Part 1. 张量（Tensor）的创建和常用方法

在实际使用PyTorch的过程中，张量（Tensor）对象是我们操作的基本数据类型。

很多时候，在我们没有特别明确什么是深度学习计算框架的时候，我们可以把PyTorch简单看成是Python的深度学习第三方库，在PyTorch中定义了适用于深度学习的基本数据结构————张量，以及张量的各类计算。其实也就相当于NumPy中定义的Array和对应的科学计算方法，正是这些基本数据类型和对应的方法函数，为我们进一步在PyTorch上进行深度学习建模提供了基本对象和基本工具。因此，在正式使用PyTorch进行深度学习建模之前，我们需要熟练掌握PyTorch中张量的基本操作方法。

当然，值得一提的是，张量的概念并非PyTorch独有，目前来看，基本上通用的深度学习框架都拥有张量这一类数据结构，但不同的深度学习框架中张量的定义和使用方法都略有差别。而张量作为数组的衍生概念，其本身的定义和使用方法和NumPy中的Array非常类似，甚至，在复现一些简单的神经网络算法场景中，我们可以直接使用NumPy中的Array来进行操作。当然，此处并不是鼓励大家使用NumPy来进行深度学习，因为毕竟NumPy中的Array只提供了很多基础功能，写简单神经网络尚可，写更加复杂的神经网络则会非常复杂，并且Array数据结构本身也不支持GPU运行，因此无法应对工业场景中复杂神经网络背后的大规模数值运算。但我们需要知道的是，工具的差异只会影响实现层的具体表现，因此，一方面，我们在学习的过程中，不妨对照NumPy中的Array来进行学习，另一方面，我们更需要透过工具的具体功能，来理解和体会背后更深层次的数学原理和算法思想。

- 查看Pytorch版本号的方法

In [9]:
```
import torch
torch.__version__
```

Out[9]: '1.8.1'

一、张量（Tensor）的基本创建及其类型

和NumPy中的dnarray一样，张量的本质也是结构化的组织了大量的数据。并且，在实际操作过程中，张量的创建和基本功能也和NumPy中的array非常类似。

1.张量（Tensor）函数创建方法-3种

张量的最基本创建方法和NumPy中创建Array的格式一致，都是 创建函数(序列）的格式：**张量创建函数：torch.tensor()**

- （1）通过列表创建张量

In [10]:
```
t = torch.tensor([1, 2])
t
```

Out[10]: tensor([1, 2])

- （2）通过元组创建张量

In [11]:
```
torch.tensor((1, 2))
```

Out[11]:
```
tensor([1, 2])
```

- （3）通过Numpy数组创建张量

In [25]:
```
import numpy as np
a = np.array((1, 2))
a
```

Out[25]:
```
array([1, 2])
```

In [26]:
```
# 通过数组创建张量
t1 = torch.tensor(a)
t1
```

Out[26]:
```
tensor([1, 2])
```

2.张量的类型-5种

- (1) 整型
  - 整数型的数组Numpy Array默认创建int32（整型）类型（但是我的版本默认创建int64类型），而张量Tensor则默认创建int64（长整型）类型
  - 无论Numpy创建的类型是int32还是int64，转为张量后都与原来保持不变，原来是int32，转为张量还是int32
  - 准确来说，外侧创建的Tensor继承了内侧Numpy Array的数据类型
  - 如果强行把浮点型Tensor设置为整型，则直接舍去小数部分，不进行四舍五入！！！！

In [49]:
```
np.array([1, 2]).dtype
```

Out[49]:
```
dtype('int64')
```

In [34]:
```
torch.tensor(np.array([1, 2])).dtype
```

Out[34]:
```
torch.int64
```

In [35]:
```
np.array([1, 2], dtype='int32').dtype
```

Out[35]:
```
dtype('int32')
```

In [36]:
```
torch.tensor(np.array([1, 2], dtype='int32')).dtype
```

Out[36]:
```
torch.int32
```

- (2) 浮点型
  - 创建浮点型数组时，Numpy Array则是默认float64（双精度浮点型），张量Tensor默认是float32（单精度浮点型）
  - 这是为什么很多浮点型数据先用Numpy Array，再转换为Tensor的原因，直接用Tensor会有精度损失

