

tensor 张量的创建与基本操作

元组和列表

定义一个普通**列表**，用中括号表示：

```
listvar = [111,3,13,True,3+4j,"abc"]print (listvar,type(listvar))
```

列表的特点：**可获取，可修改，有序**。

定义一个普通**元组**：

```
tuplevar = (False,3+4j, "aaa",456)
```

元组特点：**可获取、不可修改、有序**

综上所述，列表和元组存储的数据类型都是多种多样的，最大的区别就是列表创建完成之后可以修改，而元组一旦创建之后不可修改（但是两个元组之间可以连接）。

后面我们将看到张量的很多操作，其参数是以元组或者列表传递的。

张量 Tensor 的定义

张量在形式上就是多维数组，例如标量就是 0 维张量，向量就是一维张量，矩阵就是二维张量，而三维张量就可以想象 RGB 图片，每个 channel 是一个二维的矩阵，共有三个 channel，还可以考虑更多。

在进行张量的各种操作时，需要牢牢掌握张量的两个特性，

维度 dimension：可以理解为一个坐标轴，从 0 开始计算。张量维度更加抽象：可以表示非空间概念（如批次大小、特征通道、时间步长等），在代码中，每一个中括号代表一个维度，深度学习中一般会有 3-4 个维度（batch_size,channel,width,height）

形状 shape：对应维度的数值大小。

在张量里面我们描述某个维度的大小用形状，但是当我们特定描述张量里某个向量的时候，又会说向量的维度大小，要注意区分。

举个例子来说，张量 a 的形状为（1,2,3），则该张量有 4 维，第 0,1,2,3 维的形状大小分别为 1,2,3,4.特别注意这两个特性，因为张量的所有操作都是针对这两个特性的。

下面重点探讨一下张量的维度和形状。

对于一维张量（从 0 开始计数，即第 0 维），可以理解为**向量**，对应形状为（s0），而当期形状长度为 1 的时候，可以理解为**标量（也可以认为是 0 维张量）**；在 nlp 中对应一个单词的词向量 d_model；

对于二维张量，可以理解为**二维矩阵**，对应形状为（s0,s1），在 nlp 中对应一个序列的词向量，[n_seq,d_model]。在图像处理中对应[width,height]

这里出现一个容易混淆的点，形状（3，）（**元组（Tuple）只有 1 个元素时，必须在元素后加逗号**，否则 Python 会将其识别为“元素本身”，而非元组类型。）和形状（1,3）分别对应的是一维张量和二维张量，其中二维张量的第 0 维形状长度为 1。

两者在内存中的存储完全一致，都是连续存储的 3 个元素

（3，）表示**同质元素的集合**：如 3 个像素值、3 个温度读数

（3,1）表示**结构化数据**：3 个**独立实体**（行）

每个实体有 **1 个特征**（列）

对于三维张量，可以理解为二维张量并排成立体，对应形状为 (s_0, s_1, s_2) ，在 `nlp` 中对应 `[batch_size, n_seq, d_model]`，在图像处理中对应 `[channel, width, height]`，

对于四维张量，可以理解为一组三维张量立体，在 `nlp` 中可以对应为 `[batch_size, n_seq, n_head, d_model]`

更高维的也可以认为是低维度的某种组织结构，事实上，三维及以上均可以认为是二维矩阵以某种嵌套结构组织得到的。以形状为 $(2, 3, 2, 4)$ 的四维张量为例，其表示该张量由 2 个大小为 $(3, 2, 4)$ 的子张量组成，每个子张量又有 3 个大小为 $(2, 4)$ 的二维矩阵。

让我们通过几个具体例子来理解这个概念：

示例 1：将 3D 张量理解为 2D 张量的堆叠

假设我们有一个形状为 $[2, 3, 4]$ 的 3D 张量（例如：2 张图像，每张 3×4 像素）：

创建一个 3D 张量

```
tensor_3d = torch.tensor([
    # 第一个 2D 张量（矩阵）
    [
        [1, 2, 3, 4],
        [5, 6, 7, 8],
        [9, 10, 11, 12]
    ],
    # 第二个 2D 张量（矩阵）
    [
        [13, 14, 15, 16],
        [17, 18, 19, 20],
        [21, 22, 23, 24]
    ]
])
```

```
print("3D 张量形状:", tensor_3d.shape) # torch.Size([2, 3, 4])
```

