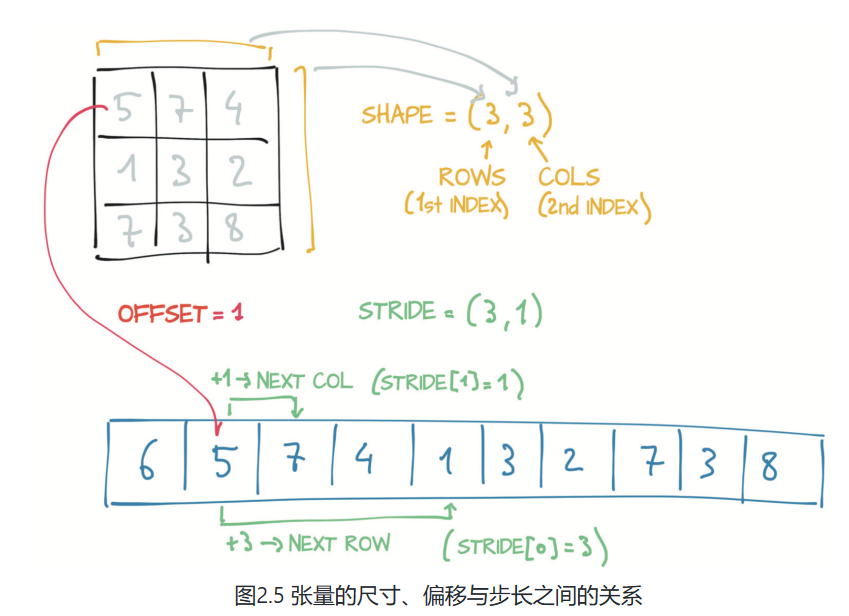
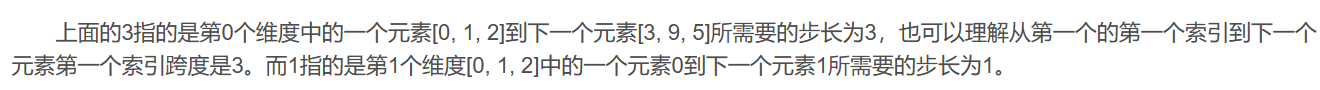
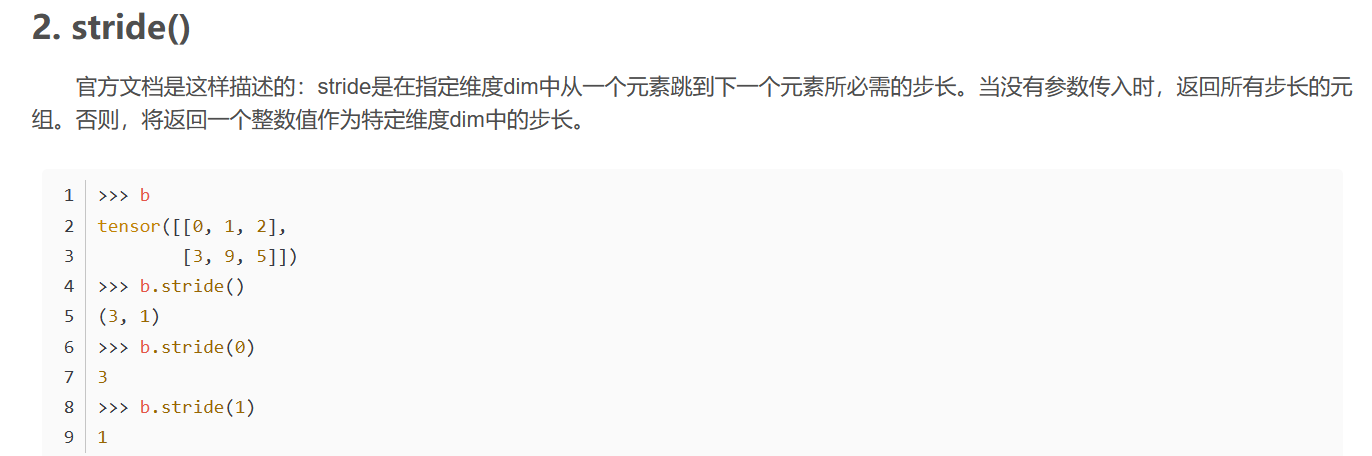
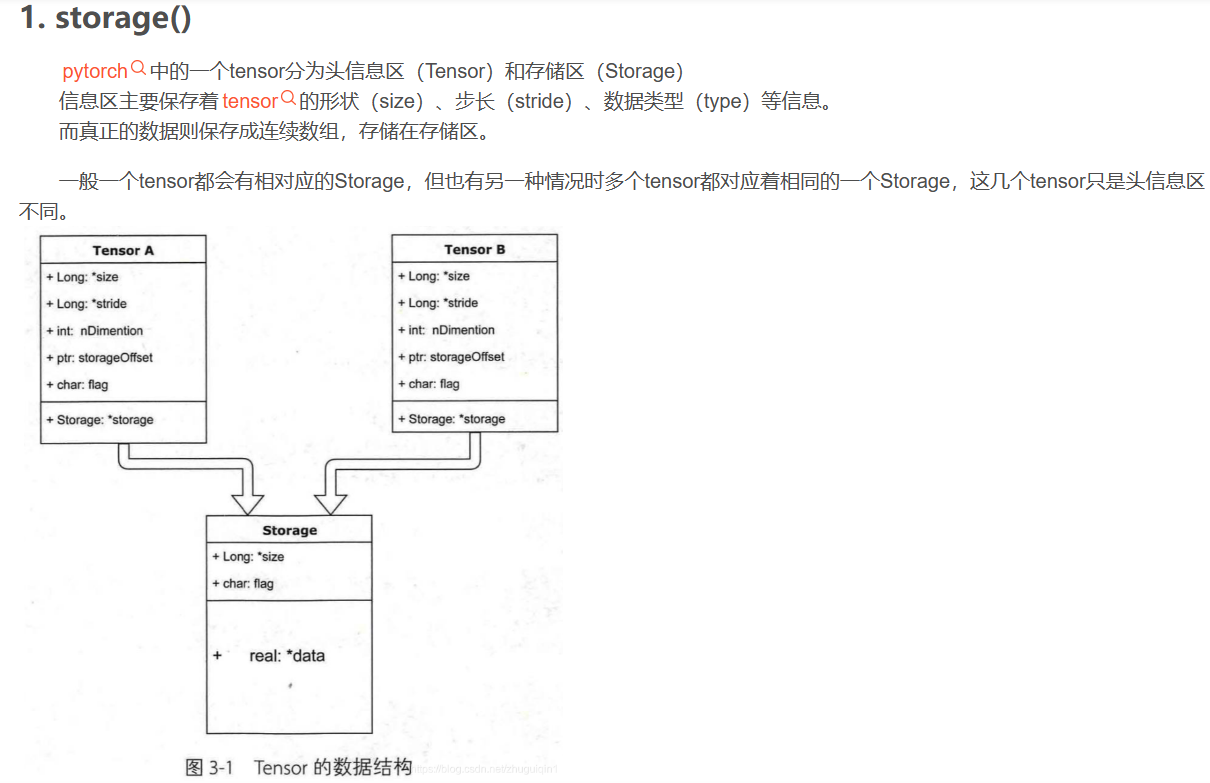
Pytorch

尺寸size：是一个元组，表示张量每个维度上有多少元素

存储偏移storage offset：存储中与张量中的第一个元素相对应的索引

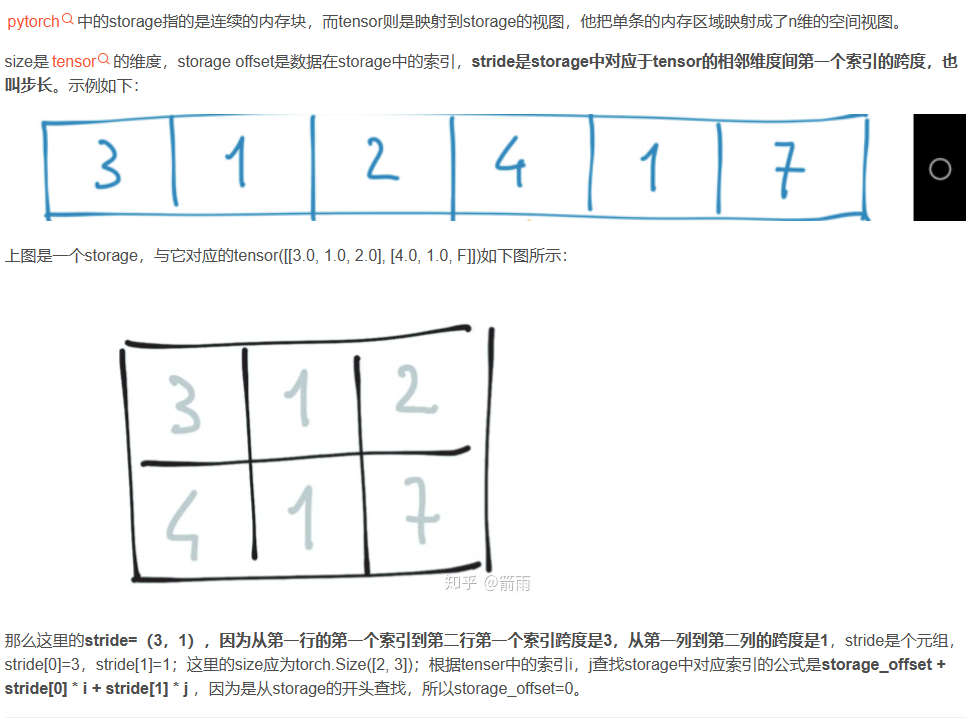
步长stride：在存储中为了沿每个维度获取下一个元素而需要跳过的元素数量



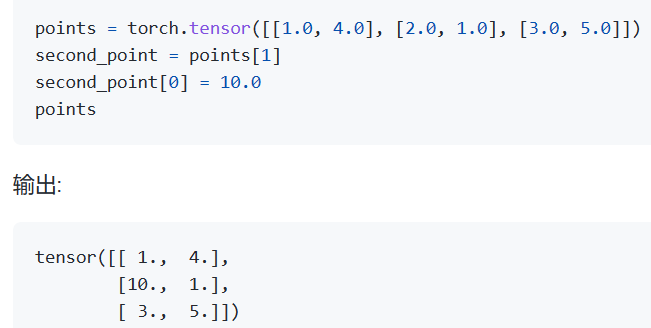
结果张量是一维的

步长：一个元组（分别表示行到下一行对应索引的跨度，列到下一列对应索引的跨度），表示当索引在每个维度上增加1时必须跳过的存储中元素的数量。

张量Tensor和和存储Storage之间的这种间接操作会使某些操作（例如转置或提取子张量）的代价很小，因为它们不会导致内存重新分配；相反，它们（仅仅）分配一个新的张量对象，该对象具有不同的尺寸、存储偏移或步长。



子张量改变会对原始张量产生影响。



为避免这种影响可以克隆子张量：second\_point = points[1].clone()

转置：points.t()

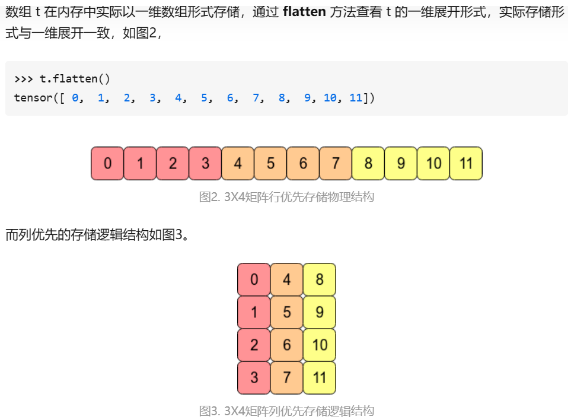
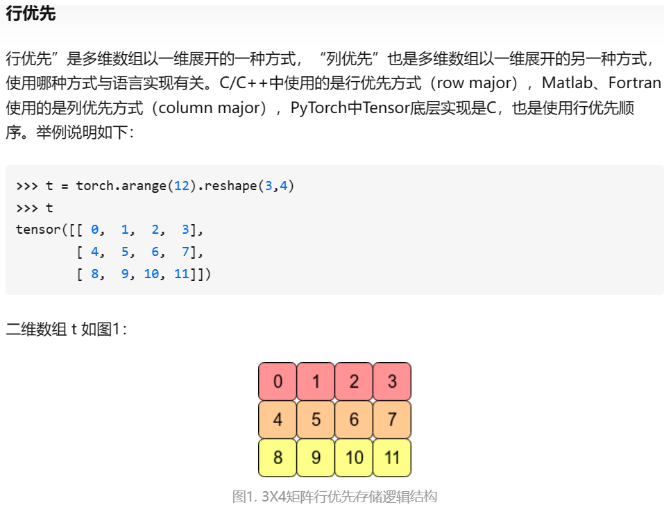
转置前和转置后的张量共享同一存储，仅仅是尺寸和步长不同：

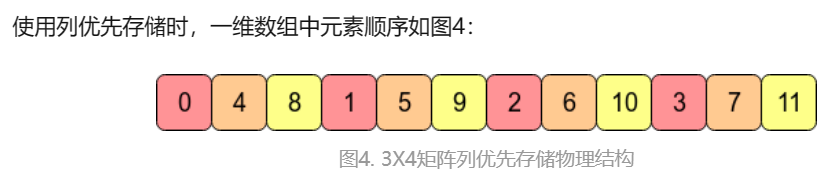
Id(points.storage())=id(points\_t.sotrage())