In [40]:

```
np.array([1.1, 2.2]).dtype
```

Out[40]:
```
dtype('float64')
```

In [41]:
```
torch.tensor(np.array([1.1, 2.2])).dtype
```

Out[41]:
```
torch.float64
```

In [46]:
```
torch.tensor([1.11, 2.2]).dtype
```

Out[46]:
```
torch.float32
```

- (3) 布尔型

In [47]:
```
t2 = torch.tensor([True, False])
t2
```

Out[47]:
```
tensor([ True, False])
```

In [48]:
```
t2.dtype
```

Out[48]:
```
torch.bool
```

和数组不同，对于张量而言，数值型和布尔型张量就是最常用的两种张量类型，相关类型总结如下。

PyTorch中Tensor类型

数据类型	dtype
32bit浮点数	torch.float32或torch.float
64bit浮点数	torch.float64或torch.double
16bit浮点数	torch.float16或torch.half
8bit无符号整数	torch.unit8
8bit有符号整数	torch.int8
16bit有符号整数	torch.int16或torch.short
16bit有符号整数	torch.int16或torch.short
32bit有符号整数	torch.int32或torch.int
64bit有符号整数	torch.int64或torch.long
布尔型	torch.bool
复数型	torch.complex64

- (4) 人为设置dtype参数

In [50]:
```
# 创建int16整型张量
torch.tensor([1.1, 2.7], dtype = torch.int16)
```

Out[50]:
```
tensor([1, 2], dtype=torch.int16)
```

- (5) 复数类型

In [18]:
```
a = torch.tensor(1 + 2j)            # 1是实部、2是虚部
a
```

Out[18]:
```
tensor(1.+2.j)
```

3.张量类型的转化

- 说明：单精度浮点指float32，半精度浮点指float16，双精度浮点指float63

- （1）张量类型的隐式转化
  - 和NumPy中array相同，当张量各元素属于不同类型时，系统会自动进行隐式转化。
    - float是级别最高的类型
    - True对应1，False对应0
    - 目的是统一类型，方便后续运算

In [42]:
```
# 浮点型和整数型的隐式转化
torch.tensor([1.1, 2]).dtype
```

Out[42]:
```
torch.float32
```

In [44]:
```
# 布尔型和数值型的隐式转化
torch.tensor([True, 2.0])
```

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

- （2）张量类型的显式转化（参数法）
  - 即在tensor等中使用dtype和torch.float参数来设置（需要详细指明精度）
  - 例如，创建张量时加上dtype=torch.int64

- （3）张量类型的显式转化（对象方法）
  - 对创建好的tensor等对象使用“方法”，如tensor.double()，这里不需要写详细精度
  - 通过.double()，.float()、.int()等方法，但是注意此方法不改变原来变量的类型，只是返回一个新的变量
  - 作用：pytorch中很多函数只支持浮点数，属于静态编程，不会自动转换，所以要人为转换

In [53]:
```
# 转化为默认浮点型（32位）
print(torch.tensor([1, 2]).float())
print(torch.tensor([1, 2]).float().dtype)
```

```
tensor([1., 2.])
torch.float32
```

In [54]:
```
# 转化为双精度浮点型
print(torch.tensor([1, 2]).double())
print(torch.tensor([1, 2]).double().dtype)
```

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

In [56]:
```
print(torch.tensor([1, 2]).dtype)
```

```
torch.int64
```

In [58]:
```
# 转化为16位整数
print(torch.tensor([1, 2]).short())
print(torch.tensor([1, 2]).short().dtype)
```

```
tensor([1, 2], dtype=torch.int16)
torch.int16
```

总结

- 当在torch函数中使用dtype参数时候，需要输入torch.float表示精度；
- 在使用方法进行类型转化时，双精度的方法名称则是double。
- 方法（2）和（3）区别在于前者改变原本的类型，而后者是是生成一个新的，在考虑到内存时注意一下

