经存在的张量进行维度的提升，比如二维提升为三维，那么该怎么进行操作呢？我们可以使用**torch.unsqueeze()**方法，具体如下：

d1 = torch.unsqueeze(input=a,dim=0)

print(a)  
print(d1)  
print(d1.size())  
print(d1.dim())

# 输出

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

[4., 5., 6.],

[7., 8., 9.]])

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

[4., 5., 6.],

[7., 8., 9.]]])

torch.Size([1, 3, 3])

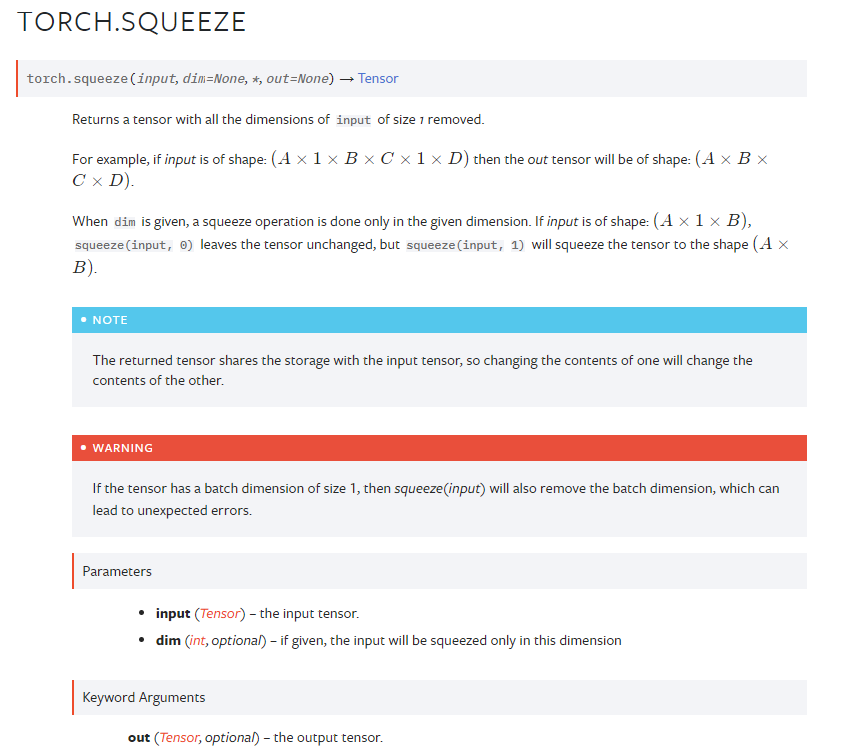
3

上述代码显示我们在张量a的dim=0的地方新增了一个维度，这样原来的张量维度从(3,3)转变成了(1,3,3)，这个理解起来还是很简单的。

相应的我们可以使用**torch.squeeze()**来降低维度，该函数的原型为：

torch.squeeze(input, dim=None, out=None)

官网中的介绍如下：



翻译一下语句就是：

（1） 如果输入是形如(A×1×B×1×C×1×D)，那么输出形状就为： (A×B×C×D)

（2）当给定dim时，操作只在给定维度上。如果输入 (A×1×B)使用squeeze(input, 0)则结果保持张量不变，如果是squeeze(input, 1)，结果形状会变成 (A×B)。

import numpy as np  
t = np.arange(8)  
t = np.reshape(t, (4,1,2))  
t = torch.from\_numpy(t)  
print(t)  
print(torch.squeeze(input=t,dim=1))  
print(torch.squeeze(input=t,dim=0))

# 输出如下：

tensor([[[0, 1]],

[[2, 3]],

[[4, 5]],

[[6, 7]]], dtype=torch.int32)

tensor([[0, 1],

[2, 3],

[4, 5],

[6, 7]], dtype=torch.int32)

tensor([[[0, 1]],

[[2, 3]],

[[4, 5]],

[[6, 7]]], dtype=torch.int32)

