

## 2 张量

LibTorch 简单教程

2025-11-21

# 1 张量模型

张量在 LibTorch 中表示为一种模型 (scheme)，有三个常用属性，分别是 *Value* *Size* *TensorOptions*，这里介绍的三个属性名称并非一一对应源代码，仅作参考使用。但是 *TensorOptions* 是源码真实存在的，可以在官方文档找到详细说明。

## 1.1 Value

*Value* 属性是指张量中存储的数值。众所周知，计算机把数字、文本、图像等人类理解的事物一律表示为二值编码（电压高低离散值）。而人类又使用“二进制计数系统”理解这些二值，所以它们就成为二进制数字了。更底层的一步理解是，使用“布尔逻辑代数”理解这些编码。所以由此可以得出，*Value* 数值包括普遍意义的“数值数字”，也包含字符串文本、图像等。

*Value* 的深层含义解释完了，我们接下来看一下实际使用中的场景。第一个是我们可以从 C++ 原生数值类型转换得到，比如 `int` `float` `double` 等。

第二个是可以从 STL 数据转化，比如 `std::vector::data` 属性获取数据指针。

第三个就从第三方库转化，比如 `cv::Mat::data` 属性获取矩阵或者图像数据指针，从 OpenCV 转化要注意数值归一化，通道顺序符合神经网络模型输入顺序。

## 1.2 Size

*Size* 属性是指张量的维度数量和每个维度索引范围。比如，*Size* = 4\*5\*8 意味着这个张量 `维度=3` 因为有三个数字（或者说两个分割标记），每个维度分配编号或者索引分别为 `0、1、2`；维度索引范围：`第 0 号维度` 取值 `0-1-2-3`，`第 1 号维度` 取值 `0-1-2-3-4`，`第 2 号维度` 取值 `0-1-2-3-4-5-6-7`。这是编程时的用法。通俗的理解就是 4 行 5 列 8 通道图像。

## 1.3 TensorOptions

*TensorOptions* 属性是一个 `class`，定义了多个子属性，具体可以查看源码和文档。这里介绍常用的三个子属性：`dtype` `grad` `device`。

`dtype`是指 *Value* 的存储方式。如果从 `int` 转化而来,则 `dtype = kInt`, 同理 `float → kFloat` `double → kDouble`。`dtype`属性可以有其它取值, 参看文档和源码。`grad`是指在自动微分时该 *Tensor* 是否保留梯度。`device`是指 *Tensor* 在哪个设备运算。常见取值有 `device = kCPU` `device = kCUDA`。

## 2 张量创建

### 2.1 工厂函数

LibTorch 提供了一系列函数快速创建特定的张量,称之为“工厂函数”。详情参看文档。

### 2.2 手动指定

我们使用 `torch::tensor(arg1, arg2)` 函数创建,需要同时手动设置 *Value*、*Size*、*TensorOptions* 属性。`arg1`同时接收 *Value* 和 *Size*, `arg2`接收 *TensorOptions*。其中重点是前两个属性 *Value*、*Size*, *Value* 非常容易理解,就是诸如 1.5 2 3.0 这样不同的值。

详细介绍 *Size* 设置。这个属性是实参 `std::initializer` 推算而来。我们注意观察配对的 `{ }` 和 `,`。

- $3 \rightarrow Size = null$
- $\{ \} \rightarrow Size = 0$
- $\{789\} \rightarrow Size = 1$
- $\{423, 9213, 2647\} \rightarrow Size = 3$
- $\{\{1,2,3\},\{7,8,9\}\} \rightarrow Size = 2 * 3$
- $\{\{1,2,3,4,5\},\{7,8,9,10,11\}\} \rightarrow Size = 2 * 5$
- $\{\{1,2,3,4,5\},\{7,8,9,10,11\},\{21,22,23,24,25\}\} \rightarrow Size = 3 * 5$
- $\{$   
     $\{\{1\},\{2\},\{3\},\{4\},\{5\}\},$   
     $\{\{7\},\{8\},\{9\},\{10\},\{11\}\},$

```
{{21},{22},{23},{24},{25}}
} → Size = 3 * 5 * 1
```

```
• {
  {000,001}, {010,011}, {020,021}, {030,031}, {040,041}},
  {100,101}, {110,111}, {120,121}, {130,131}, {140,141}},
  {200,201}, {210,211}, {220,221}, {230,231}, {240,241}}
} → Size = 3 * 5 * 2
```