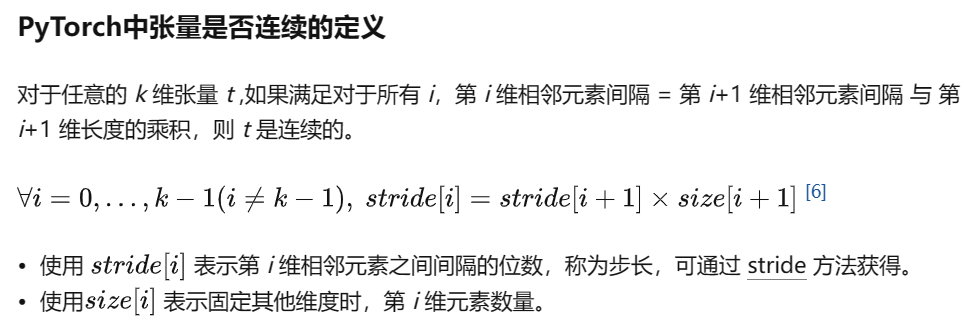
存储将张量中的元素逐行保存。

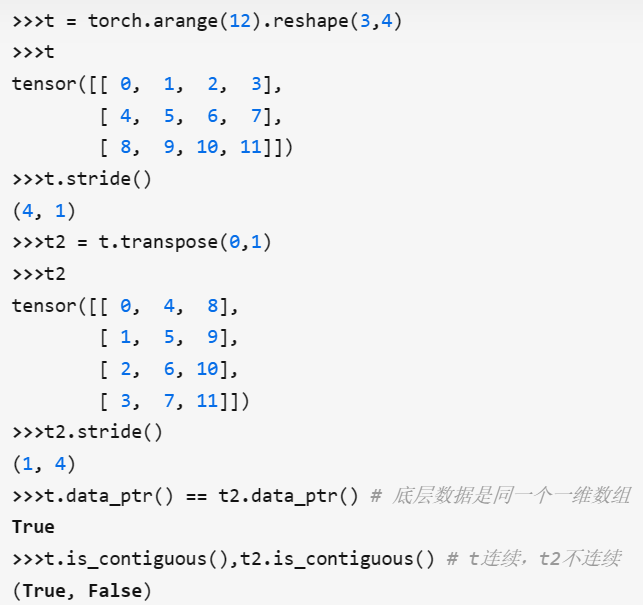
连续张量：tensor底层一维数组元素的存储顺序与Tensor按行优先一维展开的元素顺序一致——最后一维步长需为1

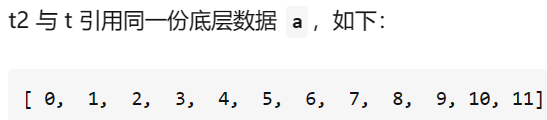
使用a.contiguous()从非连续张量获得新的连续张量

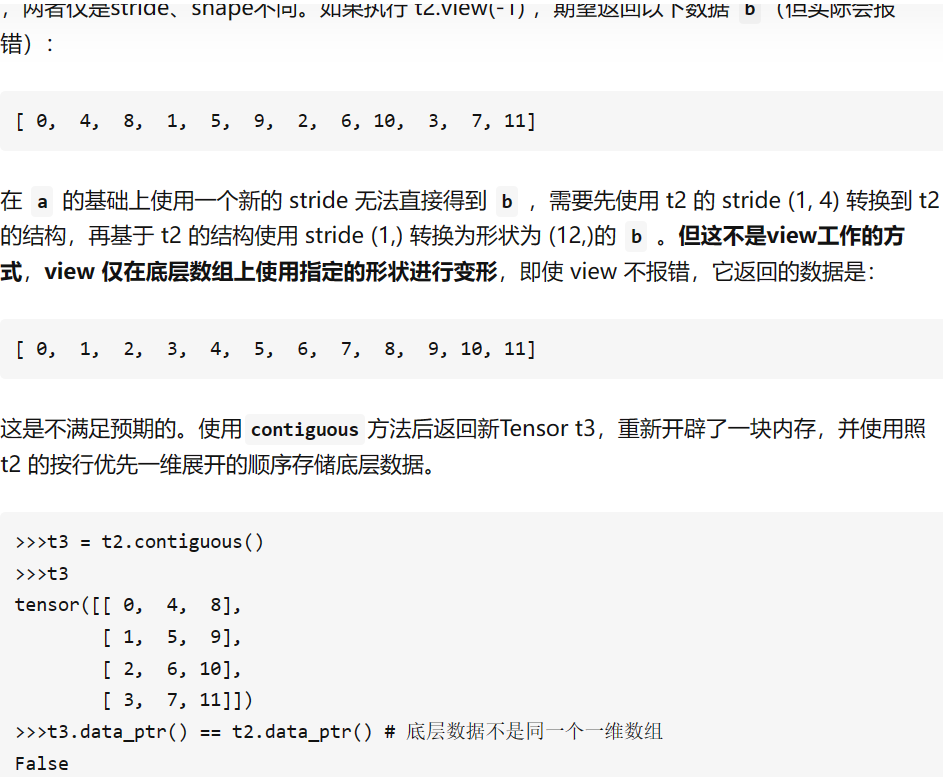
、

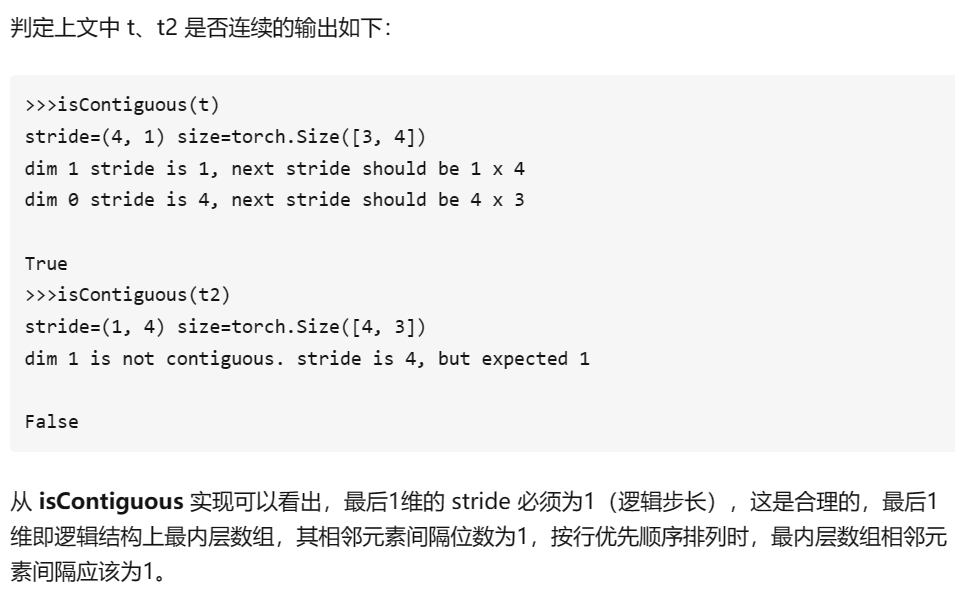












dtype数据类型指定张量可以容纳的可能值（整数/浮点数）以及每个值的字节数。

Dtype参数可能取值：

torch.float32或torch.float —— 32位浮点数----torch.Tensor/torch.FloatTensor

torch.float64或torch.double —— 64位双精度浮点数

torch.float16或torch.half —— 16位半精度浮点数

torch.int8 —— 带符号8位整数----torch.CharTensor

torch.uint8 —— 无符号8位整数----torch.ByteTensor

torch.int16或torch.short —— 带符号16位整数

torch.int32或torch.int —— 带符号32位整数

torch.int64或torch.long —— 带符号64位整数

每个torch.float、torch.double等等都有一个与之对应的具体类：torch.FloatTensor、torch.DoubleTensor等等。torch.int8对应的类是torch.CharTensor，而torch.uint8对应的类是torch.ByteTensor。torch.Tensor是torch.FloatTensor的别名，即默认数据类型为32位浮点型。

分配正确数字类型的张量，可以指定合适的dtype作为张量构造函数的参数：

Double\_points = torch.ones(10,2,dtype=torch.double)

Short\_points = torch.tensor([[1,2],[3,4]],dtype=torch.short)

Short\_points.dtype属性获取数据类型

您还可以使用相应的转换方法将张量创建函数的输出转换为正确的类型，例如

**torch.zeros(\*size, out=None, dtype=None, layout=torch.strided, device=None, requires\_grad=False)**

double\_points = torch.zeros(10, 2).double()

short\_points = torch.ones(10, 2).short()

或者用更方便的to方法：

double\_points = torch.zeros(10, 2).to(torch.double)

short\_points = torch.ones(10, 2).to(dtype=torch.short)

在实现内部，type和to执行相同的操作，即“检查类型如果需要就转换（check-and-convert-if-needed）”，但是to方法可以使用其他参数。

你始终可以使用type方法将一种类型的张量转换为另一种类型的张量：

points = torch.randn(10, 2)

short\_points = points.type(torch.short)

上例的randn返回一个符合均值为0，方差为1的正态分布（标准正态分布）中填充随机数的张量。

Torch.rand()返回一个张量，包含[0,1]均匀分布中抽取的一组随机数。

索引张量：以行为索引

some\_list = list(range(6))

some\_list[:] # 所有元素

some\_list[1:4] # 第1（含）到第4（不含）个元素

some\_list[1:] # 第1（含）个之后所有元素

some\_list[:4] # 第4（不含）个之前所有元素

some\_list[:-1] # 最末尾（不含）元素之前所有元素

some\_list[1:4:2] # 范围1（含）到4（不含），步长为2的元素

对张量索引：

points[1:] # 第1行及之后所有行，（默认）所有列

points[1:, :] # 第1行及之后所有行，所有列

points[1:, 0] # 第1行及之后所有行，仅第0列

与numpy互通：

points = torch.ones(3, 4)

points\_np = points.numpy()

points\_np

序列化张量：动态创建张量是很不错的，但是如果其中的数据对你来说具有价值，那么你可能希望将其保存到文件中并在某个时候加载回去。毕竟你可不想每次开始运行程序时都从头开始重新训练模型。

将points张量保存到ourpoints.t文件中:

torch.save(points, '../../data/chapter2/ourpoints.t')

或者，你也可以传递文件描述符代替文件名：

with open('../../data/chapter2/ourpoints.t','wb') as f:

torch.save(points, f)

将points加载回来也是一行类似代码：

points = torch.load('../../data/chapter2/ourpoints.t')

等价于

with open('../../data/chapter2/ourpoints.t','rb') as f:

points = torch.load(f)

Python通过h5py库支持HDF5，该库以NumPy数组的形式接收和返回数据。

将张量转移到GPU上运行：

设备（device）的概念，这是在设置计算机上放张量（tensor）数据的位置。 通过为构造函数指定相应的参数，可以在GPU上创建张量：

points\_gpu = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 4.0]], device='cuda')

你可以使用to方法将在CPU上创建的张量（tensor）复制到GPU：

points\_gpu = points.to(device='cuda')

这段代码返回一个具有相同数值数据的新张量，但存储在GPU的RAM中，而不是常规的系统RAM中。

现在数据已经存放在本地的GPU中，当在张量上运行数字运算时，你可以看见很好的加速效果。并且，这个新GPU支持的张量的类也更改为torch.cuda.FloatTensor（一开始输入的类型为torch.FloatTensor；torch.cuda.DoubleTensor等等也存在对应关系）。在大部分样例中，基于CPU和GPU的张量都公开面向用户相同的API，这使得与繁琐数字运算过程中无关的代码的编写更加容易。

如果你的机器拥有多个GPU，你可以通过传递从零开始的整数来确定张量分配给哪个GPU，该整数标志着机器上的GPU下标：

points\_gpu = points.to(device='cuda:0')

此时，在GPU上执行对张量的任何操作，例如将所有元素乘以一个常数。

points = 2 \* points # 在CPU上做乘法

points\_gpu = 2 \* points.to(device='cuda') # 在GPU上做乘法

请注意，当计算结果产生后，points\_gpu的张量并不会返回到CPU。这里发生的是以下三个过程：

将points张量复制到GPU

在GPU上分配了一个新的张量，并用于存储乘法的结果

返回该GPU张量的句柄

因此，如果你还想向结果加上一个常量：

points\_gpu = points\_gpu + 4

加法仍然在GPU上执行，并且没有信息流到CPU（除非您打印或访问得到的张量）。 如果要将张量移回CPU，你需要为to方法提供一个cpu参数：

points\_cpu = points\_gpu.to(device='cpu')

你可以使用速记方法cpu和cuda代替to方法来实现相同的目标

points\_gpu = points.cuda() # 默认为GPU0

points\_gpu = points.cuda(0)

points\_cpu = points\_gpu.cpu()

值得一提的是，使用to方法时，可以通过提供device和dtype参数来同时更改位置和数据类型。

张量API：

* torch.transpose(Tensor,dim0,dim1)是pytorch中的ndarray矩阵进行转置的操作

注意：transpose()一次只能在两个维度间进行转置（也可以理解为维度转换）

* torch.t()是一个类似于求矩阵的转置的函数，但是它要求输入的tensor结构维度<=2D。 当input维度为0D或者1D时，不做改变输出本身(torch version=1.0.1中会报错)，维度为2D时，输出维度的转置。

从list(range(9))创建一个张量a：

a = torch.tensor(list(range(9)))  
print(a)