大家试一试在dim参数不一样的时候，输出是怎样的，这里就不赘述了哈，好好理解即可。

**二、获取张量的元素**

张量中元素的获取也比较常见也比较重要，接下来我们来看几个常见获取元素的方式：

1. **通过切片和索引的方式**

我们知道在numpy和pandas中我们可以很简单的使用切片和索引来获取元素，PyTorch使用切片和索引的方式与它们类似，先创建一个三维的张量：

import torch  
mytensor = torch.arange(36).reshape(3,3,4)  
print(mytensor)

# 输出

tensor([[[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]],

[[12, 13, 14, 15],

[16, 17, 18, 19],

[20, 21, 22, 23]],

[[24, 25, 26, 27],

[28, 29, 30, 31],

[32, 33, 34, 35]]])

那么依次获取三个张量的方式就可以如下所示：

print(mytensor[0])  
print(mytensor[1])  
print(mytensor[2])

这里就不在打印输出的结果了，可以看出使用索引的方式索引是从0开始计数的。那么进一步的获取张量中的元素该怎么操作呢？请看演示：

*# 获取第二个维度张量的前两行元素*print(mytensor[1][0:2])  
print('\_\_\_\_\_\_\_\_\_\_\_\_\_\_')  
print(mytensor[1,0:2])  
print('\_\_\_\_\_\_\_\_\_\_\_\_\_\_')  
print(mytensor[1,0:2,:])

# 输出

tensor([[12, 13, 14, 15],

[16, 17, 18, 19]])

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

tensor([[12, 13, 14, 15],

[16, 17, 18, 19]])

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

tensor([[12, 13, 14, 15],

[16, 17, 18, 19]])

可以看出上述三个代码的效果是一致的，也还可以这样开获取：mytensor[1][1][1]。

如果需要获取其中的数值，我们可以将其装换为ndarray，即可以这样使用：

print(mytensor[1,0:2,:].numpy())

进一步的我们可以负数索引来获取元素，我们来看两个简单的例子：

print(mytensor[-1])  
print(mytensor[1][-1][-3:-1])

# 输出

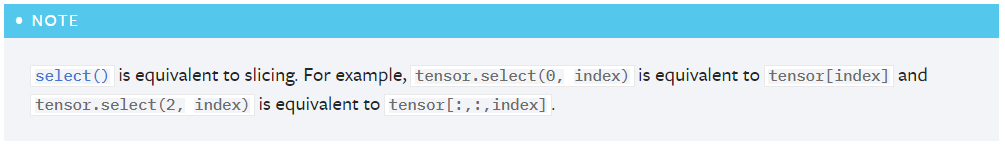
tensor([[24, 25, 26, 27],

[28, 29, 30, 31],

[32, 33, 34, 35]])

tensor([21, 22])

理解起来还是很简单的，这里我就不再解释了。这里插入一点，在PyTorch中有个函数即**torch.select()**的作用与张量切片的效果是一样的，集体的介绍如下：



1. **通过条件筛选**

条件筛选的方式在pandas获取数据的时候经常使用，在PyTorch中我们也可使用条件筛选来获取我们想要的元素。

t1 = torch.arange(12).reshape(3,4)  
t2 = torch.arange(12).reshape(3,4) + 100  
print(t1)  
print(t2)  
print(t1[t1>2])

