Lesson 2.张量的索引、分片、合并以及维度调整

张量作为有序的序列，也是具备数值索引的功能，并且基本索引方法和Python原生的列表、NumPy中的数组基本一致，当然，所有不同的是，PyTorch中还定义了一种采用函数来进行索引的方式。

而作为PyTorch中基本数据类型，张量即具备了列表、数组的基本功能，同时还充当着向量、矩阵、甚至是数据框等重要数据结构，因此PyTorch中也设置了非常完备的张量合并与变换的操作。

In [1]:
```
import torch
import numpy as np
```

一、张量的符号索引（即数据下标索引）

张量也是有序序列，我们可以根据每个元素在系统内的顺序“编号”，来找出特定的元素，也就是索引。

1.一维张量索引

一维张量的索引过程和Python原生对象类型的索引一致，基本格式遵循$[\star t:end:step]$，表示从start开始，到end之前，注意是左闭右开，索引的基本要点回顾如下。

In [2]:
```
t1 = torch.arange(1, 11)
t1
```

Out[2]:
```
tensor([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

- 从左到右，下标从零开始
  - 注意输出是0维张量，而不是整数

In [3]:
```
t1[0]
```

Out[3]:
```
tensor(1)
```

> 注：张量索引出来的结果还是零维张量， 而不是单独的数。因为一维张量由零维张量构成。要转化成单独的数，需要使用item()方法。

- 冒号分隔，表示对某个区域进行索引，也就是所谓的切片
  - 基本格式遵循$[\star t:end:step]$，最后一个数是表示间隔，表示从第strat+1个元素开始，到第end+1个元素之前，每隔step-1取一个元素
  - 注意：冒号左包含右不包含

In [4]:
```
t1[1: 8]              # 索引其中2-9号元素，并且左包含右不包含
```

Out[4]:
```
tensor([2, 3, 4, 5, 6, 7, 8])
```

- 第二个冒号，表示索引的间隔

In [5]:
```
t1[1: 8: 2]           # 索引其中2-9号元素，左包含右不包含，且隔2-1个数取一个
```

Out[5]:
```
tensor([2, 4, 6, 8])
```

- 冒号前后没有值，表示索引这个区域

In [6]:
```
t1[1: : 2]            # 从第二个元素开始索引，一直到结尾，并且隔2-1个数取一个
```

Out[6]:
```
tensor([ 2,  4,  6,  8, 10])
```

In [7]:
```
t1[: 8: 2]            # 从第一个元素开始索引到第9个元素（不包含），并且隔2-1个数取一个
```

Out[7]:
```
tensor([1, 3, 5, 7])
```

> 在Python原生数据类型中，step可以小于0，从而实现从后往前取

In [12]:
```
a = list(range(1,11))
a
```

Out[12]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [14]:
```
a[9: 1: -1] #从第9+1个元素开始，到第1+1个元素之前，间隔1-1个元素取值，因此不包含原列表中
```

Out[14]:
```
[10, 9, 8, 7, 6, 5, 4, 3]
```

> 在张量的索引中，step位必须大于0

In [15]:
```
t1[9: 1: -1]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-15-b82bb967c8e2> in <module>
----> 1 t1[9: 1: -1]