这个 3D 张量可以理解为：2 个 3×4 的矩阵沿着一个新的（批次）维度堆叠而成。

示例 2：将 4D 张量理解为 2D 张量的嵌套堆叠

在计算机视觉中，一个典型的 4D 张量形状是 `[batch_size, channels, height, width]`：

假设一个 4D 张量：2 张 RGB 图像，每张 3×4 像素

```
tensor_4d = torch.randn(2, 3, 3, 4)
print("4D 张量形状:", tensor_4d.shape) # torch.Size([2, 3, 3, 4])
```

这个 4D 张量可以理解为：

最外层：2 个样本（批次维度）

第二层：每个样本有 3 个通道（R、G、B）

最内层：每个通道是一个 3×4 的矩阵

换句话说，这是 $2 \times 3 = 6$ 个 3×4 矩阵，以特定的层次结构组织起来。

内存中的实际存储方式

理解这一点至关重要：无论张量有多少维，它在内存中总是以连续的一维数组形式存储。

多维张量是通过"步幅"（strides）和"形状"（shape）信息来解释这一维数组的：

查看张量的内存布局信息

```
tensor = torch.randn(2, 3, 4)
```

```
print("形状:", tensor.shape) # 如何解释数据
```

```
print("步幅:", tensor.stride()) # 如何访问数据
```

输出可能类似于：

形状: torch.Size([2, 3, 4])

步幅: (12, 4, 1)

这意味着：

沿维度 0 移动 1 步，需要在内存中移动 12 个元素

沿维度 1 移动 1 步，需要移动 4 个元素

沿维度 2 移动 1 步，需要移动 1 个元素

13			
1			
5			

1				5								13
---	--	--	--	---	--	--	--	--	--	--	--	----

在调用张量 api 操作时往往有两种形式，一是 `t.api_name(arg)`，二是 `api_name(t,arg)`

在代码中创建张量 Tensor 数据类型时，除了封装张量本身的数据 data 外，还会附加张量的一些性质和操作，例如数据的梯度（grad），创建 tensor 的函数（grad_fun，是求导的关键），是否为叶子节点（is_leaf），是否需要梯度（require_grad）。这部分主要涉及到梯度和损失函数，后面再用专题介绍。

张量 tensor 创建

首先来谈一下张量的创建，创建的方法有 2 大类，一是由现有的数据直接转化而来，称为转化法；二是利用各种 api 函数传入想要创建张量的维度和形状，称为 api 法。

转化法 torch.tensor()和 torch.from_numpy

转化法是从现有的数据转化而来，现有的数据从哪里来呢？主要有 2 种：

(1) `tensor()` 括号里的数据可以是列表 `list` (以 “`[]`” 表示), 即需要什么手动写什么, 很少用;

(2) 也可以是 `numpy`, 即先用 `numpy` 创建一个 `numpy` 数组, 然后直接导入 (如下);

```
torch.tensor([1])

arr = np.array([[1, 2, 3], [4, 5, 6]])
t = torch.tensor(arr, device='cuda')
```

值得注意的点:

(1) 注意数据类型, 有时候需要在数字后面加 “.” 表示 `float`, 因为求导时候需要 `float` 类型;

(2) 可以添加 `device='cuda'` 获得加速。

上述从 `numpy` 导入的数据是不共享内存的, 还有一种导入方法: `torch.from_numpy` 创建的 `tensor` 和原来的 `numpy` **共享内存**, 也即是说修改 `tensor` 就会修改原来的 `numpy`。如下