# 输出

tensor([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

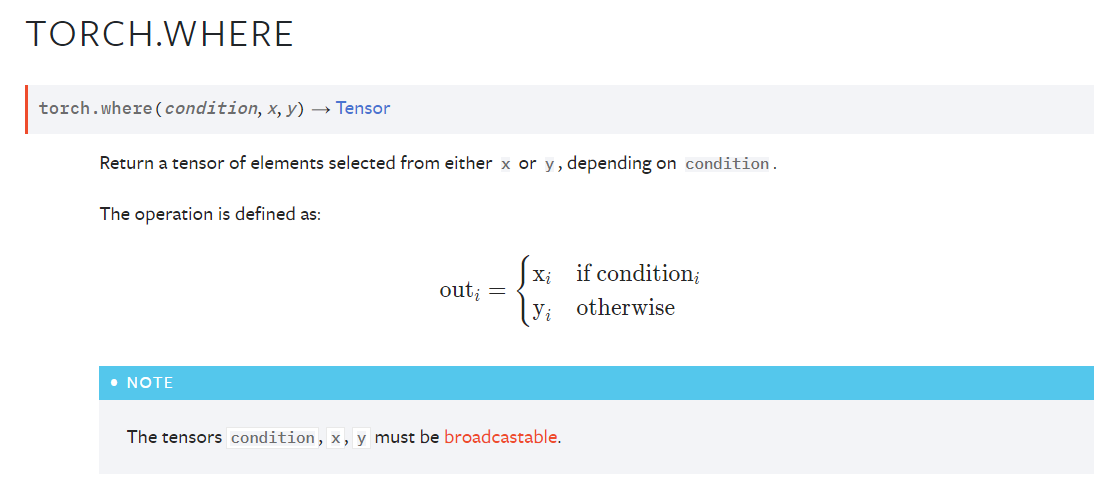
tensor([[100, 101, 102, 103],

[104, 105, 106, 107],

[108, 109, 110, 111]])

tensor([ 3, 4, 5, 6, 7, 8, 9, 10, 11])

可以看出使用这种条件筛选的方式可以获得一个dim为1的张量。使用torch.where()函数也能进行张量元素的提取：



也就是说我们可以向函数**where()**中的condition条件设置一些值，来进行元素的提取，我们来看一个案例：

print(torch.where(t1>6,t1,t2))

print(torch.where(t1>6,t1,0))

# 输出

tensor([[100, 101, 102, 103],

[104, 105, 106, 7],

[ 8, 9, 10, 11]])

tensor([[ 0, 0, 0, 0],

[ 0, 0, 0, 7],

[ 8, 9, 10, 11]])

可以看出我们可以使用int型的数据0来作为默认值进行填充。

1. **通过函数**

在学习线性代数的时候，经常遇见到提取一个矩阵上对角线元素和下对角线元素的需求，在PyTorch也提供了一些函数供我们使用：

a = torch.arange(24).reshape(4,6)  
b1 = torch.tril(a,diagonal=0)  
b2 = torch.tril(a,diagonal=2)  
b3 = torch.tril(a,diagonal=-1)  
print(a)  
print(b1)  
print(b2)  
print(b3)

# 输出

tensor([[ 0, 1, 2, 3, 4, 5],

[ 6, 7, 8, 9, 10, 11],

[12, 13, 14, 15, 16, 17],

[18, 19, 20, 21, 22, 23]])

tensor([[ 0, 0, 0, 0, 0, 0],

[ 6, 7, 0, 0, 0, 0],

[12, 13, 14, 0, 0, 0],

[18, 19, 20, 21, 0, 0]])

tensor([[ 0, 1, 2, 0, 0, 0],

[ 6, 7, 8, 9, 0, 0],

[12, 13, 14, 15, 16, 0],

[18, 19, 20, 21, 22, 23]])

tensor([[ 0, 0, 0, 0, 0, 0],

[ 6, 0, 0, 0, 0, 0],

[12, 13, 0, 0, 0, 0],

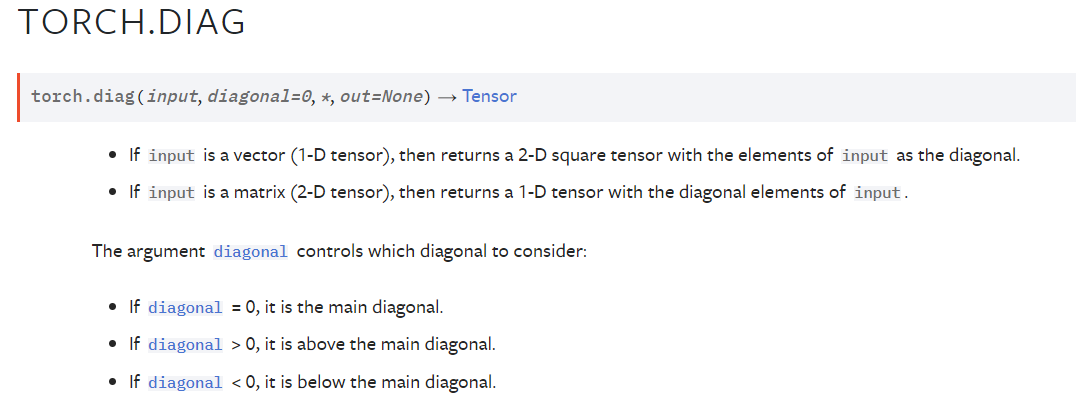
[18, 19, 20, 0, 0, 0]])