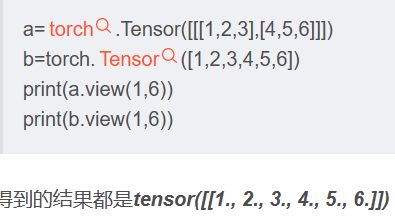
检查尺寸、偏移和步长：

a.size() a.storage\_offset() a.stride()

view(参数a,参数b,参数c)：重构/调整张量维度：



把原先tensor中的数据按照行优先的顺序排成一个一维的数据（这里应该是因为要求地址是连续存储的），然后按照参数组合成其他维度的tensor。比如说是不管你原先的数据是[[[1,2,3],[4,5,6]]]还是[1,2,3,4,5,6]，因为它们排成一维向量都是6个元素，所以只要view后面的参数一致，得到的结果都是一样的。



参数不可为空。参数中的-1就代表这个位置由其他位置的数字来推断，只要在不致歧义的情况的下，view参数就可以推断出来，也就是人可以推断出形状的情况下，view函数也可以推断出来。比如a tensor的数据个数是6个，如果view（1，-1），我们就可以根据tensor的元素个数推断出-1代表6。而如果是view（-1，-1，2），人不知道怎么推断，机器也不知道。还有一种情况是人可以推断出来，但是机器推断不出来的：view（-1，-1，6），人可以知道-1都代表1，但是机器不允许同时有两个负1。

如果没有-1，那么所有参数的乘积就要和tensor中元素的总个数一致了，否则就会出现错误。

“sys模块”：

\_\_file\_\_ #当前执行文件所在的绝对路径

Sys.path #当前执行文件下所有的路径

Sys.argv #当前执行文件所在的绝对路径，列表的形式

Sys.path.append(路径) #添加路径到当前的文件下

“os模块”：

Os.path.abspath() 规范化路径

Os.path.dirname(‘E:/text1/xxx/xxx1.py’) 获取路径：E:text1/xxx

Os.path.basename(‘E:/text1/xxx/xxx1.py’) 获取文件名：xxx1.py

Os.path.join(path1,path2) 路径拼接

Os.path.exists(path) 判断路径是否存在



/ ：表示当前路径的根路径。

./ ：表示当前路径。

../ ：表示父级路径，当前路径所在的上一级路径。

**表格数据**



comments：comments参数指定读取中，跳过以comments参数开头的行，如：类似linux系统的配置文件中用于说明的行，一般用“#”开头，我们就可以用comments参数进行跳过。

delimiter：该参数类似于python处理字符串分割中的split()函数，对数据**根据参数进行分割**处理。例如fname参数中的文件test\_csv.csv，csv文件默认是以逗号分隔数据，在查看文件时没有显示，当没有使用delimiter参数进行分割时，默认是将整个数据一起输出。



skiprows：指定读取时忽略的行数，默认从首行开始计数，也就是说，当我们设置skiprows=1时，列名就会被跳过不读取

next()返回迭代器的下一个项目：



如果分数纯粹是定性的（例如颜色），则独热编码更适合，因为它不涉及隐含的顺序或距离关系。当整数之间的分数值（例如2.4）对应用没有意义时（即要么是这个值要么是那个值），单热编码才适用。

One-Hot

Scatter()不会直接修改原来的tensor，scatter\_()在原来的基础上对tensor进行修改

A.scatter(dim,index,B)

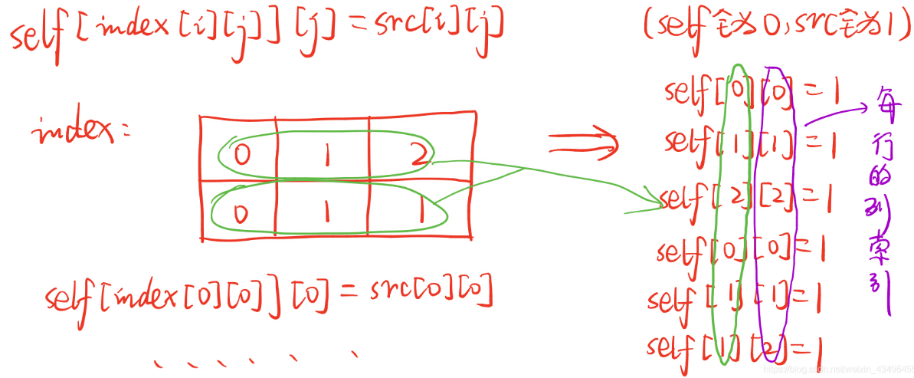
Dim:沿着哪个维度进行索引

Index:用来scatter的元素索引

Src:用来scatter的源元素，可以是一个标量（scalar，一个单独的数）或一个张量

功能：将B按照dim,index填充到A中：



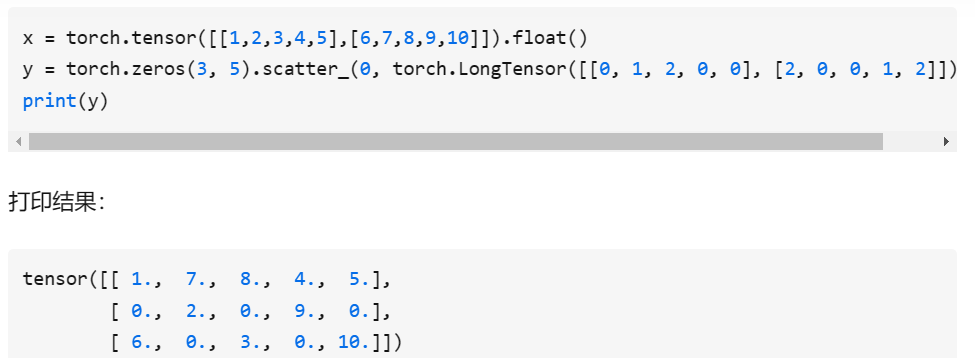


Scr: source（缩写），指向外部资源的位置，指向的内容将会应用到文档中当前标签所在位置。用于替换当前的。

上图中src指x，self指当前的，即torch.zeros(3,3)。此处“每行的列索引”以self为基准

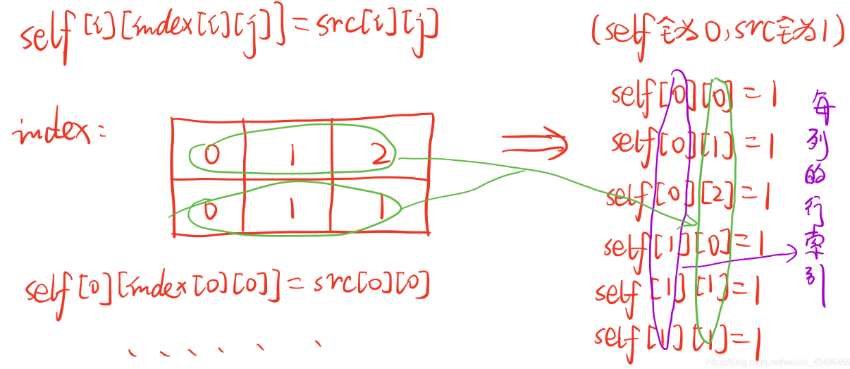
Dim=0：按行填充：

例子：



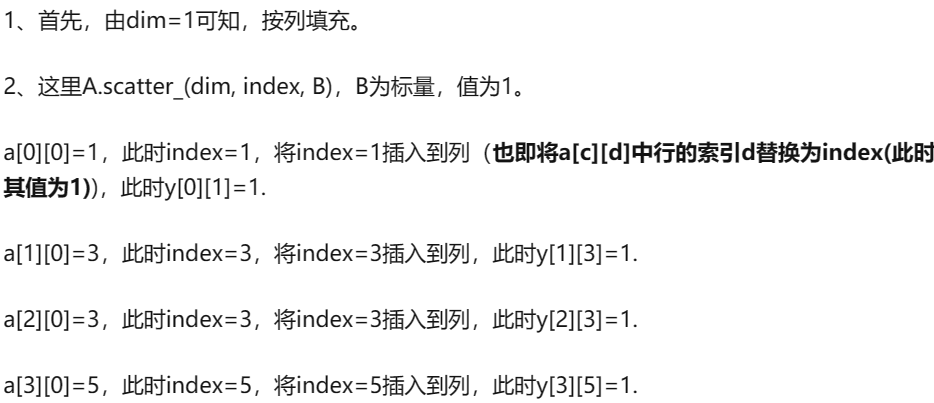


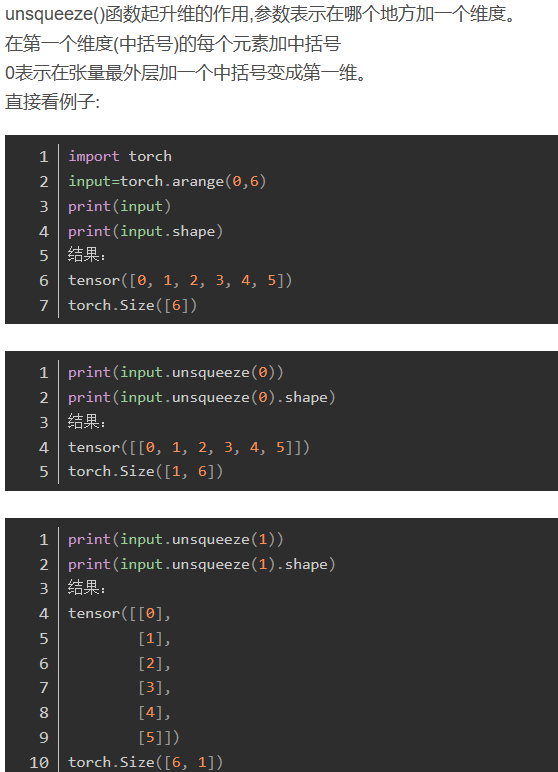




独热编码例子：







scatter\_的参数是：

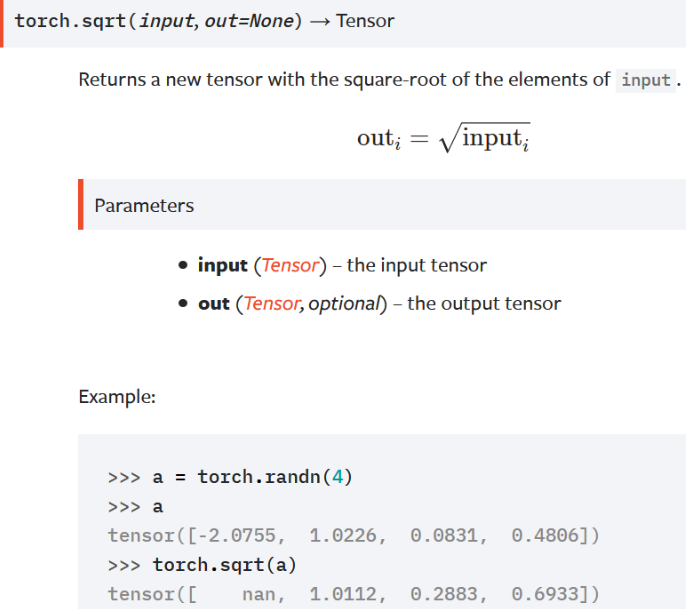
* 指定后面两个参数所处理的维度
* 列张量，指示要填充的索引：索引张量，必须具有与待填充张量相同的维数。由于target\_onehot是二维（4898x10）的，因此你需要使用unsqueeze为target添加一个额外的维，即target.unsqueeze(1)
* 包含填充元素的张量或者单个标量（上例中即1.0）

Torch.var(data,dim):方差（variance）

标准差：standard vaerance，方差的算数平方根

mean()函数的参数：dim=0,沿维度为0进行计算，即按列求平均值，返回的形状是（1，列数）；dim=1,按行求平均值，返回的形状是（行数，1）,默认不设置dim的时候，返回的是所有元素的平均值。

Torch.sqrt(input,out=none)





Torch.gt() input > other, if true,output1,if false,output0

Torch.le() input<=other

Torch.lt() input < other

torch.le

torch.le(input, other, out=None) → Tensor

逐元素比较input和other ， 即是否input<=otherinput<=other 第二个参数可以为一个数或与第一个参数相同形状和类型的张量

参数:

input (Tensor) – 要对比的张量

other (Tensor or float ) – 对比的张量或float值

out (Tensor, optional) – 输出张量。必须为ByteTensor或者与第一个参数tensor相同类型。

返回值： 一个 torch.ByteTensor 张量，包含了每个位置的比较结果(是否 input >= other )。 返回类型： Tensor

例子：

>>> torch.le(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))

1 0

1 1

[torch.ByteTensor of size 2x2]

torch.lt

torch.lt(input, other, out=None) → Tensor

逐元素比较input和other ， 即是否 input<otherinput<other

第二个参数可以为一个数或与第一个参数相同形状和类型的张量

参数:

input (Tensor) – 要对比的张量

other (Tensor or float ) – 对比的张量或float值

out (Tensor, optional) – 输出张量。必须为ByteTensor或者与第一个参数tensor相同类型。

input： 一个 torch.ByteTensor 张量，包含了每个位置的比较结果(是否 tensor >= other )。 返回类型： Tensor

例子：

>>> torch.lt(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))