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
t = torch.from_numpy(arr)
# arr[0, 0] = 0
t[0, 0] = -1
```

Api 法

特殊数字 `torch.zeros()/ones()/eye()/full()`

此类方法的共性就是利用给定的 `api` 函数, 如 `zeros`, `ones`, `eyes`, `full`, 然后在参数里指定想要创建的张量的形状 (用元组表示), 即可完成创建。

(1) `torch.zeros/ones (shape)` 创建全 0/1 张量: `tensor3 = torch.zeros/ones(2, 3)`

(2) `torch.full (shape, value)` 创建值全相同的张量: `t = torch.full((3, 3), 2)`

(3) `torch.eye(shape)` 创建单位对角矩阵

(4) `torch.zeros_like(tensor)`, `torch.ones_like(tensor)`, `torch.full_like(tensor,value)` 创建和参数张量 (本质用真实的张量形状代替指定的形状) 一致的全 0/1 张量。

等差均分 `torch.arange&linespace`

此类方法创建的是一维张量,

(1) `torch.arange (start, end,step)` 创建等差数列张量, `step` 为等差值, 默认为 1, 取值时 `[start,end)` 左闭右开, 形状大小 $= (\text{end}-\text{start})/\text{step}$ 。

`t = torch.arange(2, 10, 2)`

(2) `torch.linspace (start, end, n)`, 代表创建的张量在 `[start,end]` 中数值均分 `n` 等份, 这时会出现小数。左右均闭。 `t = torch.linspace(2, 10, 6)`

(3) `torch.logspace()`, 等 `log` 创建, 在对数尺度上从 `start` 到 `end` 均匀取 `steps` 个

点，再通过底数 base 转换为线性尺度的数值。tensor11=torch.logspace(0,2,5) # [1.0000, 3.1623, 10.0000, 31.6228, 100.0000]

概率法

此类方法的本质是依据想要使用的概率分布来创建符合要求的张量，关键还是确定张量的形状。

以正态分布为例：首先参数要有概率分布的超参数，比如正态分布标准值和方差，然后根据超参数的类型（可以是标量和张量（float 类型））来决定是否需要额外指定张量的形状。

以 mean, std 的组合为例，共有四种模式（2*2）。

mean1=torch.arange(1,5) mean2=2.0; std1=torch.ones(3,1) std2=0.3

模式 1: mean 为张量，std 为标量——输出形状以 mean 的形状为准：
tensor1=torch.normal(mean1,std2)

模式 2: mean 为标量，std 为张量——输出形状以 std 的形状为准：
tensor2=torch.normal(mean2,std2)

模式 3: mean 和 std 均为标量——必须指定输出张量的形状（size 参数）
tensor3=torch.normal(mean2,std2,size=(2,3))

模式 4: mean 和 std 均为张量，形状需要可广播，否则会引发运行时错误

tensor_normal3=torch.normal(mean1,std1)

在实际应用中，最常用的是模式 3（两个标量参数+指定形状）和模式 1（一个张量均值+标量标准差）。

对于 mean 和 std 均为标量，多维张量的每个元素都是独立的正态分布样本，整体均值会接近设定的均值，整体标准差接近设定的标准差；维度不影响分布规律，仅决定张量的形状（如 (2,3) 表示 2 行 3 列的样本集合）。

对于 mean 或 std 为张量，创建张量逐元素对应：目标张量中位置 (i,j) 的元素，服从以 mean[i,j] 为均值、std[i,j] 为标准差的正态分布。

- (1) torch.randn(shape),torch.randn_like(tensor)创建标准正态分布张量；
- (2) torch.rand(shape),torch.rand_like(tensor)创建[0,1]均匀分布。
- (3) torch.randint(low,high),torch.randint_like(tensor,low,high)创建[low,high)均匀分布。
- (4) torch.randperm(n)，创建从 0 到 n-1 的随机排列张量
- (5) torch.bernoulli(input)，创建以 input 为概率值的伯努利分布张量。

张量 tensor 读取

切片 tensor[start:step:end,start::step,:]

张量的读取与访问与列表一致，只是多了维度，其本质是根据数据在各个维度形状位置来获取，访问单个元素时，直接写出其每个维度上的形状上的位置（二维举例来说就是行和列值），切片操作就是以列表的形式用 start:end 来指代每个维度获取多少数据，这里也可以设置步长，方法是 tensor[start:step:end]，如果是到张量维度末尾结束，还可以简化为 tensor[start::step]。

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

```
# 访问单个元素 print("单个元素:", tensor[0, 1])
```

```
# 切片操作 print("切片结果:", tensor[0:2, 1:3])
```

张量索引 `index_select&masked_select`

```
torch.index_select(input,dim,select)
```

```
t = torch.randint(0, 9, size=(3, 3))
idx = torch.tensor([0, 2], dtype=torch.long) # float
t_select = torch.index_select(t, dim=0, index=idx)
```

(1) 在 `dim` 维度上按照 `index=idx` 索引数值。其中 `idx` 是张量（相当于将切片中规律的取值变为**手动指定特定维度上的形状值**）。如上，就是取张量 `t` 在维度 0 上的第 0 个和第 2 个张量。以二维为例就是取第 0 行和第 2 行。

```
torch.masked_select(input,mask)
```

```
t = torch.randint(0, 9, size=(3, 3))
mask = t.le(5) # ge is mean greater than or equal/ gt: greater than le lt
t_select = torch.masked_select(t, mask)
```

(1) `mask` 是和 `input` 同大小的布尔类型张量，寻找张量 `t` 中和 `mask` 张量为 `TRUE` 对应位置的数据并返回**一维张量**。

张量 `tensor` 形状 `shape` 变化

张量切分 `chunk&split`