ValueError: step must be greater than zero
```

2.二维张量索引

二维张量的索引逻辑和一维张量的索引逻辑基本相同，二维张量可以视为两个一维张量组合而成，而在实际的索引过程中，需要用逗号进行分隔，分别表示对哪个一维张量进行索引、以及具体的一维张量的索引。

In [16]:
```
# 利用reshape创建二维张量
t2 = torch.arange(1, 10).reshape(3, 3)
t2
```

Out[16]:
```
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
```

In [17]:
```
t2[0, 1]                    # 表示索引第0+1行、第1+1个（第1+1列的）元素
```

Out[17]:
```
tensor(2)
```

In [18]:
```
t2[0, ::2]                  # 表示索引第一行、每隔2-1个元素取一个
```

Out[18]:
```
tensor([1, 3])
```

In [19]:
```
t2[0, [0, 2]]               # 索引结果同上
```

Out[19]:
```
tensor([1, 3])
```

In [9]:
```
t2[::2, ::2]                # 表示每隔两行取一行、并且每一行中每隔两个元素取一个
```

Out[9]:
```
tensor([[1, 3],
        [7, 9]])
```

In [20]:
```
t2[[0, 2], 1]                # 索引第0+1行第1+1列和第2+1行第1+1列的元素（应该是两个元素
```

Out[20]:
```
tensor([2, 8])
```

理解：对二维张量来说，基本可以视为是对矩阵的索引，并且行、列的索引遵照相同的索引规范，并用逗号进行分隔。

3.三维张量的索引

在二维张量索引的基础上，三维张量拥有三个索引的维度。我们将三维张量视作矩阵组成的序列，则在实际索引过程中拥有三个维度，分别是索引矩阵、索引矩阵的行、索引矩阵的列。

In [21]:
```
t3 = torch.arange(1, 28).reshape(3, 3, 3)
t3
```

Out[21]:
```
tensor([[[ 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]]])
```

In [9]:
```
t3.shape
```

Out[9]:
```
torch.Size([3, 3, 3])
```

In [22]:
```
t3[1, 1, 1]           # 索引第1+1个矩阵中，第1+1行、第1+1个元素
```

Out[22]:
```
tensor(14)
```

In [14]:
```
t3[1, ::2, ::2]       # 索引第1+1个矩阵，行和列都是每隔2-1个取一个
```

Out[14]:
```
tensor([[10, 12],
        [16, 18]])
```

In [23]:
```
t3[:: 2, :: 2, :: 2]       # 每隔2-1个取一个矩阵，对于每个矩阵来说，行和列都是每隔2-1个
```

Out[23]:
```
tensor([[[ 1,  3],
         [ 7,  9]],

        [[19, 21],
         [25, 27]]])
```

理解：更为本质的角度去理解高维张量的索引，其实就是围绕张量的“形状”进行索引

- 第一个索引，对应在第一个维度上挑选
- 第二个索引，对应在第二个维度上挑选
- 以此类推

In [24]:
```
t3.shape
```

Out[24]:
```
torch.Size([3, 3, 3])
```

In [25]:
```
t3[1, 1, 1]             # 与shape一一对应
```

Out[25]:
```
tensor(14)
```

二、张量的函数索引

在PyTorch中，我们还可以使用index_select函数，通过指定index来对张量进行索引。

torch.index_select(input, dim, index, *, out=None) → Tensor

- dim参数表示按照第dim+1个维度进行索引，比如dim=0表示按照第1个维度索引
- index表示索引位置，如果是张量，则索引位置就是张量中的每一个元素

1. 一维张量的函数索引

In [27]:
```
t1
```

Out[27]:
```
tensor([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [28]:
```
t1.ndim
```

Out[28]:
```
1
```

In [29]:
```
indices = torch.tensor([1, 2])
indices
```

Out[29]:
```
tensor([1, 2])
```

In [30]:
```
t1[1: 3]    # 切片法索引
```

Out[30]:
```
tensor([2, 3])
```

In [33]:
```
t1[[1, 2]]  # 列表法索引，引出的是第1+1和第2+1个元素
```

Out[33]:
```
tensor([2, 3])
```

In [34]:
```
torch.index_select(t1, 0, indices) # 函数索引，引出的是第1+1和第2+1个元素
```

Out[34]:
```
tensor([2, 3])
```

在index_select函数中，第二个参数实际上代表的是索引的维度。对于t1这个一维向量来说，由于只有一个维度，因此第二个参数取值为0，就代表在第一个维度上进行索引

2. 高维张量的函数索引

函数索引在很多模型中应用，因为在高维张量中更加清晰

In [41]:
```
t2 = torch.arange(12).reshape(4, 3)
t2
```

Out[41]:
```
tensor([[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]])
```