0 0

1 0

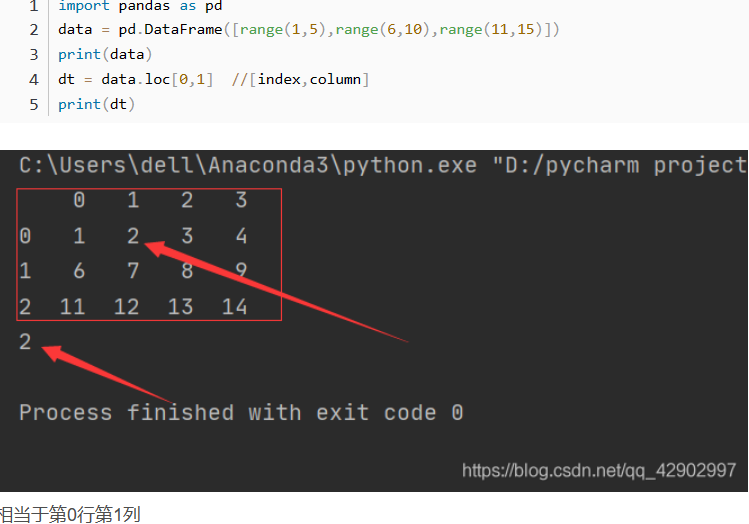
[torch.ByteTensor of size 2x2]

通过利用PyTorch中称为高级索引（advanced indexing）的功能，可以使用0/1张量来索引数据张量。此张量本质上将数据筛选为仅与索引张量中的1对应的元素（或行）。

[ ] 索引

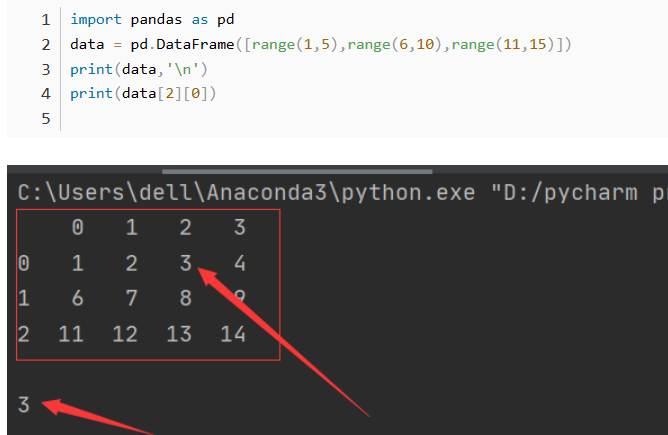
（1）data.loc[index,column]

使用.loc[ ]第一个参数是行索引，第二个参数是列索引



（2）data[column][index]

这里与上面不同，使用两个方括号的索引方式，列标签的优先级更高一些，是列在前行在后。



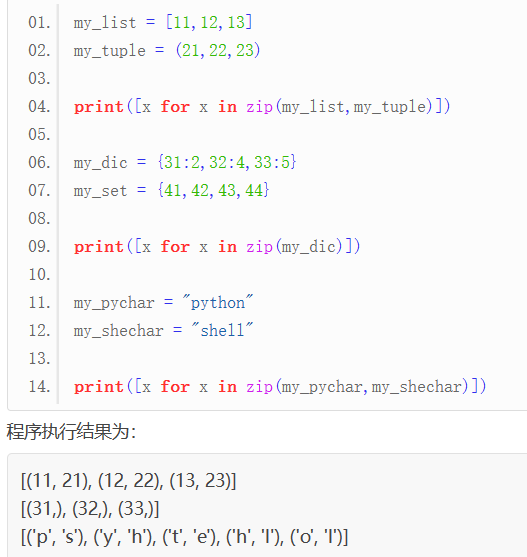
zip([iterable, …] )函数：zip()函数的定义：从参数中的多个迭代器取元素组合成一个新的迭代器；

返回： 返回一个zip对象，其内部元素为元组；可以转化为列表或元组；

传入参数： 元组、列表、字典等迭代器。

当zip()函数中只有一个参数时，zip(iterable)从iterable中依次取一个元组，组成一个元组。

iterable:一个或多个迭代器



在使用 zip() 函数“压缩”多个序列时，它会分别取各序列中第 1 个元素、第 2 个元素、... 第 n 个元素，各自组成新的元组。需要注意的是，当多个序列中元素个数不一致时，会以最短的序列为准进行压缩。

Next()和csv.reader()合用表示选取第一行标题行：

python中有个csv包（build-in），该包有个reader，按行读取csv文件中的数据

reader.next()作用：打印csv文件中的第一行标题header

allElectronicsData = open(r'C:\Users\Lenovo\Desktop\AllElectronics.csv','rt')

#打开这个csv文件储存到allElectronicsData

reader = csv.reader(allElectronicsData)

#reader = csv.reader(f) 此时reader返回的值是csv文件中每行的列表，将每行读取的值作为列表返回,此时reader是一个列表

headers = next(reader)

#python中有个csv包（build-in），该包有个reader，按行读取csv文件中的数据，也就是读取列表中的数据

Enumerate()：用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标，一般用在 for 循环当中



**\*args**的用法：当传入的参数个数未知，且**不需要知道参数名称**时。

def func\_arg(farg, \*args):

print("formal arg:", farg)

for arg in args:

print("another arg:", arg)

func\_arg(1,"youzan",'dba','四块五的妞')

print("-----------------------")

# 输出结果如下：

# formal arg: 1

# another arg: youzan

# another arg: dba

# another arg: 四块五的妞

# -----------------------

#\*\*args的用法：当传入的参数个数未知，但需要知道参数的名称时(立马想到了字典，即键值对)

def func\_kwargs(farg, \*\*kwargs):

print("formal arg:", farg)

for key in kwargs:

print("keyword arg: %s: %s" % (key, kwargs[key]))

func\_kwargs(1 ,id=1, name='youzan', city='hangzhou',age ='20',四块五的妞是 = '来日方长的')

print('--------------------')

# 输出结果如下：

# formal arg: 1

# keyword arg: id: 1

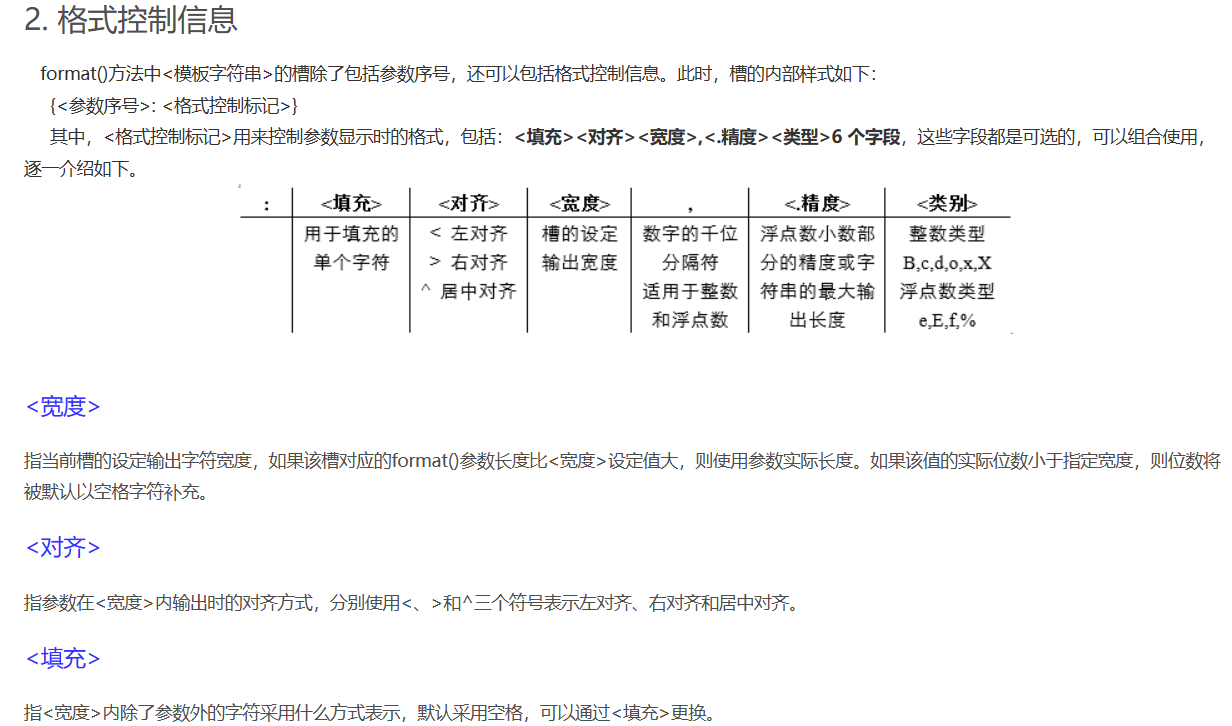
# keyword arg: name: youzan

# keyword arg: city: hangzhou

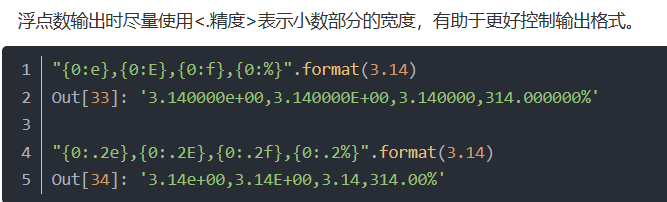
# keyword arg: age: 20

# keyword arg: 四块五的妞是: 来日方长的

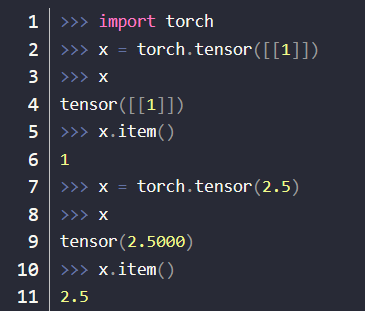
Format()用法：







.item()的用法：返回张量元素的值（当张量元素中只有一个才可以调用）



按位与运算符（&）

参加运算的两个数据，按二进制位进行“与”运算。

运算规则：0&0=0; 0&1=0; 1&0=0; 1&1=1;

即：两位同时为“1”，结果才为“1”，否则为0

Converters():目的将数据源文件中的某列数据替换成目标数据

Converters( : )

Lambda函数：

lambda [arg1 [,arg2,.....argn]] : expression

其中，lambda 是 Python 预留的关键字，[arg…] 和 expression 由用户自定义。

具体介绍如下:

[arg…] 是参数列表，它的结构与 Python 中函数(function)的参数列表是一样的。

lambda x, y: x\*y # 函数输入是x和y，输出是它们的积x\*y

lambda:None # 函数没有输入参数，输出是None

lambda \*args: sum(args) # 输入是任意个数参数，输出是它们的和(隐性要求输入参数必须能进行算术运算)

lambda \*\*kwargs: 1 # 输入是任意键值对参数，输出是1



Map( function,iterable,… )，

# ===========一般写法：===========

# 1、计算平方数

def square(x):

return x \*\* 2

map(square, [1,2,3,4,5]) # 计算列表各个元素的平方

# 结果：

[1, 4, 9, 16, 25]

# ===========匿名函数写法：============

# 2、计算平方数，lambda 写法

map(lambda x: x \*\* 2, [1, 2, 3, 4, 5])

# 结果：

[1, 4, 9, 16, 25]

# 3、提供两个列表，将其相同索引位置的列表元素进行相加

map(lambda x, y: x + y, [1, 3, 5, 7, 9], [2, 4, 6, 8, 10])

# 结果：

[3, 7, 11, 15, 19]

reduce() 函数：

描述：

reduce() 函数会对参数序列中元素进行累积。

函数将一个数据集合（链表，元组等）中的所有数据进行下列操作：用传给 reduce 中的函数 function（有两个参数）先对集合中的第 1、2 个元素进行操作，得到的结果再与第三个数据用 function 函数运算，最后得到一个结果。

# ===========一般写法：===========

# 1、两数相加

def add(x, y):

return x + y

reduce(add, [1, 3, 5, 7, 9]) # 计算列表元素和：1+3+5+7+9

# 结果：

25

"""

===========执行步骤解析：===========

调用 reduce(add, [1, 3, 5, 7, 9])时，reduce函数将做如下计算：

1 先计算头两个元素：f(1, 3)，结果为4；

2 再把结果和第3个元素计算：f(4, 5)，结果为9；

3 再把结果和第4个元素计算：f(9, 7)，结果为16；

4 再把结果和第5个元素计算：f(16, 9)，结果为25；

5 由于没有更多的元素了，计算结束，返回结果25。

"""

# ===========匿名函数写法：===========

# 2、两数相加，lambda 写法

reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])

# 结果：

15

# 当然求和运算可以直接用Python内建函数sum()，没必要动用reduce。

# 3、但是如果要把序列 [1, 3, 5, 7, 9] 变换成整数 13579，reduce就可以派上用场：

from functools import reduce

def fn(x, y):

return x \* 10 + y

reduce(fn, [1, 3, 5, 7, 9])

# 结果：

13579

sorted() 函数：

描述：

sorted() 函数对所有可迭代的对象进行排序操作。

sort 与 sorted 区别：

sort 是 list 的一个方法，而 sorted 可以对所有可迭代的对象进行排序操作。

list 的 sort 方法返回的是对已经存在的列表进行操作，无返回值，而内建函数 sorted 方法返回的是一个新的 list，而不是在原来的基础上进行的操作。

语法：

sorted(iterable[, cmp[, key[, reverse]]])

参数说明：

iterable ----> 可迭代对象。

cmp ----> 比较的函数，这个具有两个参数，参数的值都是从可迭代对象中取出，此函数必须遵守的规则为，大于则返回1，小于则返回-1，等于则返回0。

key ----> 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序。

reverse ----> 排序规则，reverse = True 降序 ， reverse = False 升序（默认）。

返回值：

返回重新排序的列表。

# ===========一般用法：===========

# 1、简单排序

a = [5,7,6,3,4,1,2]

b = sorted(a) # 使用sorted，保留原列表，不改变列表a的值

print(a)