```
torch.chunk(input,chunk,dim)
```

```
a = torch.ones((2, 7))
```

```
chunk_tensors = torch.chunk(a, chunks=3, dim=1)
```

在维度 `dim` 上进行 `chunk` 均分，如果不能整除，最后一份为余数。返回的是切分后的张量组成的元组

```
torch.split(input,int/list,dim)
```

```
t = torch.ones((2, 5))
```

```
split_tensors = torch.split(t, 2, dim=1)
```

```
# split_tensors = torch.split(t, [2, 1, 2], dim=1)
```

(1) 为 `int` 时，和 `chunk` 功能类似；

(2) 为 `list` 时，可以按照设定值切分，但总和要与输入张量对应维度上的形状大小一致。

张量 `tensor` 维度和形状变化

张量拼接 `cat&stack`

```
torch.cat(tensors,dim)
```

```
t = torch.ones((2, 3))
```



```
t_0 = torch.cat([t, t], dim=0)
t_1 = torch.cat([t, t, t], dim=1)
```

(1) 是将两个张量在原来的维度上进行拼接，这就要求两个张量其它维度的形状完全一致。

torch.stack(tensor,dim)

```
t = torch.ones((2, 3))
t_stack = torch.stack([t, t, t], dim=0)
```

(1) 是在新创建的维度（维度由参数指定）上进行拼接，如果指定的维度小于现存的维度，比如上面维度为（0 和 1，指定在维度 0 上创建，小于现存维度）则创建该维度后，后面的递推。比如 t 现在维度是 2*3，拼接后，则是 3*2*3（即新创建的维度 0 形状为 3），其中后两维的 2*3 是原来的 t。两个张量完全相同的形状。

torch.cat 像是把两张纸并排贴在一起（面积变大，厚度不变）

torch.stack 像是把两张纸叠在一起（增加了厚度维度）

选择使用哪个函数取决于你是否需要在拼接时增加新的维度。如果只是想将多个同形状张量合并成一个更大的张量，用 torch.cat；如果想保留各个张量的独立性并增加一个新的维度来组织它们，用 torch.stack。

张量 reshape&view

在 PyTorch 中，reshape 和 view 方法都用于改变张量（Tensor）的形状，二者变换前后张量的形状乘积需相等。但它们在功能和使用场景上存在一些差异。下面将详细对比这两个方法，并给出实例和输出结果。

相同点

- **功能用途：**reshape 和 view 方法的主要目的都是改变张量的形状，且新形状的元素总数必须与原张量的元素总数相同。
- **返回视图：**通常情况下，它们返回的都是原张量的视图（view），而不是副本，这意味着对返回的张量进行修改可能会影响原张量，反之亦然。

不同点

- **灵活性：**reshape 方法更加灵活，它可以处理内存布局不连续的张量，在需要时会自动复制以得到一个新的连续张量。而 view 方法只能用于连续的张量，如果张量不连续，调用 view 会报错。
- **适用场景：**如果不确定张量是否连续，或者张量可能不连续，建议使用 reshape；如果能确保张量是连续的，使用 view 可能会更高效，因为它不会进行额外的复制操作。（如何判断张量是否连续？）

torch.reshape(input,shape)

```
t = torch.randperm(8)
t_reshape = torch.reshape(t, (-1, 2, 2)) # -1
t[0] = 1024
```

也可以先确定部分维度的形状大小，剩下的一个维度用-1 表示，此时该维度的形状即是原形状乘积/其它所有维度的形状乘积。

张量维度交换 transpose

`transpose` 方法用于交换张量的两个指定维度，下面详细介绍其变化原理并给出示例。

假设原始三维张量的形状为 (2, 3, 4)，表示有 2 个大小为 (3, 4) 的二维矩阵。`transpose(0, 1)` 会将第 0 维和第 1 维交换，交换后张量的形状变为 (3, 2, 4)，即现在有 3 个大小为 (2, 4) 的二维矩阵。

输出分析

- **形状变化**：每次交换不同的维度，张量的形状会相应改变，改变规则是所交换的两个维度的大小互换，其他维度大小保持不变。
- **元素排列变化**：元素会根据新的维度顺序重新排列，例如在交换第 1 维和第 2 维时，原本在第 1 维的元素会移动到第 2 维的对应位置，反之亦然。

`torch.transpose(input, dim1, dim2)`