二、张量的维度与形变

张量作为一组数的结构化表示，也同样拥有维度的概念。简答理解，向量就是一维的数组，而矩阵则是二维的数组，以此类推，在张量中，我们还可以定义更高维度的数组。当然，张量的高维数组和NumPy中高维Array概念类似。

1.张量的维度属性-4个

- （1）张量的维度：X.ndim

In [59]:
```
t1 = torch.tensor([1, 2])
t1
```

Out[59]:
```
tensor([1, 2])
```

In [60]:
```
# 使用ndim属性查看张量的维度
t1.ndim
```

Out[60]:
```
1
```

- （2）张量的形状：X.shape或X.size()，区别在于前者是对象的参数，后者是对象的方法
  - 和NumPy不同，PyTorch中size方法返回结果和shape属性返回结果一致。
  - 举个例子：对于256*256*3的图像，可以看成3维张量(XXX.ndim=3)，形状是256*256*3，即第一个维度下有256个元素，第二个维度下有256个元素，第三个维度下有3个元素

In [61]:
```
# 使用shape查看形状
t1.shape
```

Out[61]:
```
torch.Size([2])
```

In [62]:
```
# 和size函数相同
```

```
t1.size()
```

Out[62]:
```
torch.Size([2])
```

- （3）有几个N-1维元素：len(X)
  - 具体含义是对于一个N维张量（N=XXX.ndim），其中包含几个N-1维元素，等价于XXX.shape或size返回的第N-1个元素（假设一共返回N个元素），比如[1, 2; 3, 4]是2维张量，可以看成2个1维(行)张量；

In [63]:
```
# 返回拥有几个（N-1）维元素
len(t1)
```

Out[63]:
```
2
```

- （4）张量中最基层的元素总数：X.numel()
  - 一维张量len和numel返回结果相同，但更高维度张量则不然

In [64]:
```
# 返回总共拥有几个数
t1.numel()
```

Out[64]:
```
2
```

再举一个二维张量的例子：

In [66]:
```
# 用list的list创建二维数组
t2 = torch.tensor([[1, 2], [3, 4]])
t2
```

Out[66]:
```
tensor([[1, 2],
        [3, 4]])
```

In [67]:
```
t2.ndim
```

Out[67]:
```
2
```

In [68]:
```
t2.shape
```

Out[68]:
```
torch.Size([2, 2])
```

In [69]:
```
t2.size()
```

Out[69]:
```
torch.Size([2, 2])
```

In [70]:
```
len(t2)
```

Out[70]:
```
2
```

理解：此处len函数返回结果代表t2由两个1维张量构成

In [71]:
```
t2.numel()
```

Out[71]: 4

理解：此处numel方法返回结果代表t2由总共由4个数构成

2.零维张量

在PyTorch中，还有一类特殊的张量，被称为零维张量。该类型张量只包含一个元素，但又不是单独一个数。

In [122…
```
t = torch.tensor(1)
t
```

Out[122… tensor(1)

In [123…
```
t.ndim
```

Out[123… 0

In [124…
```
t.shape
```

Out[124… torch.Size([])

In [173…
```
t.numel()
```

Out[173… 1

理解零维张量:

目前，我们可将零维张量视为拥有张量属性的单独一个数。（例如，张量可以存在GPU上，但Python原生的数值型对象不行，但零维张量可以，尽管是零维。）从学术名称来说，Python中单独一个数是scalars（标量），而零维的张量则是tensor。相当于Python basic变量移植到GPU上