[5, 7, 6, 3, 4, 1, 2]

print(b)

[1, 2, 3, 4, 5, 6, 7]

# ===========匿名函数用法：===========

L=[('b',2),('a',1),('c',3),('d',4)]

# 2、利用参数 cmp 排序

sorted(L, cmp=lambda x,y:cmp(x[1],y[1]))

# 结果：

[('a', 1), ('b', 2), ('c', 3), ('d', 4)]

# 3、利用参数 key 排序

sorted(L, key=lambda x:x[1])

# 结果：

[('a', 1), ('b', 2), ('c', 3), ('d', 4)]

# 4、按年龄升序

students = [('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]

sorted(students, key=lambda s: s[2])

# 结果：

[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

# 5、按年龄降序

sorted(students, key=lambda s: s[2], reverse=True)

# 结果：

[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]

Filter()

# ===========一般用法：===========

# 1、过滤出列表中的所有奇数

def is\_odd(n):

return n % 2 == 1

newlist = filter(is\_odd, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

print(list(newlist))

# 结果： [1, 3, 5, 7, 9]

# ===========匿名函数用法：===========

# 2、将列表[1, 2, 3]中能够被3整除的元素过滤出来

newlist = filter(lambda x: x % 3 == 0, [1, 2, 3])

print(list(newlist))

# 结果： [3]

1.0000e+00: 1\*10的(e后面零的个数次幂)

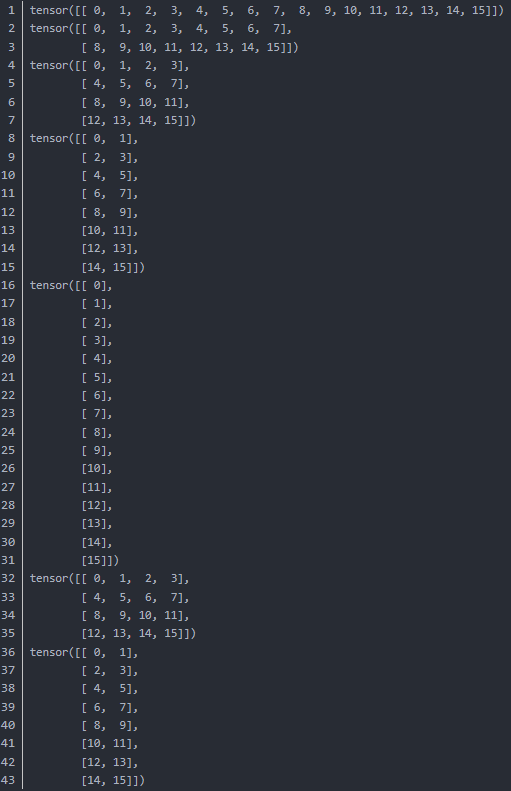
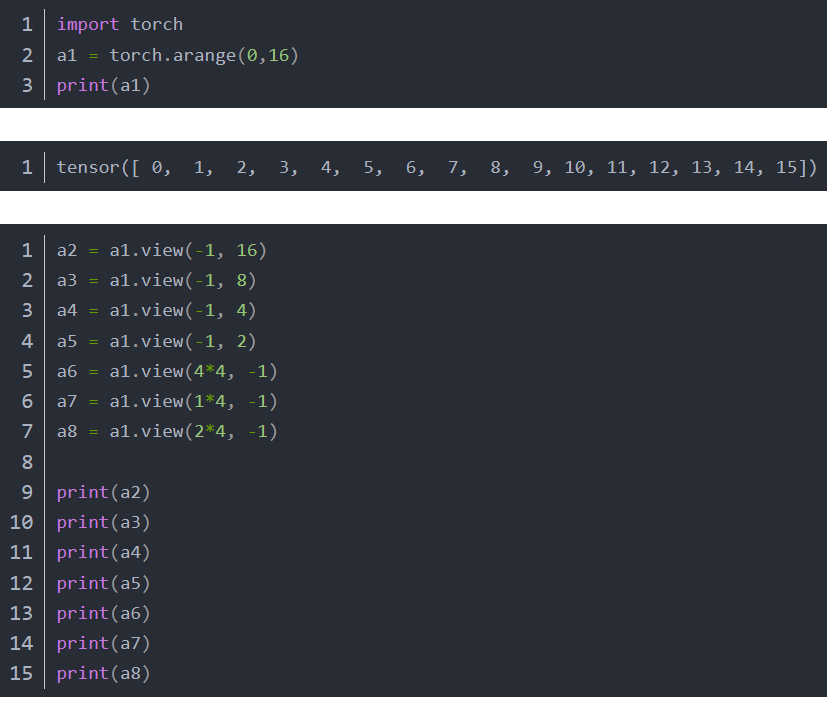
1E+05=10^5=100000

1E-03=10^(-3)=0.001

View()相当于numpy中resize()，用法有点不一样：把原先tensor中的数据按照行优先的顺序排成一个一维的数据（这里应该是因为要求地址是连续存储的），然后按照参数组合成其他维度的tensor。比如说是不管你原先的数据是[[[1,2,3],[4,5,6]]]还是[1,2,3,4,5,6]，因为它们排成一维向量都是6个元素，所以只要view后面的参数一致，得到的结果都是一样的。

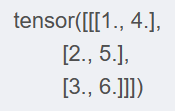


View中一个参数定为-1，自动调整该维度上的元素个数，以保证元素的总数不变。



Permute(dims): 将tensor的维度换位。比如图片img的size比如是（28，28，3）就可以利用img.permute(2,0,1)得到一个size为（3，28，28）的tensor。

利用这个函数permute（0，2，1）可以把Tensor([[[1,2,3],[4,5,6]]]) 转换成



x[:, n]表示在全部数组（维）中取第n个数据，直观来说，x[:,n]就是取所有集合的第n个数据,

x[n, :]表示在n个数组（维）中取全部数据，直观来说，x[n,:]就是取第n集合的所有数据,

无论是左边还是右边逗号都要靠近冒号：

如果冒号：的左边或者右边还有冒号，这时候就说明其中一个冒号代表的是范围（eg:1:5 从１到４）

如果冒号：左边或者右边没有任何东西，那么这时候代表全体

[a:b] 对ａ的改变是行的改变，对ｂ的改变是队列的改变

对于X[:,0];

是取二维数组中第一维的所有数据

对于X[:,1]

是取二维数组中第二维的所有数据

对于X[:,m:n]

是取二维数组中第m维到第n-1维的所有数据

对于X[:,:,0]

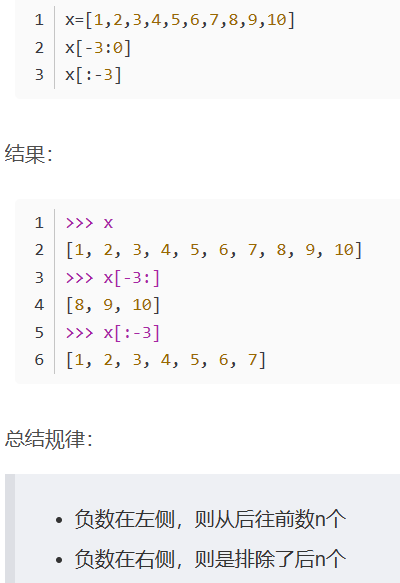
是取三维矩阵中第一维的所有数据

对于X[:,:,1]

是取三维矩阵中第二维的所有数据

对于X[:,:,m:n]

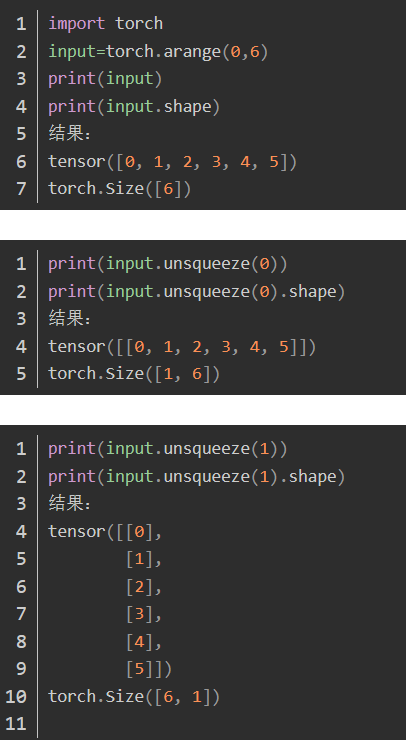
是取三维矩阵中第m维到第n-1维的所有数据

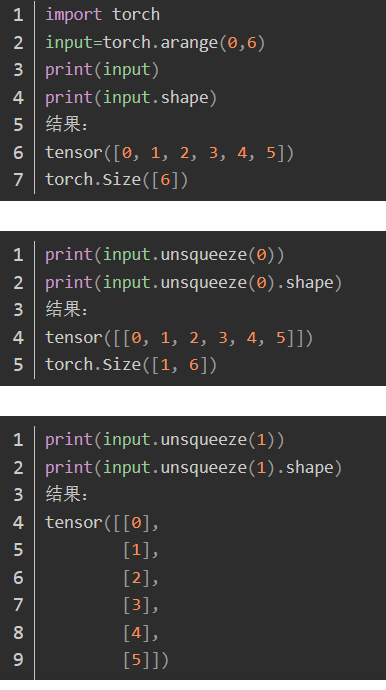


unsqueeze()函数起升维的作用,参数表示在哪个地方加一个维度。

在第一个维度(中括号)的每个元素加中括号

0表示在张量最外层加一个中括号变成第一维。





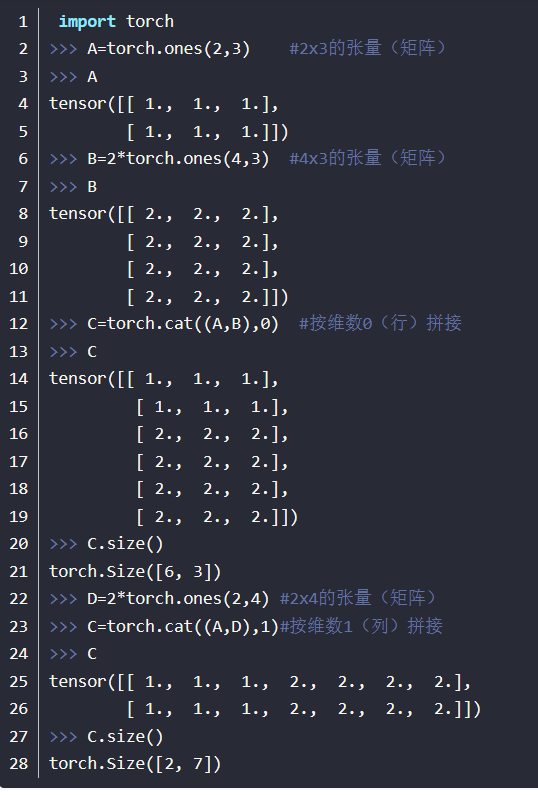
Cat( ):

1. 字面理解：torch.cat是将两个张量（tensor）拼接在一起，cat是concatnate的意思，即拼接，联系在一起。

2. 例子理解

C = torch.cat( (A,B),0 ) #按维数0拼接（竖着拼）

C = torch.cat( (A,B),1 ) #按维数1拼接（横着拼）



上面给出了两个张量A和B，分别是2行3列，4行3列。即他们都是2维张量。因为只有两维，这样在用torch.cat拼接的时候就有两种拼接方式：按行拼接和按列拼接。即所谓的维数0和维数1.

C=torch.cat((A,B),0)就表示按维数0（行）拼接A和B，也就是竖着拼接，A上B下。此时需要注意：列数必须一致，即维数1数值要相同，这里都是3列，方能列对齐。拼接后的C的第0维是两个维数0数值和，即2+4=6.

C=torch.cat((A,B),1)就表示按维数1（列）拼接A和B，也就是横着拼接，A左B右。此时需要注意：行数必须一致，即维数0数值要相同，这里都是2行，方能行对齐。拼接后的C的第1维是两个维数1数值和，即3+4=7.

