

Tensor 元素索引

❗ 参见

阅读这部分内容前，你需要知道如何 [访问 Tensor 中某个元素](#) 以及 [使用切片获取部分元素](#)。

❗ 注解

以下是本小节提到的相关内容速记：

- MegEngine 中切片将返回新的对象（而不是共用同一片内存），切片操作不会降低 Tensor 维度；
- 多维 Tensor 的索引语法形如 `a[i, j]`，也支持切片语法形如 `a[i:j, p:q]`；
- 可以使用省略符 `...` 来自动填充完整切片到剩余维度，比如 `a[i, ...]` 等同于 `a[i, :, :]`。

和 NumPy 索引对比

⚠️ NumPy 用户请注意！

不能将 NumPy 中存在的一些概念和设计直接应用于 MegEngine。

❗ 参见

在 MegEngine 中，想要 [访问 Tensor 中某个元素](#)，可以使用标准的 `x[obj]` 语法。看上去一切都和 NumPy 很相似，后者的官方文档中也对 [ndarray](#) 的各种索引方式都 [进行了解释](#)。但 MegEngine 的 Tensor 实现和 NumPy 还是略有不同，如果不清楚某些细节，可能无法对一些现象做出解释。

索引得到的对象不同

MegEngine

```
>>> x = Tensor([[1., 2.], [3., 4.]])
>>> y = x[0]
>>> y[1] = 6
>>> x
Tensor([[1. 2.]
 [3. 4.]], device=xpux:0)
```

NumPy

```
>>> x = array([[1., 2.], [3., 4.]])
>>> y = x[0]
>>> y[1] = 6
>>> x
array([[1., 6.],
 [3., 4.]])
```

出现这种情况的原因是，在 NumPy 中使用索引时，得到的是原数组的 [视图（View）](#)。改变视图中的元素，原始数组中的元素也会发生变化——这是很多 NumPy 用户初学时容易困扰的地方。而在 MegEngine 中没有视图 `view` 这一属性，通过索引或切片得到的元素或子 Tensor 和原 Tensor 占用的是不同的内存区域。

在其它地方的一些设计，二者还是一致的，接下来我们将进行介绍。

切片索引不会降低维度

MegEngine 和 NumPy 在进行切片时，都不会改变对象 [维度的个数](#)：

```
>>> M = Tensor([[1, 2, 3],
...             [4, 5, 6],
...             [7, 8, 9]])
>>> M[1:2][0:1]
Tensor([[4 5 6]], dtype=int32, device=cpux:0)
>>> M[1:2][0:1].ndim
2
```

整个过程中，切片得到的都是一个 `ndim=2` 的 Tensor。

- 执行 `M[1:2]` 得到的结果是 `[[4, 5, 6]]` 而不是 `[4, 5, 6]`。
- 对 `[[4, 5, 6]]` 进行 `[0:1]` 切片，得到的还是 `[[4, 5, 6]]`。

错误的理解思路可能是这样的：

- 执行 `M[1:2]` 得到的结果是 `[4, 5, 6]`。—— 错！切片不会降维！
- 对 `[4, 5, 6]` 进行 `[0:1]` 切片，得到的是 `4`。—— 降维了，因此也不对！

注解

- 切片的作用是从整体中取出一部分，因此不会产生降低维度的行为。
- 如果你希望切片操作后能去冗余的维度，可以使用 [squeeze](#)。

都可以使用数组索引

实际上除了切片索引，我们还可以使用整数数组进行索引得到特定位置的元素，以一维情况为例：

MegEngine	NumPy
<pre>>>> x = Tensor([1., 2., 3.]) >>> y = x[[0, 2]] >>> y Tensor([1. 3.], device=xpux:0)</pre>	<pre>>>> x = array([1., 2., 3.]) >>> y = x[[0, 2]] >>> y array([1., 3.])</pre>

索引数组的长度对应了被索引的元素的个数，在一些情况下这种机制十分有帮助。

此时 NumPy 将不会生成原始数组的视图，与 MegEngine 的逻辑一致。

警告

注意语法细节，一些用户容易将整数数组索引写成如下形式：

```
>>> x = Tensor([1., 2., 3.])
>>> y = x[0, 1, 2]
IndexError: too many indices for tensor: tensor is 1-dimensional, but 3 were indexed
```

实际上这是对 Tensor 的 n 个维度分别进行索引的语法。引出了下一小节的解释 ——

在多个维度进行索引

以下面这个由矩阵（2 维数组） M 表示的 Tensor 为例：

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad M_{(1,2)} = 6$$

虽然我们可以使用 `M[1][2]` 这样的语法得到 6 这个值，但效率并不高（参考 [访问 Tensor 中某个元素](#) 的解释）。

注解

- Python 的内置序列类型都是一维的，因此只支持单向索引，但对于具备多维属性的 Tensor，可以在多个维度直接进行索引（或者是 [在多个维度进行切片](#)，后面会进行举例），使用 `,` 作为维度之间的分隔，上面的例子则可用 `M[1, 2]` 访问元素，而没有必要使用多个方括号 `M[1][2]`。
- 感兴趣的用户可以了解试着背后的细节：在 Python 中要正确处理这种形式的 `[]` 运算符，对象的特殊方法 `__getitem__` 和 `__setitem__` 需要以元组的形式来接受传入的索引。也即是说如果要得到 `M[i, j]` 的值，Python 实际上会调用 `M.__getitem__((i, j))`。

```
>>> M = Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M[1,2]
Tensor(6, dtype=int32, device=xpux:0)
```