In [42]:
```
t2.shape
```

Out[42]:
```
torch.Size([4, 3])
```

In [44]:
```
indices = torch.arange(1, 3)
indices
```

Out[44]:
```
tensor([1, 2])
```

In [45]:
```
torch.index_select(t2, 0, indices)    #在第0+1个维度上取出索引为1+1和2+1的部分
```

Out[45]:
```
tensor([[3, 4, 5],
        [6, 7, 8]])
```

dim参数取值为0，代表在shape的第0+1个维度上索引

In [46]:
```
torch.index_select(t2, 1, indices)  #在第1+1个维度上取出索引为1+1和2+1的部分
```

Out[46]:
```
tensor([[ 1,  2],
        [ 4,  5],
        [ 7,  8],
        [10, 11]])
```

dim参数取值为1，代表在shape的第1+1个维度上索引

下面是一个案例

In [49]:
```
import torch
a=torch.arange(4*512*28*28).view(4,512,28,28)
```

```
index=np.random.choice(32,5)# 在0到31内随机选5个值
select=torch.index_select(a,1,index=torch.tensor(index,dtype=int))
print(index)
print(a.shape)
print(select.shape)
```

```
[ 3 18 19  0 24]
torch.Size([4, 512, 28, 28])
torch.Size([4, 5, 28, 28])
```

torch.index_select与直接索引的区别

In [52]:

```
import torch
a=torch.arange(40).view(2,4,5)
index=torch.tensor([1])
select_1=torch.index_select(a,dim=0,index=index)
select_2=a[0,:,:]
print(select_1)
print(select_2)
print(select_1.shape) #函数索引后还是3维张量
print(select_2.shape) #直接索引后变成2维张量
```

```
tensor([[[20, 21, 22, 23, 24],
         [25, 26, 27, 28, 29],
         [30, 31, 32, 33, 34],
         [35, 36, 37, 38, 39]]])
tensor([[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]])
torch.Size([1, 4, 5])
torch.Size([4, 5])
```

直接进行索引和利用torch.index_select索引最大的区别就在于：

- 直接进行索引数组维数会降低，利用torch.index_select索引数组维数不变

三、tensor.view()方法

在正式介绍张量的切分方法之前，需要首先介绍PyTorch中的.view()方法。该方法会返回一个类似视图的结果，该结果和原张量对象共享一块数据存储空间，并且通过.view()方法，还可以改变对象结构，生成一个不同结构，但共享一个存储空间的张量。当然，共享一个存储空间，也就代表二者是“浅拷贝”的关系，修改其中一个，另一个也会同步进行更改。

In [53]:

```
t = torch.arange(6).reshape(2, 3)
t
```

Out[53]:

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

In [56]:

```
te = t.view(3, 2)                # 构建一个数据相同，但形状不同的“视图”对象
te
```

Out[56]:

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

In [55]:

```
t
```

Out[55]:
```
tensor([[0, 1, 2],
        [3, 4, 5]])
```

> 二维张量也可以用一维赋值方法，如下面只给了行索引，而未给列索引，因此把第1行的所有元素都赋值为1

In [57]:
```
t[0] = 1                          # 对t进行修改
```

In [58]:
```
t
```

Out[58]:
```
tensor([[1, 1, 1],
        [3, 4, 5]])
```

In [59]:
```
te                                # te同步变化
```

Out[59]:
```
tensor([[1, 1],
        [1, 3],
        [4, 5]])
```

tensor.view()方法也可以用于升维或降维，但是注意view用于升维或降维时，不改变原张量的维度

In [61]:
```
tr = t.view(1, 2, 3)              # 维度也可以修改
tr
```

Out[61]:
```
tensor([[[1, 1, 1],
         [3, 4, 5]]])
```

“视图”的作用就是节省空间，而值得注意的是，在接下来介绍的很多切分张量的方法中，返回结果都是“视图”，而不是新生成一个对象。

- 视图的思想来源于数据库
- 视图使编程者可以改变、分片、合并张量，而不用占用新的空间，与基础数据类型如list等的分片不同，后者会生成新的对象，占用更多内存

四、张量的分片函数

1.分块：chunk函数

chunk函数能够按照某维度，对张量进行均匀切分，并且返回结果是原张量的视图。

In [63]:
```
t2 = torch.arange(12).reshape(4, 3)
t2
```

Out[63]:
```
tensor([[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]])
```

torch.chunk(input, chunks, dim=0) → List of Tensors

- input (Tensor) – the tensor to split

- chunks (int) – number of chunks to return，也就是要切成几块
- dim (int) – dimension along which to split the tensor，也就是在哪个维度上切