从2维例子可以看出，使用torch.cat((A,B),dim)时，除拼接维数dim数值可不同外其余维数数值需相同，方能对齐。

3.实例

在深度学习处理图像时，常用的有3通道的RGB彩色图像及单通道的灰度图。张量size为cxhxw,即通道数x图像高度x图像宽度。在用torch.cat拼接两张图像时一般要求图像大小一致而通道数可不一致，即h和w同，c可不同。当然实际有3种拼接方式，另两种好像不常见。比如经典网络结构：U-Net

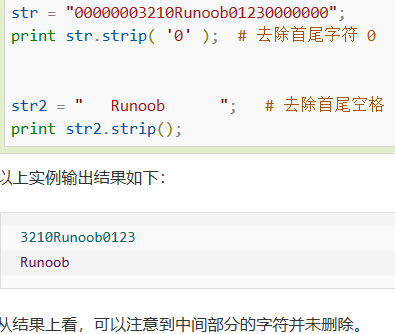
里面用到4次torch.cat,其中copy and crop操作就是通过torch.cat来实现的。可以看到通过上采样（up-conv 2x2）将原始图像h和w变为原来2倍，再和左边直接copy过来的同样h,w的图像拼接。这样做，可以有效利用原始结构信息。

4.总结

使用torch.cat((A,B),dim)时，除拼接维数dim数值可不同外其余维数数值需相同，方能对齐。

Str.strip([chars])

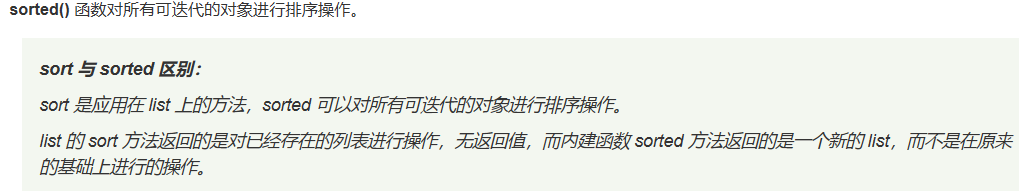
Chars——移除字符串头尾指定的字符序列。默认为空格



Ord( )函数：

chr() 函数（对于8位的ASCII字符串）或 unichr() 函数（对于Unicode对象）的配对函数，它以一个字符（长度为1的字符串）作为参数，返回对应的 ASCII 数值，或者 Unicode 数值，如果所给的 Unicode 字符超出了你的 Python 定义范围，则会引发一个 TypeError 的异常。

Sorted( )



sorted(iterable, cmp=None, key=None, reverse=False)

参数说明：

iterable -- 可迭代对象。

cmp -- 比较的函数，这个具有两个参数，参数的值都是从可迭代对象中取出，此函数必须遵守的规则为，大于则返回1，小于则返回-1，等于则返回0。

key -- 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序。

reverse -- 排序规则，reverse = True 降序 ， reverse = False 升序（默认）。



set() 函数创建一个无序不重复元素集，可进行关系测试，删除重复数据，还可以计算交集、差集、并集等。



如何将编码压缩为更易于管理的大小，并限制大小增长？好吧，可以使用浮点数向量，而不是使用多个0和一个1的向量。举例来说，一个含100个浮点数的向量就可以表示很大量的词汇。关键是找到一种有效的方法，以一种有助于下游学习的方式将单个单词映射到这个100维空间。这种技术称为嵌入（embedding）。

原则上，你可以遍历词汇表并为每个单词生成100个随机浮点数。 这种方法可能是有效的，因为你可以将大量词汇塞入100个数字中，但是它会丢弃掉基于语义或上下文的单词之间的任何距离信息。使用这种词嵌入的模型不得不处理其输入向量中的少量结构。理想的解决方案是以这样的方式生成嵌入：用于同一上下文的单词映射到嵌入空间的邻近区域。

如果要手工设计解决此问题的方法，你有可能决定通过沿轴映射基本名词和形容词来构建嵌入空间。你可以生成一个二维空间，在该空间中，两个坐标轴分别映射到名词“水果”（0.0-0.33）、“花”（0.33-0.66）和“狗”（0.66-1.0），以及形容词“红色”（0.0-0.2）、“橙色”（0.2-0.4）、“黄色”（0.4-0.6）、“白色”（0.6-0.8）和“棕色”（0.8-1.0）。你现在的目标是将水果、花和狗放置在嵌入中。

开始嵌入单词时，可以将“苹果”映射到“水果”和“红色”象限中的某个数。同样，你可以轻松地映射“橘子”、“柠檬”、“荔枝”和“猕猴桃”（五颜六色的水果）。然后，你可以从花开始，分配“玫瑰”、“罂粟”、“水仙花”、“百合”和...好吧，不存在很多棕色的花。好，“太阳花”可以推出“花”、“黄色”和“棕色”，而“雏菊”可以推出“花”、“白色”和“黄色”。也许你应该更新“猕猴桃”以将其映射到“水果”、“棕色”和“绿色”附近。对于狗和颜色，“redbone（译者注：狗的品种）”、“fox”可能是“橙色”、“金毛”和“贵宾犬”可是“白色”的，以及...大多数种类的狗都是“棕色”的。

尽管对于大型语料库而言，手动进行此映射并不可行，但你应注意，尽管嵌入大小仅为2，但你描述了除基数8个之外的15个不同的单词，如果你花一些创造性的时间，可能还会嵌入更多的单词。

你可能已经猜到了，这种工作是可以自动进行的。通过处理大量文本语料库，你可以生成与此类似的嵌入。主要区别在于嵌入向量具有100到1000个元素，并且坐标轴不直接映射到某个词义，但是意思相近的词映射到嵌入空间也是相近的，其轴可能是任意的浮点维（floating-point dimensions）。

尽管实际使用的算法（比如word2vec）对于我们在此要关注的内容来说有点超出范围，但值得一提的是，嵌入通常是使用神经网络并试图根据句中邻近词（上下文）预测某个词而生成的。在这种情况下，你可以从独热编码的单词开始，使用（通常是相当浅的）神经网络来生成嵌入。当嵌入可用时，你就可以将其用于下游任务。

生成的嵌入的一个有趣的方面是，相似的词不仅会聚在一起，还会与其他词保持一致的空间关系。如果你要使用“苹果”的嵌入向量，并加上和减去其他词的嵌入向量，就可以进行类比，例如苹果 - 红色 - 甜 + 酸，最后可能得到一个类似“柠檬”的向量。

KNN和Kmeans区别：

KNN（K nearest Neighbors）当预测一个新的值x，根据它距离最近的k个点是什么类别来判断x属于哪个类别-属于监督学习，类别已知，通过对已知分类的数据进行训练和学习，找到不同类的特征，再对未分类的数据进行分类。

Kmeans属于非监督学习，通过聚类分析将数据聚合成几个群体。聚类不需要对数据进行训练和学习

imageio. volread (uri, format = None, \*\* kwargs) ：Reads a volume from the specified file.

用jupyter notebook打开指定盘符：anaconda prompt里先D:，再jupyter notebook

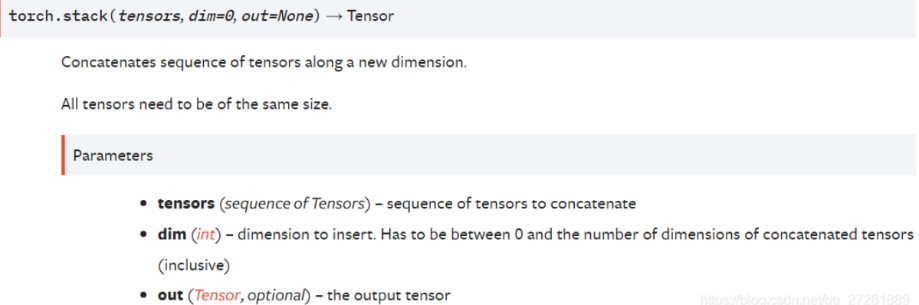
在jupyter notebook里安装包：! pip install xxx

运行的快捷键：ctrl+enter

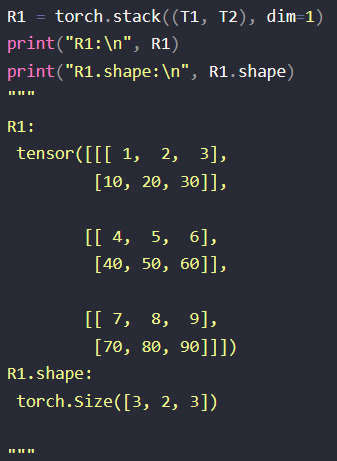
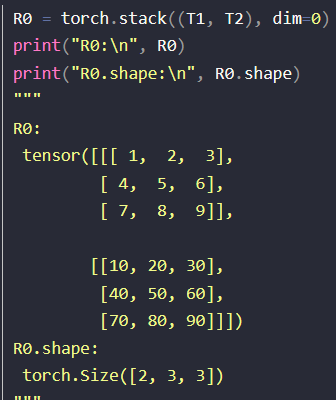
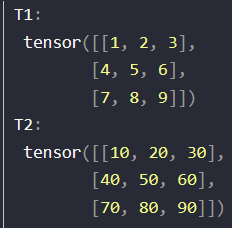
Torch.stack( ) 拼接函数

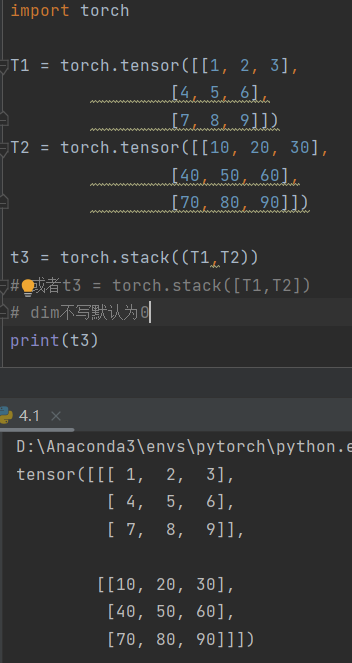
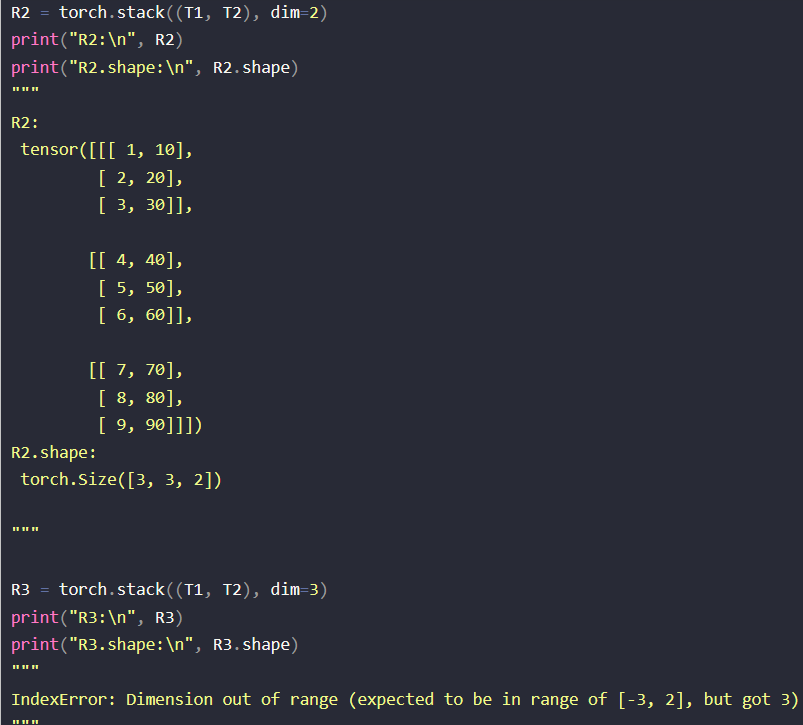
沿着一个新维度对输入张量序列进行连接。 序列中所有的张量都应该为相同形状。

即：把多个2维的张量凑成一个3维的张量；多个3维的凑成一个4维的张量…以此类推，也就是在增加新的维度进行堆叠。









输入的是tensors不是tensor。

Tensors可以认为是tensor的列表或者是tensor的元组。

（1）iteration：表示1次迭代（也叫training step），每次迭代更新1次网络结构的参数；

（2）batch-size：1次迭代所使用的样本量；

（3）epoch：1个epoch表示过了1遍训练集中的所有样本。

值得注意的是，在深度学习领域中，常用带mini-batch的随机梯度下降算法（Stochastic Gradient Descent, SGD）训练深层结构，它有一个好处就是并不需要遍历全部的样本，当数据量非常大时十分有效。此时，可根据实际问题来定义epoch，例如定义10000次迭代为1个epoch，若每次迭代的batch-size设为256，那么1个epoch相当于过了2560000个训练样本。

跑完整个训练数据集，叫做一个epoch。一个epoch包含多个episode。一个episode完成一次模型验证，保存最优模型，简单来说就是多少个step进行一次模型验证。假设整个训练数据集有n=10,000个样本，batch\_size=10，那么一个epoch就包含10,000/10=1,000个step（或iteration）。假设episodes=100，即一个episode包含100个step，那么一个epoch就包含1,000/100=10个episode。每一个episode完成后，进行一次模型验证，并保存模型（一般模型性能没有提升，则不保存）。



Epoch（时期）：

当一个完整的数据集通过了神经网络一次并且返回了一次，这个过程称为一次>epoch。（也就是说，所有训练样本在神经网络中都 进行了一次正向传播 和一次反向传播 ）

再通俗一点，一个Epoch就是将所有训练样本训练一次的过程。

然而，当一个Epoch的样本（也就是所有的训练样本）数量可能太过庞大（对于计算机而言），就需要把它分成多个小块，也就是就是分成多个Batch 来进行训练。\*\*

Batch（批 / 一批样本）：

将整个训练样本分成若干个Batch。

Batch\_Size（批大小）：

每批样本的大小。

Iteration（一次迭代）：

训练一个Batch就是一次Iteration（这个概念跟程序语言中的迭代器相似）。

mnist 数据集有张图片作为训练数据，张图片作为测试数据。假设现在选择 Batch\_Size = 对模型进行训练。迭代次。

每个 Epoch 要训练的图片数量：(训练集上的所有图像)

训练集具有的 Batch 个数：

每个 Epoch 需要完成的 Batch 个数：

每个 Epoch 具有的 Iteration 个数：（完成一个Batch训练，相当于参数迭代一次）

每个 Epoch 中发生模型权重更新的次数：

训练 10 个Epoch后，模型权重更新的次数：

不同Epoch的训练，其实用的是同一个训练集的数据。第1个Epoch和第10个Epoch虽然用的都是训练集的图片，但是对模型的权重更新值却是完全不同的。因为不同Epoch的模型处于代价函数空间上的不同位置，模型的训练代越靠后，越接近谷底，其代价越小。