```
# torch.transpose
t = torch.rand((2, 3, 4))
t_transpose = torch.transpose(t, dim0=1, dim1=2) # c*h*w  h*w*c
```

(1) 维度变换之后，数据是如何变化的？

(2) `torch.t()` 二维张量（矩阵）转置

至于维度的变换，从数学上看，可以认为是对应维度的形状长度的调换，从物理意义上看，可以理解为数据结构的重新组织，简单的二维张量的调换，可以理解为长和宽的对换；而对于 transformer 中的多头注意力的张量的维度变换，如从 `[batch_size, n_seq, n_head, d_model]` 变换为 `[batch_size, n_head, n_seq, d_model]`，第 0 维是批次大小一直不变，第 1 维原来是序列长度，后面是头数和每头的词向量长度，代表一个单词的不同空间的特征表示，变换为第 1 维是头数，后面是序列长度和每头的词向量长度，代表一个头空间的不同单词（一个序列）的特征表示，这样就可以挖掘不同头空间的特征表示了。

原始维度：每个研究团队(批次)有多个专家(头)，每个专家处理问题的不同方面(序列位置)

交换后维度：按照专家领域组织，每个专家查看所有团队的问题的特定方面

上面是物理意义上的解释，那么数据具体是如何变化的呢？下面结合实例从数学索引上（坐标）解释说明：

假设我们有多头注意力的张量 `[batch=2, seq=3, heads=2, features=2]`

```
multi_head_tensor = torch.tensor([
    # 批次 0
    [
        # 序列中的单词 0
        [[1, 2], [3, 4]], # 每个序列中的单词有 2 个头空间，每个头空间的单词特征长度为 2
    ]
])
```



```

    # 序列中的单词 1
    [[5, 6], [7, 8], ],
    # 序列中的单词 2
    [[9, 10], [11, 12]]
],
# 批次 1
[
    # 序列中的单词 0
    [[13, 14], [15, 16]],
    # 序列中的单词 1
    [[17, 18], [19, 20]],
    # 序列中的单词 2
    [[21, 22], [23, 24]]
]
])

# 交换序列长度（单词数）和头维度 [batch, seq, heads, features] -> [ batch,heads,
seq, features]
transposed_multi_head = multi_head_tensor.transpose(1, 2)
# 输出为:
tensor([ # 批次 0
    [
        # 头 0 空间
        [[1, 2], [5, 6], [9, 10]],# 对于同一个头的空间，每个序列的三个单词的特征
        长度为 2
        # 头 1 空间
        [[3, 4], [7, 8], [11, 12]]
    ],

    # 批次 1
    [[[13, 14], [17, 18], [21, 22]],

    [[15, 16], [19, 20], [23, 24]]]
])

```

然后对比一下维度 1 和 2 变换前后的数据，可以发现交换两个维度会交换这两个维度的索引顺序，对于每个元素，其在这两个维度上的坐标会互换。，具体到上面的例子，对应第一个数字 1，原来索引是[0,0,0,0]，变换后不变，这是因为它在维度 1 和 2 上的坐标相同；对于数字 5，原来的坐标为[0,1,0,0]变换后的坐标为[0,0,1,0]。

张量维度压缩扩充 squeeze&unsqueeze

torch.squeeze(input,dim)

(1) `squeeze` 方法用于移除张量中维度大小为 1 的维度。如果指定了维度参数，则只移除该指定维度上大小为 1 的维度，若指定维度不为 1，则不会压缩；若不指定维度参数，则默认移除所有大小为 1 的维度。

```
t = torch.rand((1, 2, 3, 1))
t_sq = torch.squeeze(t)#形状变为 (2,3)
t_0 = torch.squeeze(t, dim=0)#形状变为 (2,3,1)
t_1 = torch.squeeze(t, dim=1)#形状仍为 (1,2,3,1)

torch.unsqueeze(),
```

`unsqueeze` 方法在指定位置插入一个维度大小为 1 的新维度。在指定维度上设置为 1，其它维度形状不变。`torch.unsqueeze(input,dim)`。`dim` 为 0 意味着在维度 0 之前插入，其它维度后移。