易错点：仅有一个元素时，若以列表的形式传入，则为一维张量，若以单个元素的形式传入，则为零维张量，前者可以理解成一维向量，后者可以理解成标量

In [81]:
```
t0 = torch.tensor([1]) # 一维张量
t0.ndim
```

Out[81]: 1

In [82]:
```
t = torch.tensor(1) # 零维张量
t.ndim
```

Out[82]: 0

3.高维张量

一般来说，三维及三维以上的张量，我们就将其称为高维张量。当然，在高维张量中，最常见的还是三维张量。我们可以将其理解为二维数组或者矩阵的集合。

- 编程时，特别是写函数，不要想着输入1个得到1个，而要想输入一个矩阵得到一个结果矩阵，从而高效

- N维张量可以看成多个相同形状的N-1维张量的集合形式！！！！！！
- N维张量的创建方法，我们可以先创建M个N-1维的数组，然后将其拼成一个N维的张量

In [72]:
```
a1 = np.array([[1, 2, 2], [3, 4, 4]])
a1
```

Out[72]:
```
array([[1, 2, 2],
       [3, 4, 4]])
```

In [73]:
```
a2 = np.array([[5, 6, 6], [7, 8, 8]])
a2
```

Out[73]:
```
array([[5, 6, 6],
       [7, 8, 8]])
```

In [74]:
```
# 由两个形状相同的二维数组创建一个三维的张量
t3 = torch.tensor([a1, a2])
t3
```

Out[74]:
```
tensor([[[1, 2, 2],
         [3, 4, 4]],

        [[5, 6, 6],
         [7, 8, 8]]])
```

In [75]:
```
t3.ndim
```

Out[75]:
```
3
```

In [76]:
```
t3.shape            # 包含两个，两行三列的矩阵的张量。
```

Out[76]:
```
torch.Size([2, 2, 3])
```

- shape = [2, 2, 3]可以这么理解，该张量由2个矩阵组成（第一个2），每个矩阵均为2行3列（后面的2和3），也可以理解为，该张量由3个矩阵组成（最后的3），每个矩阵均为2行2列（前面的两个2），还可以层次化理解，即该张量由2个元素A组成（第一个2），元素A又由2个元素B组成（第二个2），元素B则由3个最基本的元素组成。

In [77]:
```
len(t3)
```

Out[77]:
```
2
```

In [78]:
```
t3.numel()
```

Out[78]:
```
12
```

4.张量的形变-2种

张量作为数字的结构化集合，其结构也是可以根据实际需求灵活调整的。

（1）flatten拉平：将任意维度张量转化为一维张量

- 默认是按行拉平

In [83]:
```
t2
```

Out[83]:
```
tensor([[1, 2],
        [3, 4]])
```

In [84]:
```
t2.flatten()
```

Out[84]:
```
tensor([1, 2, 3, 4])
```

In [85]:
```
t3
```

Out[85]:
```
tensor([[[1, 2, 2],
         [3, 4, 4]],

        [[5, 6, 6],
         [7, 8, 8]]])
```

In [86]:
```
t3.flatten()
```

Out[86]:
```
tensor([1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 8])
```

注：如果将零维张量使用flatten，则会将其转化为一维张量!!!

In [88]:
```
t
```

Out[88]:
```
tensor(1)
```

In [89]:
```
t.flatten()
```

Out[89]:
```
tensor([1])
```

In [90]:
```
t.flatten().ndim
```

Out[90]:
```
1
```

（2）reshape方法：任意变形

In [92]:
```
t1
```

Out[92]:
```
tensor([1, 2])
```

In [93]:
```
# 转化为两行、一列的向量
t1.reshape(2, 1)
```

Out[93]:
```
tensor([[1],
        [2]])
```

注意：reshape转化后的维度由该方法输入的参数"个数"决定，该个数为N，则形变后就是N维张量!!!