总共完成30000次迭代，相当于完成了个Epoch

epoch

当一个完整的数据集经过神经网络一次，并返回一次，这个过程称为一个epoch。

为什么需要多个epoch

在深度学习中，向神经网络传递整个数据集一次是远远不够的，而需要多次在神经网络上训练。从欠拟合的状态向适当拟合靠近。当然一不小心也可能会过拟合。也就是说不同的数据集，最适的epoch是不同的，会受到数据集多样性的影响。

batch

当数据集很大的时候，对于每个epoch，很难将所有的数据集一次读入到内存中，这是需要将数据集分为几次读入，每次称为一个batch。

batch size

即batch中样本的数量。

mini-batch

需要先介绍下梯度下降的两种方法。

批梯度下降（batch gradient decent）

这种方法每次使用整个batch计算损失，调整参数。性能相对较好，但是计算量大，速度慢。

随机梯度下降（stochastic gradient decent）

每次选取一个数据调整参数，计算很快，但是收敛性能不好，容易在最优点附近震荡。

小批量梯度下降（mini-batch gradient decent）

现在解释mini-batch。这里指的是一种梯度下降的方法，算是融合了上述两种方法的优点。也就是说把batch分成小batch，在小batch上梯度下降。

Verbose日志显示，三个参数可选：0，1，2



% 1.除法取余数；2.格式化输出

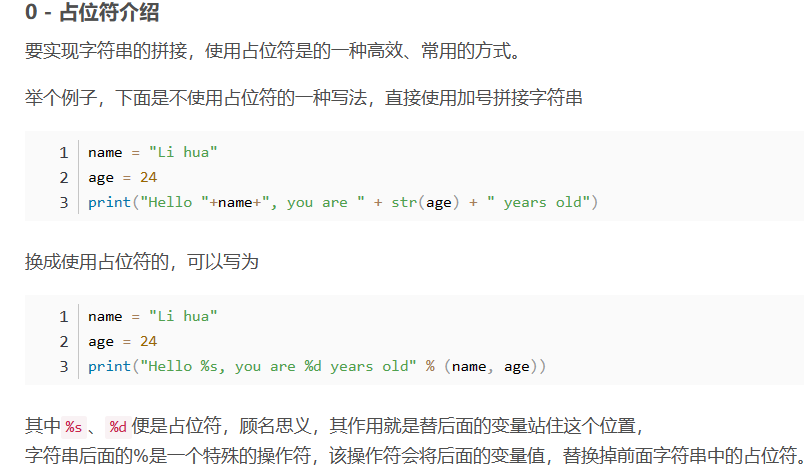
%x --- hex 十六进制

%d --- dec 十进制

%o --- oct 八进制

%g --- 浮点数

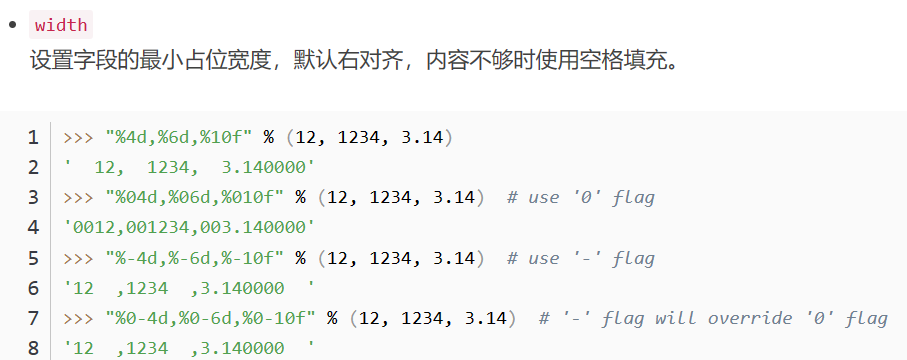
%s ---字符串



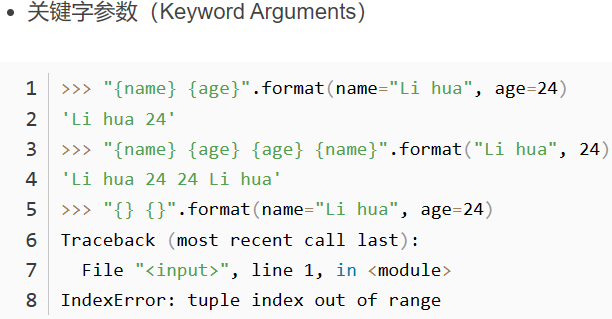
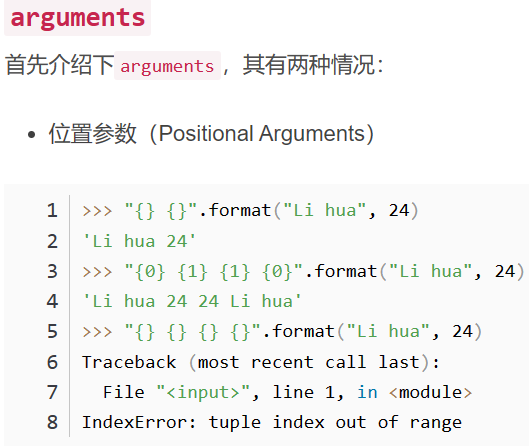








% --format



tensor.detach()

返回一个新的tensor，从当前计算图中分离下来的，但是仍指向原变量的存放位置,不同之处只是requires\_grad为false，得到的这个tensor永远不需要计算其梯度，不具有grad。

即使之后重新将它的requires\_grad置为true,它也不会具有梯度grad

这样我们就会继续使用这个新的tensor进行计算，后面当我们进行反向传播时，到该调用detach()的tensor就会停止，不能再继续向前进行传播

注意：使用detach返回的tensor和原始的tensor共同一个内存，即一个修改另一个也会跟着改变。

Autograd,grad( )

情景：用NN求解PDE（偏微分方程）输出值对输入变量（不是weights和biases）求导；训练WGAN-GP（GAN发展而来）也会用到网络对输入变量的求导。

autograd.grad(outputs, inputs, grad\_outputs=None, retain\_graph=None, create\_graph=False, only\_inputs=True, allow\_unused=False)

outputs: 求导的因变量（需要求导的函数）

inputs: 求导的自变量

grad\_outputs: 如果 outputs为标量，则grad\_outputs=None,也就是说，可以不用写; 如果outputs 是向量，则此参数必须写

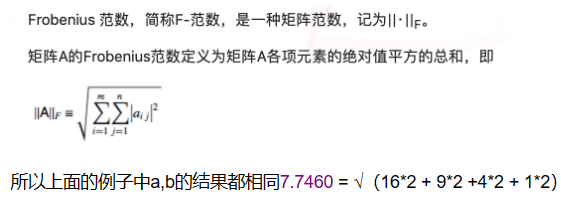
torch.norm(input, p='fro', dim=None, keepdim=False, out=None, dtype=None)

fro:frobenius norm, nuc:nuclear norm

torch.norm() 返回范数3.

1)常用范数p-范数

2)若不指明p，计算Frobenius范数



3)p=’inf’,求各项元素绝对值中的最大值

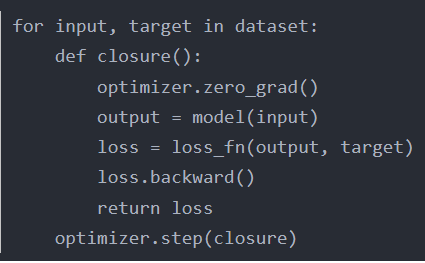
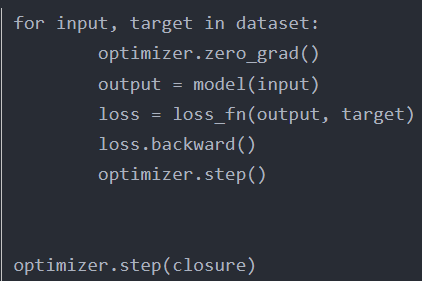
requires\_grad: 如果需要为张量计算梯度，则为True，否则为False。我们使用pytorch创建tensor时，可以指定requires\_grad为True（默认为False），

grad\_fn： grad\_fn用来记录变量是怎么来的，方便计算梯度，y = x\*3,grad\_fn记录了y由x计算的过程。

grad：当执行完了backward()之后，通过x.grad查看x的梯度值。

**grad在反向传播过程中是累加的(accumulated)，这意味着每一次运行反向传播，梯度都会累加之前的梯度，所以一般在反向传播之前需把梯度清零。**

**Optimizer和optimizer.step()**





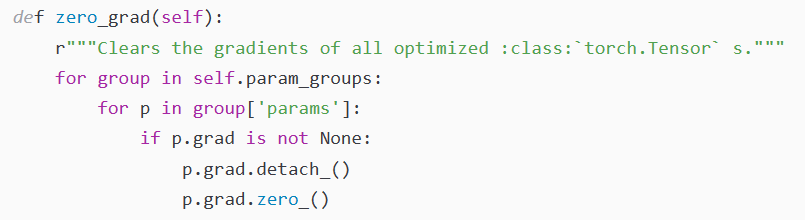
这三个函数的作用是先将梯度归零（optimizer.zero\_grad()），然后反向传播计算得到每个参数的梯度值（loss.backward()），最后通过梯度下降执行一步参数更新（optimizer.step()）

常见的参数变量：

**Param\_groups**: Optimizer类在实例化时会在构造函数中创建一个param\_groups列表，列表中有num\_groups个长度为6的param\_group字典（num\_groups取决于你定义optimizer时传入了几组参数），每个param\_group包含了 ['params', 'lr', 'momentum', 'dampening', 'weight\_decay', 'nesterov'] 这6组键值对。

**param\_group['params']**：由传入的模型参数组成的列表，即实例化Optimizer类时传入该group的参数，如果参数没有分组，则为整个模型的参数model.parameters()，每个参数是一个torch.nn.parameter.Parameter对象。

1. optimizer.zero\_graad()



optimizer.zero\_grad()函数会遍历模型的所有参数，通过p.grad.detach\_()方法截断反向传播的梯度流，再通过p.grad.zero\_()函数将每个参数的梯度值设为0，即上一次的梯度记录被清空。

因为训练的过程通常使用mini-batch方法，所以如果不将梯度清零的话，梯度会与上一个batch的数据相关，因此该函数要写在反向传播和梯度下降之前。

1. loss.backward()

PyTorch的反向传播(即tensor.backward())是通过autograd包来实现的，autograd包会根据tensor进行过的数学运算来自动计算其对应的梯度。

具体来说，torch.tensor是autograd包的基础类，如果你设置tensor的requires\_grads为True，就会开始跟踪这个tensor上面的所有运算，如果你做完运算后使用tensor.backward()，所有的梯度就会自动运算，tensor的梯度将会累加到它的.grad属性里面去。

更具体地说，损失函数loss是由模型的所有权重w经过一系列运算得到的，若某个w的requires\_grads为True，则w的所有上层参数（后面层的权重w）的.grad\_fn属性中就保存了对应的运算，然后在使用loss.backward()后，会一层层的反向传播计算每个w的梯度值，并保存到该w的.grad属性中。

如果没有进行tensor.backward()的话，梯度值将会是None，因此loss.backward()要写在optimizer.step()之前。

1. optimizer.step()

def step(self, closure=None):

"""Performs a single optimization step.

Arguments:

closure (callable, optional): A closure that reevaluates the model

and returns the loss.

"""

loss = None

if closure is not None:

loss = closure()

for group in self.param\_groups:

weight\_decay = group['weight\_decay']

momentum = group['momentum']

dampening = group['dampening']

nesterov = group['nesterov']

for p in group['params']:

if p.grad is None:

continue

d\_p = p.grad.data

if weight\_decay != 0:

d\_p.add\_(weight\_decay, p.data)

if momentum != 0:

param\_state = self.state[p]

if 'momentum\_buffer' not in param\_state:

buf = param\_state['momentum\_buffer'] = torch.clone(d\_p).detach()

else:

buf = param\_state['momentum\_buffer']

buf.mul\_(momentum).add\_(1 - dampening, d\_p)

if nesterov:

d\_p = d\_p.add(momentum, buf)

else:

d\_p = buf

p.data.add\_(-group['lr'], d\_p)

return loss

step()函数的作用是执行一次优化步骤，通过梯度下降法来更新参数的值。因为梯度下降是基于梯度的，所以在执行optimizer.step()函数前应先执行loss.backward()函数来计算梯度。

注意：optimizer只负责通过梯度下降进行优化，而不负责产生梯度，梯度是tensor.backward()方法产生的。

**Shuffle( )**

import random

random.shuffle (lst )

lst可以时一个列表。Shuffle()需要导入random模块

torch.randperm(n)：将0~n-1（包括0和n-1）随机打乱后获得的数字序列，函数名是random permutation缩写

【sample】

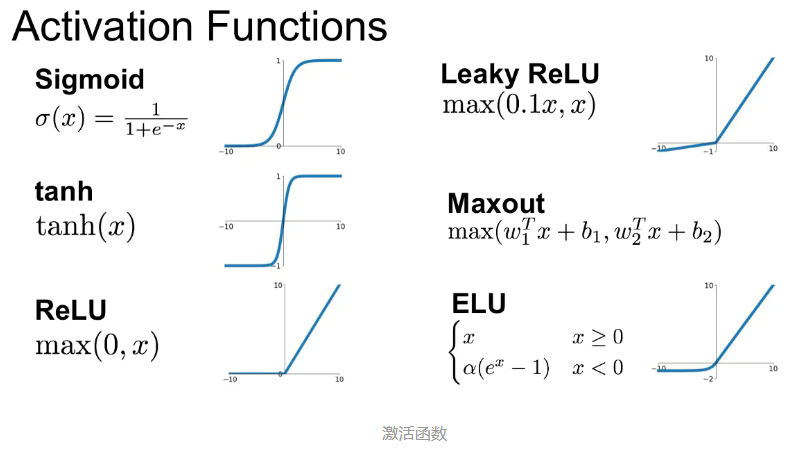
torch.randperm(10)