注意该函数返回的是一个元祖，其中的元素与原张量的维度相同，并没有产生降维

In [66]:
```
tc = torch.chunk(t2, 4, dim=0)            # 在第零个维度上（按行），进行四等分
tc
```

Out[66]:
```
(tensor([[0, 1, 2]]),
 tensor([[3, 4, 5]]),
 tensor([[6, 7, 8]]),
 tensor([[ 9, 10, 11]]))
```

注意：chunk返回结果是一个视图，不是新生成了一个对象，因此要时刻意识到分块结果的改变，会影响原张量

In [72]:
```
tc[0] # 返回元祖中的第一个值
```

Out[72]:
```
tensor([[1, 1, 2]])
```

In [73]:
```
tc[0][0] # 取出二维张量中的第一行
```

Out[73]:
```
tensor([1, 1, 2])
```

In [69]:
```
tc[0][0][0] = 1            # 修改tc中的值
```

In [70]:
```
tc
```

Out[70]:
```
(tensor([[1, 1, 2]]),
 tensor([[3, 4, 5]]),
 tensor([[6, 7, 8]]),
 tensor([[ 9, 10, 11]]))
```

In [71]:
```
t2                         # 原张量也会对应发生变化
```

Out[71]:
```
tensor([[ 1,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]])
```

当原张量不能均分时，chunk不会报错，而是会返回次一级均分的结果，即chunks-1等分，如果次一级还是不能均分，则继续向下找次一级等分，如果直到chunks=2也无法等分时，就会出现不均匀分片的结果，如原张量某一维度上是5，如果chunks设为4，则会生成3块，对应维度分别是2、2、1

In [99]:
```
torch.chunk(t2, 3, dim=0)            # 次一级均分结果
```

Out[99]:
```
(tensor([[1, 1, 2],
         [3, 4, 5]]),
```

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

In [103…
```
len(torch.chunk(t2, 3, dim=0))
```

Out[103… 2

In [100…
```
torch.chunk(t2, 5, dim=0)              # 次一级均分结果
```

Out[100…
```
(tensor([[1, 1, 2]]),
 tensor([[3, 4, 5]]),
 tensor([[6, 7, 8]]),
 tensor([[ 9, 10, 11]]))
```

进阶用法

In [77]:
```
x = torch.randn(4,5)
x
```

Out[77]:
```
tensor([[-0.0254,  0.4216,  1.7971,  0.6708, -1.0939],
        [ 0.1823, -0.4759,  1.3940,  0.6531,  0.3483],
        [ 0.2320,  1.5522,  0.5226, -1.7277,  0.5668],
        [ 0.2708,  1.2064,  0.5778,  1.3170,  0.1301]])
```

In [79]:
```
x.chunk(4,0)
```

Out[79]:
```
(tensor([[-0.0254,  0.4216,  1.7971,  0.6708, -1.0939]]),
 tensor([[ 0.1823, -0.4759,  1.3940,  0.6531,  0.3483]]),
 tensor([[ 0.2320,  1.5522,  0.5226, -1.7277,  0.5668]]),
 tensor([[0.2708, 1.2064, 0.5778, 1.3170, 0.1301]]))
```

enumerate 函数返回index和对应元素，组成元祖

In [78]:
```
for data in enumerate(x.chunk(4,0)): # 在第一维度切分4块
    print(data)
```

```
(0, tensor([[-0.0254,  0.4216,  1.7971,  0.6708, -1.0939]]))
(1, tensor([[ 0.1823, -0.4759,  1.3940,  0.6531,  0.3483]]))
(2, tensor([[ 0.2320,  1.5522,  0.5226, -1.7277,  0.5668]]))
(3, tensor([[0.2708, 1.2064, 0.5778, 1.3170, 0.1301]]))
```