- 本质是改变了size或shape的结果，因此参数的具体值要符合原来的size

- 转化后生成一维张量
  - 注：reshape(N,)是转为1维张量的特殊写法

In [94]:
```
t1.reshape(2)
```

Out[94]:
```
tensor([1, 2])
```

In [95]:
```
t1.reshape(2).ndim
```

Out[95]:
```
1
```

In [96]:
```
# 注，另一种表达形式
t1.reshape(2, )
```

Out[96]:
```
tensor([1, 2])
```

- 转化后生成二维张量

In [97]:
```
t1.reshape(1, 2)          # 生成包含一个两个元素的二维张量
```

Out[97]:
```
tensor([[1, 2]])
```

In [98]:
```
t1.reshape(1, 2).ndim
```

Out[98]:
```
2
```

- 转化后生成三维张量
  - 低维度的张量也可以拉伸成高纬度张量，只需要添加1即可
  - 只需要满足reshape里的所有参数相乘等于总的底层元素数即可

In [99]:
```
t1.reshape(1, 1, 2)
```

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

In [100…
```
t1.reshape(1, 2, 1)
```

Out[100…
```
tensor([[[1],
         [2]]])
```

In [101…
```
# 注意转化过程维度的变化
t1.reshape(1, 2, 1).ndim
```

Out[101…
```
3
```

如何利用reshape方法，将高维度张量拉平?

In [102…
```
t3
```

Out[102…
```
tensor([[[1, 2, 2],
```

```
        [3, 4, 4]],

       [[5, 6, 6],
        [7, 8, 8]]])
```

In [103…
```
t3.reshape(t3.numel())
```

Out[103…
```
tensor([1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 8])
```

三、特殊张量的创建方法

在很多数值科学计算的过程中，都会创建一些特殊取值的张量，用于模拟特殊取值的矩阵，如全0矩阵、对角矩阵等。因此，PyTorch中也存在很多创建特殊张量的函数。

- 注：以下大多数默认是浮点型张量

1.特殊取值的张量创建方法-11种

- （1）全0张量
  - 注：由于zeros就已经确定了张量元素取值，因此该函数传入的参数实际上是决定了张量的形状

In [224…
```
torch.zeros([2, 3])            # 创建全是0的，两行、三列的张量（矩阵）
```

Out[224…
```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

- （2）全1张量

In [225…
```
torch.ones([2, 3])
```

Out[225…
```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

- （3）单位矩阵

In [226…
```
torch.eye(5)
```

Out[226…
```
tensor([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.]])
```

- （4）对角矩阵
  - 在PyTorch中，需要先把对角线上的元素化成一维张量，然后去创建对角矩阵，注意不能直接用list生成对角张量！！

In [232…
```
t1
```

Out[232…
```
tensor([1, 2])
```

In [235…
```
torch.diag(t1)
```

Out[235…
```
tensor([[1, 0],
        [0, 2]])
```

In [237…
```
torch.diag([1, 2])              # 不能使用list直接创建对角矩阵
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-237-314c189cb631> in <module>
----> 1 torch.diag([1, 2])              # 不能使用list直接创建对角矩阵

TypeError: diag(): argument 'input' (position 1) must be Tensor, not list
```

- （5）rand：服从0-1均匀分布的张量

In [239…
```
torch.rand(2, 3)
```

Out[239…
```
tensor([[0.9223, 0.9948, 0.2804],
        [0.8130, 0.2890, 0.5319]])
```

- （6）randn：服从标准正态分布的张量

In [240…
```
torch.randn(2, 3)
```

Out[240…
```
tensor([[-1.2513,  0.6465, -2.3011],
        [ 0.8447,  1.6856,  1.3615]])
```

- （7）normal：服从指定正态分布的张量
  - 注意前面的2和3两个参数是设置均值和方差的

In [245…
```
torch.normal(2, 3, size = (2, 2))           # 均值为2，标准差为3的张量
```

Out[245…
```
tensor([[2.4660, 1.4952],
        [6.0202, 0.7525]])
```

- （8）randint：整数随机采样结果

In [263…
```
torch.randint(1, 10, [2, 4])                # 在1-10之间随机抽取整数，组成两行四列
```

Out[263…
```
tensor([[5, 8, 8, 3],
        [6, 1, 4, 2]])
```

- （9）arange/linspace：生成数列
  - arrange左闭右开，按照设定间隔取数字，与range是完全等价的
  - linspacezu左右均闭，按照设定的参数等距离地取几个数字

In [250…
```
torch.arange(5)                             # 和range相同
```

Out[250…
```
tensor([0, 1, 2, 3, 4])
```

In [252…
```
torch.arange(1, 5, 0.5)                     # 从1到5（左闭右开），每隔0.5取值一个
```