可以理解成，在第 0 轴索引值为 1，第 1 轴索引值为 2 的位置去直接访问元素。

推广到一般情况，在访问 n 维 Tensor（假定为 T ）的特定某个元素时，可以使用如下语法：

$$T_{[i_1, i_2, \dots, i_n]}$$

即我们要提供 i_1, i_2, \dots, i_n 共 n 个索引值，此时不需要层层降维索引，而是直接得到对应元素。

如果提供的索引数组个数不足 n，则需要了解 [多维索引的缺省情况](#)。

在多个维度进行切片

注解

在某个维度上进行索引，除了索引特定元素以外，还可以进行切片操作，来获取特定部分元素。

- 既然我们可以在多个维度进行索引，自然地，我们可以从多个维度进行切片；
- 问题在于，用户容易忽视 [切片索引不会降低维度](#) 这一特点，尤其是和多个 `[]` 使用时。

现在需要从下面这个 2 维 Tensor 中切出蓝色部分的元素：

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$


一些人会写成 `M[1:3][0:2]`，此时将得到非预期结果：

```
>>> M[1:3][0:2]
Tensor([[4 5 6]
       [7 8 9]], dtype=int32, device=xpux:0)
```


这是因为 `[]` 操作是顺序进行解释的，它背后的逻辑顺序是：

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \downarrow_{1:3} = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \downarrow_{0:2} = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
>>> T = M[1:3]
>>> T
Tensor([[4 5 6]
       [7 8 9]], dtype=int32, device=xpux:0)
>>> T[0:2]
Tensor([[4 5 6]
       [7 8 9]], dtype=int32, device=xpux:0)
```

 **警告**

由于切片操作并不会降低维度，所以上面的写法等于每次都在 `axis=0` 进行切片。

 **参见**

如果你不清楚 `axis` 的概念，可以参考 [Tensor 的轴](#)。

正确的做法是像 [在多个维度进行索引](#) 一样，使用 `,` 对维度进行区分：

```
>>> M[1:3, 0:2]
Tensor([[4 5]
       [7 8]], dtype=int32, device=xpux:0)
```

可以理解成在第 0 轴使用 `1:3` 切片，在第 1 轴使用 `0:2` 切片，求它们的交集：

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \downarrow_{1:3} \cap \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \xrightarrow{0:2} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

推广到一般情况，在访问 `n` 维 Tensor（假定为 T ）的特定部分的元素时，要求使用如下语法：


$$T[s_1, s_2, \dots, s_n]$$

即我们要提供 s_1, s_2, \dots, s_n 共 `n` 个切片，每个切片针对特定第维度。

如果提供的切片个数不足 `n`, 则需要了解 [多维索引的缺省情况](#)。

 **注解**

多维切片时，`x[obj]` 内部的 `obj` 由给定的不同维度的切片组成。

 **参见**

- 对于 `ndim` 特别大的 Tensor（假设超过 1000 维），有些时候我们只想对某一个轴进行索引，或进行特定操作，此时我们可以使用 [gather](#) 或 [scatter](#)
- 这两个方法分别对应于 [numpy.take_along_axis](#) 和 [numpy.put_along_axis](#)

多维切片时使用省略符号

在对 Tensor 进行多维切片时，允许对部分不做切片的维度进行省略（Ellipsis）表示。它的正确写法是三个英语句号 `...` 而不是 Unicode 码位 U+2026 表示的半个省略号 `⋯`。Python 解析器会将 `...` 看作是一个符号，就像 `start:end:step` 符号可以表示切片对象一样，省略符号其实是 `Ellipsis` 对象的别名，用于尽可能地在该位置插入尽可能多的完整切片：以将切片语法拓展到所有维度。

举个例子，如果 `T` 是一个 4 维 Tensor, 那么则有：

- `T[i, ...]` 是 `T[i, :, :, :]` 的缩写；
- `T[..., i]` 是 `T[:, :, :, i]` 的缩写；
- `T[i, ..., j]` 是 `T[i, :, :, j]` 的缩写。


多维索引的缺省情况

如果索引一个多维 Tensor 时给定的索引数少于实际的维数 `ndim`, 将得到一个子 Tensor:

```
>>> M[2]
Tensor([7 8 9], dtype=int32, device=xpux:0)
>>> M[2,:]
Tensor([7 8 9], dtype=int32, device=xpux:0)
>>> M[:,2]
Tensor([3 6 9], dtype=int32, device=xpux:0)
```

- 此时其它维度的元素将被完整地保留，等同于使用 `:` 作为缺省维度的默认索引;
- 根据给定的明确索引数，得到的子 Tensor 维度个数将对应地减少。

高级索引方式

 参见

参考 [NumPy Advanced Indexing](#) .