In [82]:
```
for data in enumerate(x.chunk(4,1)): # 在第2维度切分4块
    print(data)                      # chunk很有可能生成不均匀的块
```

```
(0, tensor([[-0.0254,  0.4216],
        [ 0.1823, -0.4759],
        [ 0.2320,  1.5522],
        [ 0.2708,  1.2064]]))
(1, tensor([[ 1.7971,  0.6708],
        [ 1.3940,  0.6531],
        [ 0.5226, -1.7277],
        [ 0.5778,  1.3170]]))
(2, tensor([[-1.0939],
        [ 0.3483],
        [ 0.5668],
        [ 0.1301]]))
```

2.拆分：split函数

split既能进行均分，也能进行自定义切分。当然，需要注意的是，和chunk函数一样，split返

回结果也是view。

torch.split(tensor, split_size_or_sections, dim=0)

- tensor (Tensor) – tensor to split.
- split_size_or_sections (int) or (list(int)) – size of a single chunk or list of sizes for each chunk，输入一个数值时表示均分，传入列表则按照列表切分，但是要求列表元素之和等于原张量该维度上的数字
- dim (int) – dimension along which to split the tensor.

In [83]:
```
t2 = torch.arange(12).reshape(4, 3)
t2
```

Out[83]:
```
tensor([[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]])
```

In [109…
```
torch.split(t2, 2, 0)              # 第二个参数只输入一个数值时表示均分，第三个参数表示切
```

Out[109…
```
(tensor([[0, 1, 2],
         [3, 4, 5]]),
 tensor([[ 6,  7,  8],
         [ 9, 10, 11]]))
```

In [121…
```
torch.split(t2, [1, 3], 0)             # 第二个参数输入一个序列时，表示按照序列数值进行
```

Out[121…
```
(tensor([[0, 1, 2]]),
 tensor([[ 3,  4,  5],
         [ 6,  7,  8],
         [ 9, 10, 11]]))
```

注意，当第二个参数位输入一个序列时，序列的各数值的和必须等于对应维度下形状分量的取值。例如，上述代码中，是按照第一个维度进行切分，而t2总共有4行，因此序列的求和必须等于4，也就是1+3=4，而序列中每个分量的取值，则代表切块大小。

In [122…
```
torch.split(t2, [1, 1, 1, 1], 0)
```

Out[122…
```
(tensor([[0, 1, 2]]),
 tensor([[3, 4, 5]]),
 tensor([[6, 7, 8]]),
 tensor([[ 9, 10, 11]]))
```

In [123…
```
torch.split(t2, [1, 1, 2], 0)
```

Out[123…
```
(tensor([[0, 1, 2]]),
 tensor([[3, 4, 5]]),
 tensor([[ 6,  7,  8],
         [ 9, 10, 11]]))
```

In [85]:
```
ts = torch.split(t2, [1, 2], 1)
ts
```

Out[85]:
```
(tensor([[0],
         [3],
         [6],
```

```
        [9]]),
 tensor([[ 1,  2],
        [ 4,  5],
        [ 7,  8],
        [10, 11]]))
```

In [86]:
```
ts[0][0] # 取出第一个元祖的元素（是一个二维张量），然后取出二维张量的第一行
```

Out[86]:
```
tensor([0])
```

In [87]:
```
ts[0][0] = 1                    # view进行修改
```

In [88]:
```
ts
```

Out[88]:
```
(tensor([[1],
         [3],
         [6],
         [9]]),
 tensor([[ 1,  2],
         [ 4,  5],
         [ 7,  8],
         [10, 11]]))
```

In [89]:
```
t2                              # 原对象同步改变
```

Out[89]:
```
tensor([[ 1,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]])
```