Out[252…
```
tensor([1.0000, 1.5000, 2.0000, 2.5000, 3.0000, 3.5000, 4.0000, 4.5000])
```

In [256…
```
torch.linspace(1, 5, 3)                    # 从1到5（左右都包含），等距取三个数
```
Out[256…
```
tensor([1., 3., 5.])
```

- （10）empty：生成未初始化的指定形状矩阵，每个元素都无限接近0

In [257…
```
torch.empty(2, 3)
```
Out[257…
```
tensor([[0.0000e+00, 1.7740e+28, 1.8754e+28],
        [1.0396e-05, 1.0742e-05, 1.0187e-11]])
```

- （11）full：根据指定形状，填充指定数值

In [265…
```
torch.full([2, 4], 2)
```
Out[265…
```
tensor([[2, 2, 2, 2],
        [2, 2, 2, 2]])
```

2.创建指定形状的数组-加后缀

- _like，根据其他张量的形状进行填充等，只需要在以上函数后加_like即可，如full_like、zeros_like等

In [281…
```
t1
```
Out[281…
```
tensor([1, 2])
```

In [267…
```
t2
```
Out[267…
```
tensor([[1, 2],
        [3, 4]])
```

In [269…
```
torch.full_like(t1, 2)                # 根据t1形状，填充数值2
```
Out[269…
```
tensor([2, 2])
```

In [275…
```
torch.randint_like(t2, 1, 10)
```
Out[275…
```
tensor([[4, 8],
        [5, 8]])
```

In [284…
```
torch.zeros_like(t1)
```
Out[284…
```
tensor([0, 0])
```

_like需要注意转化前后数据类型一致性的问题，即两个张量的类型必须一致！

In [285…
```
torch.randn_like(t1)                    # t1是整数，而转化后将变为浮点数，此时代码将报
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-285-82e06104e8a8> in <module>
----> 1 torch.randn_like(t1)

RuntimeError: "normal_kernel_cpu" not implemented for 'Long'
```

In [287…
```
t10 = torch.tensor([1.1, 2.2])        # 重新生成一个新的浮点型张量
t10
```

Out[287…
```
tensor([1.1000, 2.2000])
```

In [288…
```
torch.randn_like(t10)                 # 即可执行相应的填充转化
```

Out[288…
```
tensor([1.0909, 0.0379])
```

四、张量（Tensor）和其他相关类型之间的转化方法

- 张量、数组和列表是较为相似的三种类型对象，在实际操作过程中，经常会涉及三种对象的相互转化。
- 在此前张量的创建过程中，我们看到torch.tensor函数可以直接将数组或者列表转化为张量，而我们也可以将张量转化为数组或者列表。另外，前文介绍了0维张量的概念，此处也将进一步给出零维张量和数值对象的转化方法。

- 1.张量转化为数组：.numpy方法或np.array()函数

In [110…
```
t1=torch.tensor([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10], dtype=torch.int64)
t1
```

Out[110…
```
tensor([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [111…
```
t1.numpy()
```

Out[111…
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [113…
```
# 当然，也可以通过np.array函数直接转化为array
np.array(t1)
```

Out[113…
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

- 2.张量转化为列表：.tolist方法

In [114…
```
t1.tolist()
```

Out[114…
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- 3. 张量转化为列表: list函数
  - ### 需要注意的是，此时转化的列表是由一个个零维张量构成的列表，而非张量的数值组成的列表。
    - 这是因为构成一维张量的不是普通的数字，而是零维张量

In [115…
```
list(t1)
```

Out[115…
```
[tensor(1),
 tensor(2),
 tensor(3),
 tensor(4),
 tensor(5),
 tensor(6),
 tensor(7),
 tensor(8),
 tensor(9),
 tensor(10)]
```

- 4.张量转化为数值：.item()方法
  - 作用：取出零维张量中的元素值

在很多情况下，我们需要将最终计算的结果张量转化为单独的数值进行输出，此时需要使用.item方法来执行。

In [3]:
```
n = torch.tensor(1)
n
```

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

In [4]:
```
n.item()
```

Out[4]:
```
1
```

五、张量的深拷贝

- 和Python中其他对象类型一样，等号赋值操作实际上是浅拷贝，需要进行深拷贝，则需要使用clone方法

In [118…
```
t1
```

Out[118…
```
tensor([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [119…
```
t11 = t1                              # t11是t1的浅拷贝
t11
```