```
t = torch.randn(2, 3, 4) # 原始形状: (2, 3, 4)
print(t.unsqueeze(0).shape) # torch.Size([1, 2, 3,
4])print(t.unsqueeze(1).shape) # torch.Size([2, 1, 3,
4])print(t.unsqueeze(2).shape) # torch.Size([2, 3, 1,
4])print(t.unsqueeze(3).shape) # torch.Size([2, 3, 4,
1])print(t.unsqueeze(-1).shape) # torch.Size([2, 3, 4, 1]) (等价于
dim=3) print(t.unsqueeze(-4).shape) # torch.Size([1, 2, 3, 4]) (等
价于 dim=0)
```

(1) 不会改变张量中元素的总数，但可以改变张量的形状，使用 `unsqueeze` 来添加缺失的维度，使其能够进行某些操作（例如广播、卷积层要求输入数据具有通道维度等）。

(2) 维度的索引从 0 开始，并且可以是负数，负数表示从后往前数（-1 表示最后一个维度）。

举个简单的例子，假设我们有一个一维张量（向量，`torch.Size([3])`），我们想要将其变成一个行向量（二维张量，形状为（1,3）），可以在维度 0 上添加新维度；同样，如果我们想将其变成一个列向量（二维张量，形状为（3,1）），可以在维度 1 上添加新维度（或者用 `dim=-1` 在最后一个维度后添加）。

(3) 那么可以在任意维度添加吗？需要在指定的维度索引在有效范围内。有效索引范围是`[-input.dim()-1, input.dim())`。即对于 `n` 维张量，有效索引范围是`[-n-1, n]`，但通常我们使用`[0, n]`或负数索引`[-n-1, -1]`

对于一个三维张量（形状为（a, b, c）），索引范围是`[0,3]`或`[-4,3]`：当 `dim=0/-4` 时，新形状为（1, a, b, c）；当 `dim=1/-3` 时，新形状为（a, 1, b, c），当 `dim=2/-2`：新形状为[a, b, 1, c]；当 `dim=3/-1` 时，在原始三维中，`dim=3` 相当于在最后一个维度之后添加。

(4) 一个常见的操作是使用 `None` 在索引时增加维度。例如：`y = x[None, :]` # 相当于在 `dim=0` 上添加，形状为(1,3)；`z = x[:, None]` # 相当于在 `dim=1` 上添加，形状为(3,1)。

(5) 物理存储不变：两种扩展都不改变数据在内存中的顺序，只是改变了维度解释方式。

广播机制

广播（Broadcasting）是一种强大的机制，它允许在**不同形状**的张量之间进行**算术运算**。可以让你在不进行显式的循环或数据复制的情况下，对不同形状的张量进行操作，从而提高代码的效率和简洁性。

1. **维度对齐**：从张量的**最后一个维度**开始比较，如果两个张量的维度大小相同，或者其中一个张量的维度大小为 1，则认为这两个维度是兼容的。
2. **维度扩展**：如果一个张量的某个维度大小为 1，而另一个张量在该维度上有更大的大小，则将大小为 1 的维度扩展为与另一个张量相同的大小。
3. **形状匹配**：如果两个张量在所有维度上都兼容，则可以进行广播操作。意味着从最后一个维度需要一直比较到第一个维度，即两个张量的所有对应维度都必须相等，或者其中一个维度为 1。

张量 arr1 的形状为 **(2, 3)**，张量 arr2 的形状为 **(1, 2)**。从最后一个维度开始比较，**3 和 2 不相等，且大小都不为 1**，因此这两个张量不兼容，无法进行广播操作。

张量 tensor 的数学运算

基本算术运算

基本算数运算包括**加法、减法、乘法（*）、除法（/）**等，这些运算是**逐元素的操作**，可以是**完全相同形状**的张量，也可以是满足**广播规则**情况的不同形状的张量。

```
import torch
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])
# 加法
add_result = a + b
print("加法结果:", add_result)
# 减法
sub_result = a - b
print("减法结果:", sub_result)
# 乘法
mul_result = a * b
print("乘法结果:", mul_result)
# 除法
div_result = a / b
print("除法结果:", div_result)
```

tensor 幂运算**

使用 `**` 运算符进行逐元素的幂运算。

```
import torch
# 创建一个示例张量
tensor = torch.tensor([1, 2, 3])
# 幂运算
power_result = tensor ** 2
print("幂运算结果:", power_result)
```

三角函数运算

PyTorch 提供了如 `torch.sin`、`torch.cos`、`torch.tan` 等三角函数运算。