**官网中介绍到参数diagonal用来控制对角线。**

1. 如果diagonal = 0，保留对角线及其上面的元素将被保留；
2. 如果diagonal > 0，保留与主对角线相同数量的对角线。
3. 如果diagonal < 0，排除与主对角线下方相同数量的对角线。

与上对角线对应的是下对角线元素的获取，在Torch中使用的是**torch.triu()**函数。这里就不在赘述了其与**torch.tril()**类似。

**Torch.diag()**也能产生我们需要的张量，函数的说明如下，大家自己实验一下：



**三、[张量的拼接与拆分](https://zhuanlan.zhihu.com/write)**

有时候我们需要进行张量的拼接，就好比我们需要拼接两个dataframe一样。Torch也提供了较多的方法供我们使用：

a = torch.arange(12).reshape(3,4)  
b = torch.rand(3,4)  
c= torch.cat((a,b),dim=0)  
d= torch.cat((a,b),dim=1)  
print(c)  
print(d)

# 输出

tensor([[ 0.0000, 1.0000, 2.0000, 3.0000],

[ 4.0000, 5.0000, 6.0000, 7.0000],

[ 8.0000, 9.0000, 10.0000, 11.0000],

[ 0.7166, 0.7443, 0.7123, 0.4425],

[ 0.4828, 0.9462, 0.6050, 0.5358],

[ 0.4135, 0.3879, 0.2861, 0.1595]])

tensor([[ 0.0000, 1.0000, 2.0000, 3.0000, 0.7166, 0.7443, 0.7123, 0.4425],

[ 4.0000, 5.0000, 6.0000, 7.0000, 0.4828, 0.9462, 0.6050, 0.5358],

[ 8.0000, 9.0000, 10.0000, 11.0000, 0.4135, 0.3879, 0.2861, 0.1595]])

可以看出我们可以再横轴和纵轴两个方向进行拼接。注意**torch.cat((a,b),dim=0)和torch.cat((b,a),dim=0)**的结果是不一样的，大家自己想象一下结果。

另外stack()函数也能进行拼接：

e = torch.stack((a,b),dim=0)  
f = torch.stack((a,b),dim=1)  
print(e)  
print(f)

# 输出

tensor([[[0.0000e+00, 1.0000e+00, 2.0000e+00, 3.0000e+00],

[4.0000e+00, 5.0000e+00, 6.0000e+00, 7.0000e+00],

[8.0000e+00, 9.0000e+00, 1.0000e+01, 1.1000e+01]],

[[4.4792e-01, 7.0716e-01, 3.5640e-01, 9.7867e-03],

[9.8460e-01, 5.3677e-01, 5.9004e-01, 9.8523e-01],

[8.1127e-01, 6.6594e-01, 8.4023e-01, 8.9863e-01]]])