> tensor的split方法和array的split方法有很大的区别，array的split方法array_split是根据索引进行切分，只能进行等分。

In [90]:
```
array = np.arange(9)
array
```

Out[90]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

In [91]:
```
np.array_split(array, 4)
```

Out[91]:
```
[array([0, 1, 2]), array([3, 4]), array([5, 6]), array([7, 8])]
```

五、张量的合并操作

张量的合并操作类似与列表的追加元素，可以拼接、也可以堆叠。

- 拼接函数：cat，返回结果为新张量，而不是视图

torch.cat(tensors, dim=0, *, out=None) → Tensor

- tensors (sequence of Tensors) – any python sequence of tensors of the same type. Non-empty tensors provided must have the same shape, except in the cat dimension.
- dim (int, optional) – the dimension over which the tensors are concatenated

- out (Tensor, optional) – the output tensor

out为可选参数，与直接返回的结果相同

In [116…
```
a = torch.zeros(2, 3)
b = torch.ones(2, 3)
Y = torch.tensor([])
X=torch.cat([a, b], out=Y)
print(id(X)==id(Y))
print(id(X.storage)==id(Y.storage))
```

```
True
True
```

PyTorch中，可以使用cat函数实现张量的拼接。

In [102…
```
a = torch.zeros(2, 3)
b = torch.ones(2, 3)
c = torch.zeros(3, 3)
print(a)
print(b)
print(c)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

In [104…
```
re = torch.cat([a, b])                    # 按照行进行拼接，dim默认取值为0
re
```

Out[104…
```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [1., 1., 1.],
        [1., 1., 1.]])
```

In [108…
```
re[0][0]=5.0
re
```

Out[108…
```
tensor([[5., 0., 0.],
        [0., 0., 0.],
        [1., 1., 1.],
        [1., 1., 1.]])
```

In [110…
```
a          # 说明合并后是新张量，而不是视图
```

Out[110…
```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

In [111…
```
torch.cat([a, b], 1)                    # 按照列进行拼接
```

Out[111…
```
tensor([[0., 0., 0., 1., 1., 1.],
        [0., 0., 0., 1., 1., 1.]])
```

在拼接时，两个张量除了待拼接维度外，如果其他维度上的值不同，则会报错

In [112…

```
torch.cat([a, c], 1)                    # 形状不匹配时将报错
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-112-8bdd1a857266> in <module>
----> 1 torch.cat([a, c], 1)                # 形状不匹配时将报错

RuntimeError: Sizes of tensors must match except in dimension 1. Got 2 and 3 i
n dimension 0 (The offending index is 1)
```

注意理解，拼接的本质是实现元素的堆积，也就是构成a、b两个二维张量的各一维张量的堆积，最终还是构成二维向量。

- 堆叠函数：stack

和拼接不同，堆叠不是将元素拆分重装，而是简单的将各参与堆叠的对象分装到一个更高维度的张量里。

堆叠和拼接的主要区别在于：堆叠会增加张量维度，而拼接不改变张量维度

In [117…
```
a
```

Out[117…
```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

In [118…
```
b
```

Out[118…
```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

In [119…
```
torch.stack([a, b])                    # 堆叠之后，生成一个三维张量
```

Out[119…
```
tensor([[[0., 0., 0.],
         [0., 0., 0.]],

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

In [120…
```
torch.stack([a, b]).shape
```

Out[120…
```
torch.Size([2, 2, 3])
```

In [158…
```
torch.cat([a, b])
```

Out[158…
```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [1., 1., 1.],
        [1., 1., 1.]])
```

注意对比二者区别，拼接之后维度不变，堆叠之后维度升高。拼接是把一个个元素单独提取出来之后再放到二维张量中，而堆叠则是直接将两个二维张量封装到一个三维张量中，因此，堆叠的要求更高，参与堆叠的张量必须shape形状完全相同。

In [121…
```
a
```

Out[121…
```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

In [122…
```
c
```

Out[122…
```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

In [123…
```
torch.cat([a, c])                    # 横向拼接时，对行数没有一致性要求
```

Out[123…
```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

In [125…
```
torch.stack([a, c])                  # 报错是因为a和c的shape不同
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-125-d0064288b164> in <module>
----> 1 torch.stack([a, c])                  # 报错是因为a和c的shape不同

RuntimeError: stack expects each tensor to be equal size, but got [2, 3] at en
try 0 and [3, 3] at entry 1
```