Out[119…
```
tensor([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [120…
```
t1[1]
```

Out[120…
```
tensor(2)
```

In [121…
```
t1[1] = 10                            # t1修改
```

In [122…
```
t1
```

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

In [123…
```
t11                                    # t11会同步修改
```

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

此处t1和t11二者指向相同的对象。而要使得t11不随t1对象改变而改变，则需要对t11进行深拷贝，从而使得t11单独拥有一份对象。

In [124…
```
t11 = t1.clone()
```

In [125…
```
t1
```

Out[125… tensor([1, 10, 3, 4, 5, 6, 7, 8, 9, 10])

In [126…
```
t11
```

Out[126… tensor([1, 10, 3, 4, 5, 6, 7, 8, 9, 10])

In [127…
```
t1[0]
```

Out[127… tensor(1)

In [128…
```
t1[0] = 100
t1
```

Out[128… tensor([100, 10,   3,   4,   5,   6,   7,   8,   9, 10])

In [129…
```
t11
```

Out[129… tensor([1, 10, 3, 4, 5, 6, 7, 8, 9, 10])

补充

pytorch中的.Tensor、.tensor、.from_numpy、.as_tensor区别

> Tensor 和tensor唯一区别在于方法名中t的大小写，大写字母T（Tensor）是类构造函数，第二种小写（tensor）是工厂函数。其中，torch.as_tensor 和 torch.from_numpy 也是工厂函数。

In [1]:
```
import torch
import numpy as np
data = np.array([1, 2, 3])

Tensor = torch.Tensor(data)
tensor = torch.tensor(data)
from_numpy = torch.from_numpy(data)
```

```
as_tensor = torch.as_tensor(data)
print('输出的结果：')
print(Tensor)
print(tensor)
print(from_numpy)
print(as_tensor)

print('输出的类型：')
print(Tensor.dtype)
print(tensor.dtype)
print(from_numpy.dtype)
print(as_tensor.dtype)
```

```
输出的结果：
tensor([1., 2., 3.])
tensor([1, 2, 3])
tensor([1, 2, 3])
tensor([1, 2, 3])
输出的类型：
torch.float32
torch.int64
torch.int64
torch.int64
```

> 构造函数在构造一个张量时使用全局默认值，而工厂函数则根据输入推断数据类型。通过torch.get_default_dtype()可以查看dtype的全局默认值是torch.float32。

In [2]:
```
torch.get_default_dtype()
```

Out[2]:
```
torch.float32
```

工厂函数是根据传入的数据选择一个dtype，因此可以隐式转换

In [4]:
```
torch.tensor(np.array([1, 2, 3]))
```

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

In [5]:
```
torch.tensor(np.array([1., 2., 3.]))
```

Out[5]:
```
tensor([1., 2., 3.], dtype=torch.float64)
```

In [6]:
```
torch.tensor(np.array([1, 2, 3]), dtype=torch.float64)
```

Out[6]:
```
tensor([1., 2., 3.], dtype=torch.float64)
```

> Tensor 和tensor是深拷贝，在内存中创建一个额外的数据副本，不共享内存，所以不受数组改变的影响。from_numpy和as_tensor是浅拷贝，在内存中共享数据，他们不同之处就是在于对内存的共享。

In [7]:
```
import torch
import numpy as np
data = np.array([1, 2, 3])

Tensor = torch.Tensor(data)
```

```
tensor = torch.tensor(data)
from_numpy = torch.from_numpy(data)
as_tensor = torch.as_tensor(data)
print('改变前：')
print(Tensor)
print(tensor)
print(from_numpy)
print(as_tensor)
data[0] = 0
data[1] = 0
data[2] = 0
print('改变后：')
print(Tensor)
print(tensor)
print(from_numpy)
print(as_tensor)
```