```
import torch
# 创建一个示例张量
angle_tensor = torch.tensor([0, torch.pi / 2, torch.pi])
# 正弦函数运算
sin_result = torch.sin(angle_tensor)
print("正弦函数运算结果:", sin_result)
# 余弦函数运算
cos_result = torch.cos(angle_tensor)
print("余弦函数运算结果:", cos_result)
```

求和、均值、最大值、最小值运算

可以使用 `torch.sum`、`torch.mean`、`torch.max`、`torch.min` 等函数对张量进行相应的统计运算。

还可以指定维度，`torch.sum(input, dim, keepdim=False)` 函数用于对输入张量 `input` 沿着指定的维度 `dim` 进行求和。`keepdim` 参数用于控制输出张量是否保持和输入张量相同的维度数。注意虽然维度保持了，但是保持方法是原本消失的维度的形状变为 1。

`torch.max(input, dim, keepdim=False)` 函数用于对输入张量 `input` 沿着指定的维度 `dim` 找出最大值，并返回最大值和对应的索引。

```
import torch
# 创建一个示例张量
stat_tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])
# 求和
# 按行求和（指定维度 0）
sum_row = torch.sum(tensor, dim=0)
print("按行求和结果:", sum_row)
# 按列求和（指定维度 1）
```

```

sum_col = torch.sum(tensor, dim=1)
print("按列求和结果:", sum_col)
# 按行求和并保持维度
sum_row_keepdim = torch.sum(tensor, dim=0, keepdim=True)
print("按行求和并保持维度结果:", sum_row_keepdim)
# 均值
mean_result = torch.mean(stat_tensor.float())
# 需要转换为浮点型以计算均值
print("均值结果:", mean_result)
# 最大值
# 按行找最大值（指定维度 0）
max_row, max_row_indices = torch.max(tensor, dim=0)
print("按行找最大值结果:", max_row)
print("按行找最大值对应的索引:", max_row_indices)
# 按列找最大值（指定维度 1）
max_col, max_col_indices = torch.max(tensor, dim=1)
print("按列找最大值结果:", max_col)
print("按列找最大值对应的索引:", max_col_indices)
# 按行找最大值并保持维度
max_row_keepdim, _ = torch.max(tensor, dim=0, keepdim=True)
print("按行找最大值并保持维度结果:", max_row_keepdim)
# 最小值
min_result = torch.min(stat_tensor)
print("最小值结果:", min_result)

```

矩阵乘法 `torch.matmul (t1,t2)` 或 `@`

在 PyTorch 中，`torch.matmul` 是处理**矩阵乘法**的核心函数，它对二维及以上的多维张量有特殊的处理逻辑，尤其适合批量矩阵乘法（batch matrix multiplication）。其核心规则是：**将前 N-2 个维度视为“批次维度”，仅对最后两个维度执行标准矩阵乘法**，同时支持批次维度的广播（broadcasting）。

1. 忽略前序维度（所有维度除了最后两个），仅对最后两维执行标准矩阵乘法（要求 `a.shape[-1] == b.shape[-2]`）。
2. 前序维度（称为“批次维度”）需要满足**广播条件**（如形状相同，或其中一个为 1），最终结果的前序维度为广播后的形状。
3. 结果的形状为：`[广播后的批次维度] + (a.shape[-2], b.shape[-1])`。

以下是常见的多维张量乘法场景，结合形状变化理解更清晰：

二维张量 × 二维张量（最基础的矩阵乘法）

- 规则：符合标准矩阵乘法， $(m, n) \times (n, p) \rightarrow (m, p)$ 。

```
a = torch.randn(2, 3) # 形状 (2, 3)
b = torch.randn(3, 4) # 形状 (3, 4)
c = torch.matmul(a, b) # 结果形状 (2, 4)
```

2. 一维张量 × 一维张量（点积）

- 规则：视为两个列向量的点积，结果为标量（0 维张量）。

```
a = torch.randn(3) # 形状 (3,)
b = torch.randn(3) # 形状 (3,)
c = torch.matmul(a, b) # 结果形状 标量，等价于 a·b
```

3. 一维张量 × 二维张量

- 规则：一维张量被视为**行向量**（形状扩展为 $(1, n)$ ），与二维矩阵相乘后再挤压掉多余维度。

```
a = torch.randn(3) # 形状 (3,) → 视为 (1, 3)
b = torch.randn(3, 4) # 形状 (3, 4)
c = torch.matmul(a, b) # 结果形状 (4,)，等价于 (1,3)×(3,4)=(1,4) → 挤压为 (4,)
```

4. 二维张量 × 一维张量

- 规则：一维张量被视为**列向量**（形状扩展为 $(n, 1)$ ），相乘后挤压维度。