tensor([[[0.0000e+00, 1.0000e+00, 2.0000e+00, 3.0000e+00],

[4.4792e-01, 7.0716e-01, 3.5640e-01, 9.7867e-03]],

[[4.0000e+00, 5.0000e+00, 6.0000e+00, 7.0000e+00],

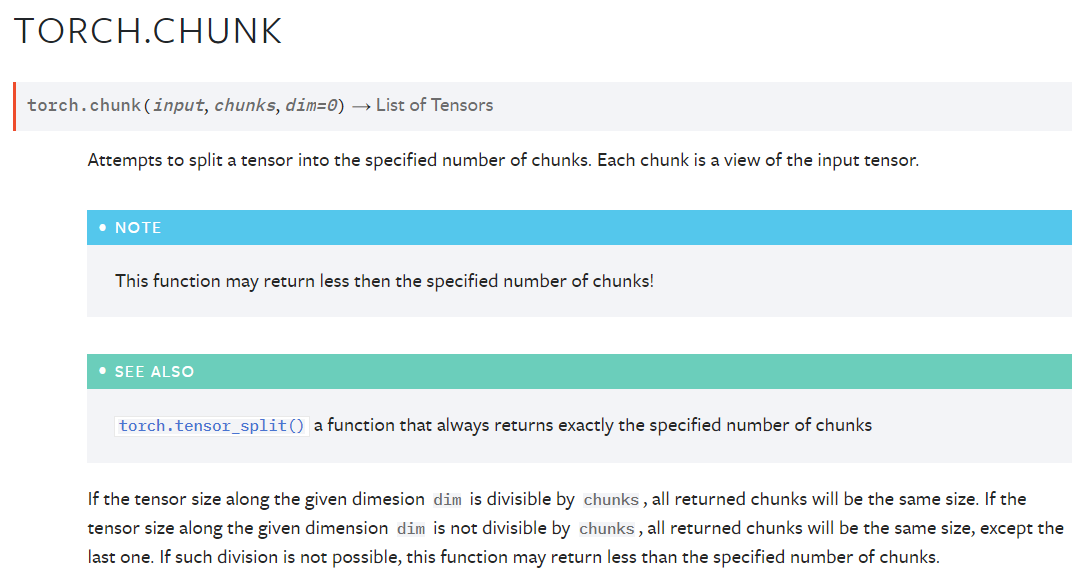
[9.8460e-01, 5.3677e-01, 5.9004e-01, 9.8523e-01]],

[[8.0000e+00, 9.0000e+00, 1.0000e+01, 1.1000e+01],

[8.1127e-01, 6.6594e-01, 8.4023e-01, 8.9863e-01]]])

可以看出这个**stack()和cat()**的结果还是有一定差别的，**stcak()**有一种堆叠的感觉。

那么对于张量的拆分Torch也提供了函数**torch.chunk()和torch.split()**:



上述是**chunk()**函数的介绍，这个函数返回的结果是张量的列表：

a = torch.arange(12).reshape(3,4)  
print(torch.chunk(a,chunks=2,dim = 0))

# 输出

(tensor([[0, 1, 2, 3],

[4, 5, 6, 7]]), tensor([[ 8, 9, 10, 11]]))

也就是说将原张量拆为了两个张量，以列表形式返回。相应的我们可以从列方向进行拆分：

torch.chunk(a,chunks=2,dim = 1)

**torch.split()**函数也能将原始张量进行拆开：

a = torch.arange(12).reshape(3,4)

res1,res2,res3 = torch.split(a,(1,1,1),dim = 0)  
print(res1)  
print(res2)  
print(res3)

# 输出

tensor([[0, 1, 2, 3]])

tensor([[4, 5, 6, 7]])

tensor([[ 8, 9, 10, 11]])

这里要注意1+1+1=3,也就是拆开的个数要与原始的张量的相应维度上一致。

**troch.tensor\_split()**也能进行拆分，我们来看例子：

print(torch.tensor\_split(a, 3))  
print(torch.tensor\_split(a, 3, dim=1))

# 输出

(tensor([[0, 1, 2, 3]]), tensor([[4, 5, 6, 7]]), tensor([[ 8, 9, 10, 11]]))

(tensor([[0, 1],

[4, 5],

[8, 9]]),

tensor([[ 2],

[ 6],

[10]]),

tensor([[ 3],

[ 7],

[11]]))

总结一下：

（1）torch.chunk()可以将张量分割为特定数量的块。  
（2）torch.split()可将张量分割为特定数量的块时，指定块大小。

（3）torch.tensor\_split()可将张量分割为特定数量的块时，也可指定块大小。

其他关于张量的拼接和拆分的案例我们在实际的工程项目中遇到的时候再和大家分享。

**四、总结**

以上就是本次的文章的全部内容，下次文章我们来看一下Torch的张量其他相关的知识。