```
改变前：
tensor([1., 2., 3.])
tensor([1, 2, 3])
tensor([1, 2, 3])
tensor([1, 2, 3])
改变后：
tensor([1., 2., 3.])
tensor([1, 2, 3])
tensor([0, 0, 0])
tensor([0, 0, 0])
```

分析：

- torch.as_tensor()和torch.from_numpy() 函数使得numpy数组与Pytorch张量之间切换可以非常快。
  - 因为创建新的Pytorch张量时，数据是共享的，而不是后台复制的。共享数据比复制数据更有效使用更少的内存。因为数据没有写到内存中的两个位置，而是只有一个位置。
- torch.tensor()是经常使用的。如果想做内存优化，使用torch.as_tensor()。
  - 这个为什么要比torch.from_numpy()好呢？因为，torch.as_tensor()函数可以接受list、tuple、ndarray、scalar等数据类型，而torch.from_numpy()仅接受Numpy数组。
- 需要注意的是，Numpy 在 64 位机子上浮点数默认的数据类型是 float64，而 Pytorch 默认的是 float32。所以为了确保转换后的数据类型是 float32，以及兼顾适用性，使用 torch.as_tensor() 都是更好的选择。

深入研究下torch.as_tensor()

torch.as_tensor(data, dtype=None,device=None)->Tensor : 为data生成tensor

- 参数：
  - data (array_like) – Initial data for the tensor. Can be a list, tuple, NumPy ndarray, scalar, and other types.
  - dtype (torch.dtype, optional) – the desired data type of returned tensor. Default: if None, infers data type from data.
  - device (torch.device, optional) – the device of the constructed tensor. If None and data is a tensor then the device of data is used. If None and data is not a tensor

then the result tensor is constructed on the CPU.

> 说明：dtype和device是可选参数：
>
> - 如果未填写dtype和device：
>   - 未填写dtype，生成的张量默认与data类型一致（整型、浮点数等），
>   - 未填写device：
>   - 如果data本身是张量，则生成的张量默认与data的device一致；
>   - 如果data本身不是张量，则生成的张量默认是CPU上的；
> - 如果填写了dtype和device：
>   - 如果data本身是张量，且其dtype和device与填写的参数相同，则生成的张量是浅拷贝；
>   - 如果data本身是ndarray，且其dtype与填写的参数相同，device为cpu，则生成的张量是浅拷贝；
>   - 其他情况不共享内存，即填写的dtype和device和data相比，只要有一个发生变化，就变成深拷贝

下面是Pytorch官方给出的例子：

In [ ]:
```
# 案例一
a = numpy.array([1, 2, 3])
t = torch.as_tensor(a)     # tensor([ 1,  2,  3])
# 浅拷贝
t[0] = -1                  # a改变 array([-1,  2,  3])

# 案例二
a = numpy.array([1, 2, 3])
t = torch.as_tensor(a, device=torch.device('cuda'))      # tensor([ 1,  2,  3]
# 深拷贝
t[0] = -1           # a不变 array([1,  2,  3])
```

下面是是一个模型中的例子，以后尽量使用 `torch.as_tensor`

In [ ]:
```
class AnchorGenerator(nn.Module):
    ...
    def generate_anchors(self, scales, aspect_ratios, dtype=torch.float32, de
        scales = torch.as_tensor(scales, dtype=dtype, device=device)
        aspect_ratios = torch.as_tensor(aspect_ratios, dtype=dtype, device=de
        h_ratios = torch.sqrt(aspect_ratios)
        w_ratios = 1 / h_ratios

        ws = (w_ratios[:, None] * scales[None, :]).view(-1)
        hs = (h_ratios[:, None] * scales[None, :]).view(-1)

        base_anchors = torch.stack([-ws, -hs, ws, hs], dim=1) / 2
        return base_anchors.round()
```