```
a = torch.randn(2, 3) # 形状 (2, 3)
b = torch.randn(3) # 形状 (3,) → 视为 (3, 1)
c = torch.matmul(a, b) # 结果形状 (2,)，等价于 (2,3)×(3,1)=(2,1) → 挤压为 (2,)
```

5. 高维张量 × 高维张量（批量矩阵乘法）

这是 Transformer 等模型中最常见的场景（如多头注意力中的 Q、K、V 相乘），核心是**前序维度作为批次，最后两维做矩阵乘法**。

示例 1：批次维度完全匹配

```
a = torch.randn(2, 5, 3, 4) # 前序批次维度 (2,5)，最后两维 (3,4)
b = torch.randn(2, 5, 4, 6) # 前序批次维度 (2,5)，最后两维 (4,6)
c = torch.matmul(a, b) # 结果形状 (2,5,3,6) # 逻辑：对每个 (2,5) 批次，执行 (3,4)×(4,6)=(3,6) 的矩阵乘法
```

示例 2：批次维度支持广播（某一维度为 1 时可扩展）

```
a = torch.randn(2, 1, 3, 4) # 批次维度 (2,1)
b = torch.randn(1, 5, 4, 6) # 批次维度 (1,5)
c = torch.matmul(a, b) # 结果形状 (2,5,3,6) # 逻辑：批次维度先广播为 (2,5)，再对每个批次执行 (3,4)×(4,6)=(3,6)
```

示例 3：Transformer 中的多头注意力计算（关键场景）

在多头注意力中，Q、K 的形状为 $(batch_size, num_heads, seq_len, d_k)$ ，K 转置后为 $(batch_size, num_heads, d_k, seq_len)$ ：

```
q = torch.randn(32, 8, 10, 64) # (batch_size=32, num_heads=8, seq_len=10, d_k=64)
```



```
k = torch.randn(32, 8, 10, 64) # 同上
k_t = k.transpose(-2, -1) # 转置后: (32, 8, 64, 10)
attn_scores = torch.matmul(q, k_t) # 结果: (32, 8, 10, 10) # 逻辑: 前序维度 (32,8)
为批次, 最后两维 (10,64)×(64,10)=(10,10)
```

6. 高维张量 × 二维张量

- 规则: 二维张量会被广播到高维张量的批次维度 (相当于二维张量的批次维度都为 1), 再执行批量乘法。

```
a = torch.randn(2, 3, 4) # 批次维度 (2,), 最后两维 (3,4)
```

```
b = torch.randn(4, 5) # 二维矩阵 (4,5)
```

```
c = torch.matmul(a, b) # 结果形状 (2,3,5) # 逻辑: b 广播为 (2,4,5), 再与
a (2,3,4) 执行批量乘法
```

`torch.bmm(input, mat2, out=None)`

`torch.bmm` 是 PyTorch 中的一个函数, 是 `torch.matmul` 的一个特例, 专门用于执行批量矩阵乘法 (Batch Matrix Multiplication)。即输入必须是两个 3D 张量, 第一个维度为批量维度, 后两个维度为矩阵维度。例如, 如果输入是 (b, n, m) 和 (b, m, p) , 则输出为 (b, n, p) 。并且不支持广播, `torch.matmul` 可以处理 (a, b, n, m) 和 (b, m, p) 这样的输入 (广播后相乘), 而 `torch.bmm` 只能处理两个 3D 张量且批量维度必须相等。

在深度学习中, 很多时候需要同时处理多个矩阵乘法, 例如在处理一批数据样本时, 每个样本都需要进行矩阵乘法运算。`torch.bmm` 可以高效地完成这一任务, 它会对批量中的每一对矩阵分别进行矩阵乘法。

参数: `input`: 输入的批量矩阵, 形状为 (b, n, m) , 其中 b 是批量大小, n 是矩阵的行数, m 是矩阵的列数。

- `mat2`: 第二个批量矩阵, 形状为 (b, m, p) , 这里的批量大小 b 必须和 `input` 的批量大小相同, m 要和 `input` 矩阵的列数相同, p 是矩阵的列数。
- `out` (可选): 指定输出的张量, 如果提供了该参数, 结果将存储在这个张量中。

返回值: 返回一个形状为 (b, n, p) 的批量矩阵, 其中 b 是批量大小, n 是第一个输入矩阵的行数, p 是第二个输入矩阵的列数。