在Python基础数据类型中，append()会把第二个list作为整体放入第一个list中，而extend()则会把第二个list完全打散，然后放在第一个list后面

In [145…
```
a=[1,3,5,7]
b=[2,4,6,8]
```

In [146…
```
a.append(b)
a
```

Out[146…
```
[1, 3, 5, 7, [2, 4, 6, 8]]
```

In [147…
```
a=[1,3,5,7]
b=[2,4,6,8]
```

In [148…
```
a.extend(b)
a
```

Out[148…
```
[1, 3, 5, 7, 2, 4, 6, 8]
```

六、张量维度变换

此前我们介绍过，通过reshape方法，能够灵活调整张量的形状。而在实际操作张量进行计算时，往往需要另外进行降维和升维的操作，当我们需要除去不必要的维度时，可以使用squeeze函数，而需要手动升维时，则可采用unsqueeze函数。

In [ ]

In [149…
```
a = torch.arange(4)
a
```

Out[149…
```
tensor([0, 1, 2, 3])
```

In [150…
```
a2 = a.reshape(1, 4)
a2
```

Out[150…
```
tensor([[0, 1, 2, 3]])
```

In [151…
```
torch.squeeze(a2).ndim  # 重新降维至1维张量
```

Out[151…
```
1
```

- squeeze函数：删除不必要的维度，也就是保留值不是1的维度

In [152…
```
t = torch.zeros(1, 1, 3, 1)
t
```

Out[152…
```
tensor([[[[0.],
          [0.],
          [0.]]]])
```

In [153…
```
t.shape
```

Out[153…
```
torch.Size([1, 1, 3, 1])
```

t张量解释：一个包含一个三维的四维张量，三维张量只包含一个三行一列的二维张量。

In [154…
```
torch.squeeze(t)
```

Out[154…
```
tensor([0., 0., 0.])
```

In [155…
```
torch.squeeze(t).shape
```

Out[155…
```
torch.Size([3])
```

转化后生成了一个一维张量

In [156…
```
t1 = torch.zeros(1, 1, 3, 2, 1, 2)
t1.shape
```

Out[156…
```
torch.Size([1, 1, 3, 2, 1, 2])
```

In [157…
```
torch.squeeze(t1)
```

Out[157…
```
tensor([[[0., 0.],
         [0., 0.]],

        [[0., 0.],
         [0., 0.]],
```

```
        [[0., 0.],
         [0., 0.]]])
```

In [158…
```
torch.squeeze(t1).shape
```

Out[158…
```
torch.Size([3, 2, 2])
```

简单理解，squeeze就相当于提出了shape返回结果中的1

- unsqeeze函数：手动升维，目的是使用矩阵乘法（二维张量及以上），而不是用向量乘法（一维张量）

In [161…
```
t = torch.zeros(1, 2, 1, 2)
t.shape
```

Out[161…
```
torch.Size([1, 2, 1, 2])
```

方式是在dim处插入一个值为1的新维度

In [164…
```
torch.unsqueeze(t, dim = 0)                    # 在第1个维度索引上升高1个维度，不改变t本身
```

Out[164…
```
tensor([[[[[0., 0.]],

           [[0., 0.]]]]])
```

In [165…
```
torch.unsqueeze(t, dim = 0).shape
```

Out[165…
```
torch.Size([1, 1, 2, 1, 2])
```

In [166…
```
torch.unsqueeze(t, dim = 2).shape              # 在第3个维度索引上升高1个维度
```

Out[166…
```
torch.Size([1, 2, 1, 1, 2])
```

In [167…
```
torch.unsqueeze(t, dim = 4).shape              # 在第5个维度索引上升高1个维度
```

Out[167…
```
torch.Size([1, 2, 1, 2, 1])
```

注意理解维度和shape返回结果一一对应的关系，shape返回的序列有几个元素，张量就有多少维度。

注意torch.squeeze和torch.unsqueeze都不改变原张量的shape，需要用一个新的变量来接收升维或降维的结果