我们可以发现配对的 $\{\}$ ——既有 $\{$ 又有 $\}$ 才算配对——会创建一个维度，同时 $,$ 则分割每个维度自己负责的元素。例如  $Size = 2 * 3$  0 号维度的元素有 2 个分别是  $\{1,2,3\}$  和  $\{7,8,9\}$ ，这两个元素被 1 个数量的 $,$ 隔开。它的单个元素由配对花括号 $\{\}$ 和 $,$ 互相协作产生，类比 1 个大集合包含 3 个小集合，那么大集合的元素就成为了小集合，每个小集合包含 5 个实数，小集合的元素就成为实数而不是集合。可见元素是一个相对而言的概念。至此，我们可以总结出规律。 $\{\}$ 创建维度，而 $,$ 确定“元素”数量并且分隔它们。

$Size = 5 * 2 * 4$  的张量应该怎么写呢？

这个张量又是什么形状？

```
{ { { , , }, { , , }, { , , }, { , , }, { , , }, { , , }, { , , }, { , , }, { , , }, { , , } }, { { , , }, { , , }, { , , }, { , , }, { , , }, { , , }, { , , }, { , , } } }
```

## 2.3 数据指针转换

到这里就比较简单了，使用函数

```
torch::from_blob(void* Value , at::IntArrayRef Size , const at::TensorOptions& )
```

在网上可以找到使用方式，其中实参传递给  $Size$  时，是 `std::initializer` 格式——注意，这个和上节的推算不一样，只是长得像——比如  $Size = 3 * 5 * 2 \rightarrow \{3, 5, 2\}$ ，这就确认了张量的  $Size$ 。当只有一个维度时，可以省略花括号， $Size = 3 \rightarrow \{3\} \iff 3$ 。

总之，传参写法就是，花括号包起来并且分隔改为逗号。可以发现，这种方式创建维度  $null$  的张量传入 $\{\}$ 即可。其实在官方文档可以查到， $null$  其实就是标量。

思考一下，我们把视角脱离，不再执着符号外貌。当我们看见类似  $\{\{ \{ , , \}, \{ , , \}, \{ , , \}, \{ , , \}, \{ , , \}, \{ , , \}, \{ , , \}, \{ , , \}, \{ , , \}, \{ , , \} \}$  符号时，以张量推算视角和  $Size$  传参视角会有什么不同的结果？参看下面代码，欢迎补充内容。

```

1 #include <iostream>
2
3 #include <torch/torch.h>
4
5
6 int main()
7 {
8     float a[800] = { 2, 3, .1 };
9
10    //无括号
11    std::cout << torch::from_blob(a, 0) << std::endl;
12    std::cout << torch::from_blob(a, 1) << std::endl;
13    std::cout << torch::from_blob(a, 2) << std::endl;
14    //有括号
15    std::cout << torch::from_blob(a, { }) << std::endl;
16    std::cout << torch::from_blob(a, { 2 }) << std::endl;
17    std::cout << torch::from_blob(a, { 2,5 }) << std::endl;
18    std::cout << torch::from_blob(a, { 2,5,7 }) << std::endl;
19    //嵌套括号
20    std::cout << torch::from_blob(a, { }) << std::endl;
21    std::cout << torch::from_blob(a, { { } }) << std::endl;
22    std::cout << torch::from_blob(a, { { { } } }) << std::endl;
23    /*
24    std::cout << torch::from_blob(a, { { { { } } } }) << std::endl;
25    会报错，最多嵌套三层
26    */
27    std::cout << ( a[1] = { } ) << std::endl;
28    std::cout << torch::from_blob(a, { 0 , }) << std::endl; //末尾逗号
29    std::cout << torch::from_blob(a, { { } , }) << std::endl; //末尾逗号
30    std::cout << torch::from_blob(a, { { } , { } }) << std::endl;
31    std::cout << torch::from_blob(a, { { } , { } , { } }) << std::endl;
32    /*
33    std::cout << torch::from_blob(a, { { { } } , { } , { } , }) << std::endl;
34    报错，运算符优先级问题
35    */
36
37    return 0;
38 }
39

```

Listing 1: { C++ 代码示例 }

## 2.4 转换为非张量

使用 `Tensor::data_ptr<type>()` 获取张量数据指针，`type` 想要的数据类型。还有一个函数 `Tensor::item<type>()` 只针对标量使用。

## 2.5 使用构造函数

构造函数 `torch::Tensor()` 可以直接初始化张量，它有多个重载，参见官方文档。这里涉及一个深浅拷贝问题。赋值构造是浅拷贝，内存中的数据只有单份，一荣俱荣一损俱损。使用 `Tensor::clone()` 函数创建全新的内存空间，以单独存储数据。

# 3 张量计算

## 4 张量索引

## 5 张量形状操作

## 6 张量互相操作

## 7 张量小结