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

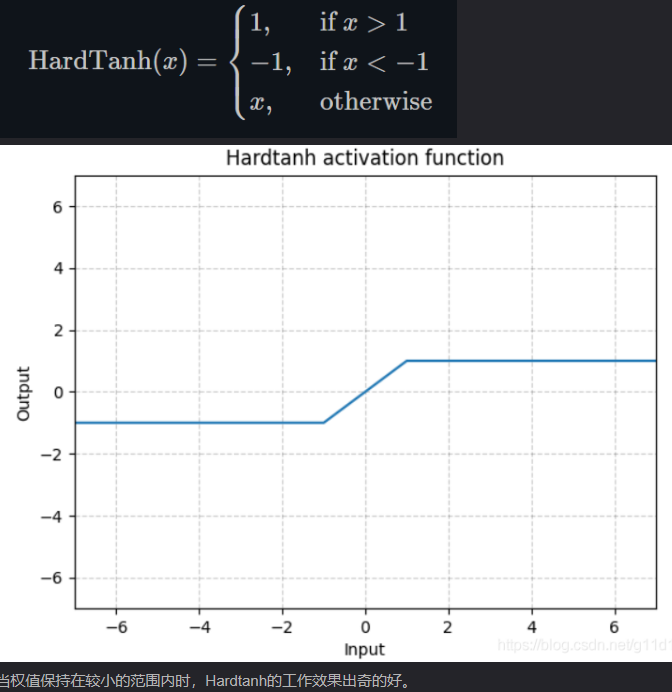
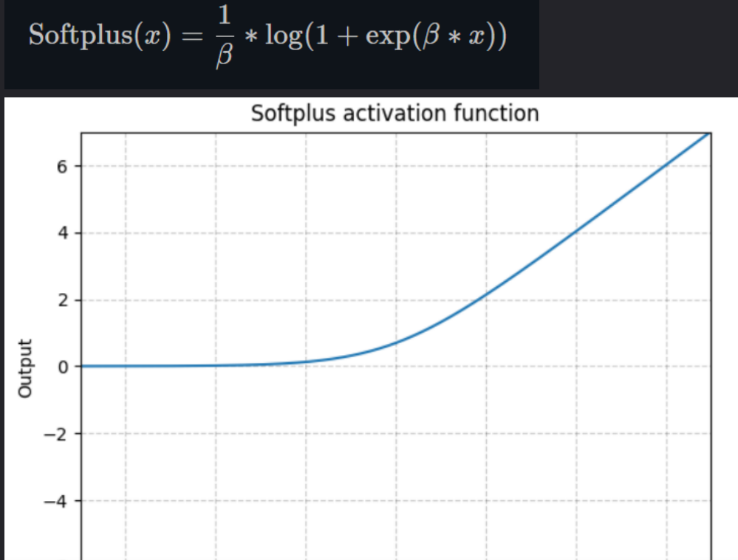
**训练集train，验证集validation，测试集test**

测试集只使用一次，即在训练完成后评价最终的模型时使用。不参与学习参数、不参与超参数选择。仅仅用于模型的评价。

**交叉验证**：将dataset分为train和test，再将train作交叉验证，分为train和validation



LeakyReLU`函数将标准`ReLU`修改为在负输入时具有小的正斜率（该斜率通常为0.01，但为清楚起见，此处显示的斜率为0.1）而不是严格为零



激活函数：

· 非线性。在没有激活函数的情况下重复应用

会产生多项式。非线性的激活函数允许整个网络能近似更复杂的函数。

·可微。激活函数是可微的这样就可以计算穿过它们的梯度。不可微的离散点是无伤大雅的，例如Hardtanh和ReLU。

·具有至少一个敏感范围，其输入的轻微变化会导致输出中相应的变化

·具有至少一个不敏感/饱和范围，其中输入的变化导致输出的变化很小甚至没有变化

·当输入变为负无穷大时接近/达到下限

·当输入变为正无穷大时接近/达到上限

PyTorch的**nn.Linear（）**是用于设置网络中的全连接层的，需要注意在二维图像处理的任务中，全连接层的输入与输出一般都设置为二维张量，形状通常为[batch\_size, size]，不同于卷积层要求输入输出是四维张量。

in\_features指的是输入的二维张量的大小，即输入的[batch\_size, size]中的size。

  out\_features指的是输出的二维张量的大小，即输出的二维张量的形状为[batch\_size，output\_size]，当然，它也代表了该全连接层的神经元个数。

  从输入输出的张量的shape角度来理解，相当于一个输入为[batch\_size, in\_features]的张量变换成了[batch\_size, out\_features]的输出张量。

nn.Linear()的初始化：

in\_feature： int型, 在forward中**输入Tensor最后一维的通道数**

out\_feature： int型, 在forward中**输出Tensor最后一维的通道数**

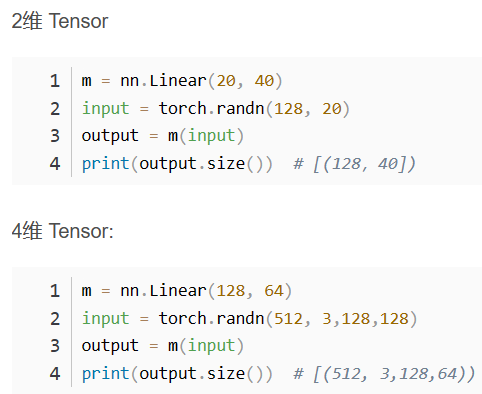
bias: bool型， Linear线性变换中是否添加bias偏置

nn.Linear()的执行：（即执行forward函数）

out=nn.Linear(input)

input: 表示输入的Tensor，可以有多个维度

output: 表示输入的Tensor，可以有多个维度



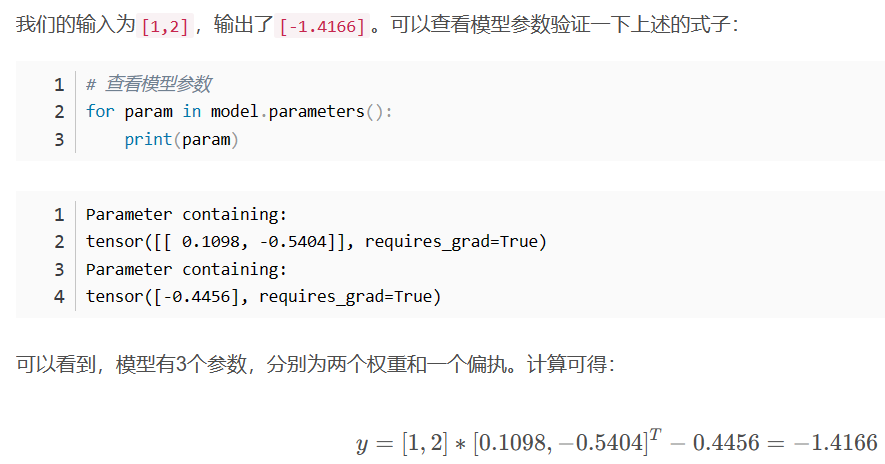
torch.nn.Linear(in\_features, # 输入的神经元个数

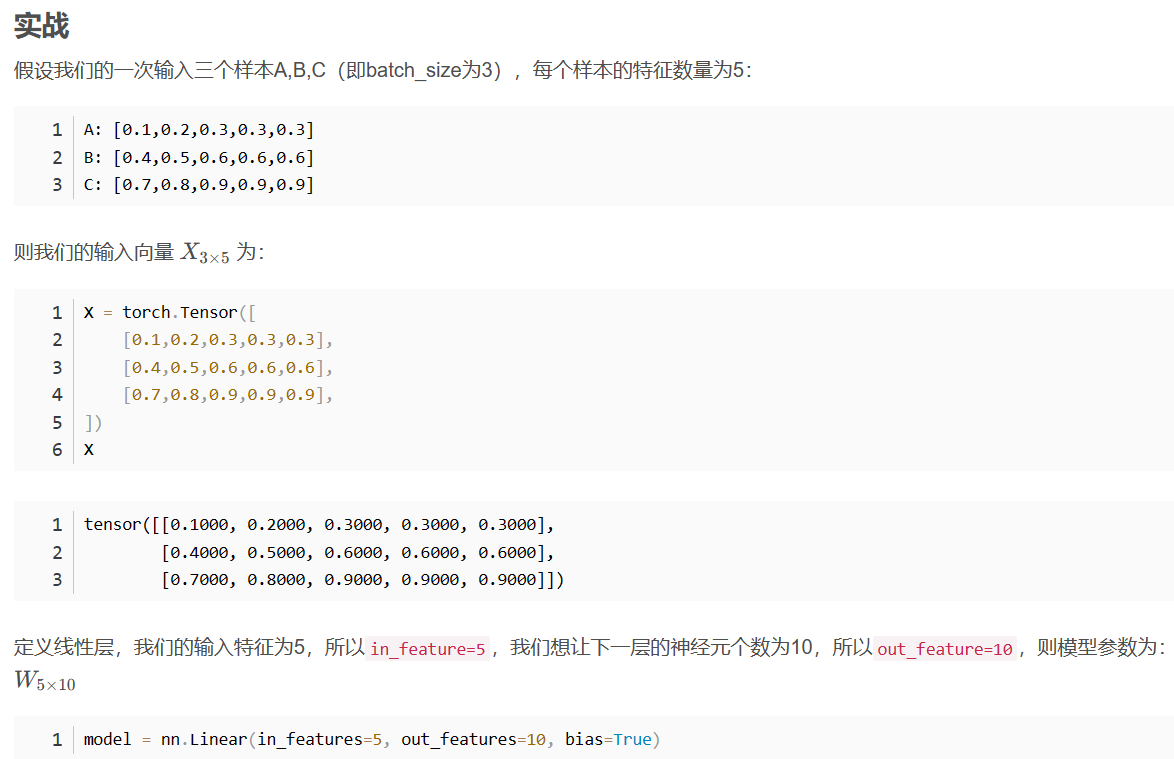
out\_features, # 输出神经元个数

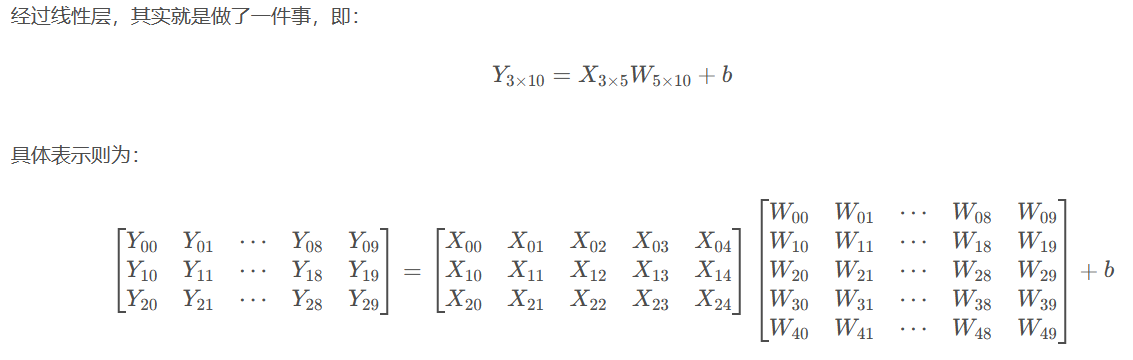
bias=True # 是否包含偏置

)









-> python函数定义的函数名后面，为函数添加元数据,描述函数的返回类型，从而方便开发人员使用

**内置变量**：用双下划线开头且结尾的变量





\*args：传递一个可变参数列表给函数实参



\*\*kwargs：将一个可变的关键字参数的字典传给函数实参，同样参数列表长度可以为0或为其他值。



**register\_forward\_hook**

**手动在forward之前注册hook，hook在forward执行以后被自动执行**

（1）hook

由于pytorch会自动舍弃图计算的中间结果，所以想要获取这些数值就需要使用hook。

每次前向传播执行结束后会执行钩子函数（hook）。前向传播的钩子函数具有如下形式：hook(module, input, output) -> None，而反向传播则具有如下形式：hook(module, grad\_input, grad\_output) -> Tensor or None。

Pytorch中包含forward和backward两个钩子注册函数，用于获取其中输入和输出——不改变网络的定义代码，也不需要在forwaed函数中return某个感兴趣层的输出

register\_forward\_hook()函数必须在forward（）函数调用之前被使用

**torch.nn.Sequential**

作为一个容器，实现简单的顺序连接模型





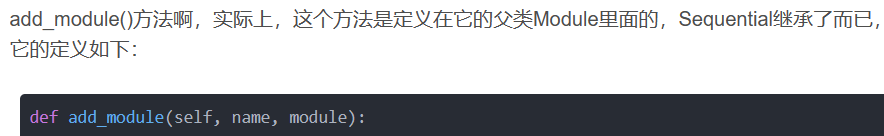
注意：从上面的结果中可以看出，这个时候每一个层都有了自己的名称，但是此时需要注意，我并不能够通过名称直接获取层，依然只能通过索引index，即

model[2] 是正确的

model["conv2"] 是错误的

这其实是由它的定义实现的，看上面的Sequenrial定义可知，只支持index访问。





**super().\_\_init\_\_()**

super()用来调用父类(基类)的方法，\_\_init\_\_()是类的构造方法，

super().\_\_init\_\_() 就是调用父类的init方法，同样可以使用super()去调用父类的其他方法。

