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，仅作理解使用。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。这里介绍常用的子属性：$dtype$ $requires\_grad$ layout $device$。

$dtype$ 是指 $Value$ 的存储方式。如果从 int 转化而来，则 $dtype = kInt$，同理 $float \rightarrow kFloat$ $double \rightarrow kDouble$。$dtype$ 属性可以有其它取值，参看文档和源码。$requires\_grad$ 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 $device = kCPU$ $device = kCUDA$。layout 是指稠密存储还是稀疏存储。常见取值有 $layout = kStrided$ $layout = kSparse$。

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}}
  } $\rightarrow Size = 3 * 5 * 1$
- {
  {{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}}
  } $\rightarrow Size = 3 * 5 * 2$

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(viod* $Value$, 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 } $\Longleftrightarrow$ 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。

```cpp
# include <iostream>
# include <torch/torch.h>


int main()
{
  float a[800] = { 2,  3,   .1 };

  //无括号
  std::cout << torch::from_blob(a, 0) << std::endl;
  std::cout << torch::from_blob(a, 1) << std::endl;
  std::cout << torch::from_blob(a, 2) << std::endl;
  //有括号
  std::cout << torch::from_blob(a, {  }) << std::endl;
  std::cout << torch::from_blob(a, { 2 }) << std::endl;
  std::cout << torch::from_blob(a, { 2,5 }) << std::endl;
  std::cout << torch::from_blob(a, { 2,5,7 }) << std::endl;
  //嵌套括号
  std::cout << torch::from_blob(a, {  }) << std::endl;
  std::cout << torch::from_blob(a, { {} }) << std::endl;
  std::cout << torch::from_blob(a, { {{}} }) << std::endl;
  /*
  std::cout << torch::from_blob(a, {{{{}}}}) << std::endl;
  会报错，最多嵌套三层
  */
  std::cout << ( a[1] = { } ) << std::endl;
  std::cout << torch::from_blob(a, { 0  , }) << std::endl;  //末尾逗号
  std::cout << torch::from_blob(a, { { } , }) << std::endl; //末尾逗号
  std::cout << torch::from_blob(a, { { } , {} }) << std::endl;
  std::cout << torch::from_blob(a, { {    } , {} , {}   }) << std::endl;
  /*
  std::cout << torch::from_blob(a, { { {} } , {} , {} , }) << std::endl;
  报错，运算符优先级问题
  */

  return 0;
}
```

Listing 1: { 数据指针转换 }

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 张量数学运算

3.1 数学运算概述

张量的数学运算主要针对张量的 $Value$ 属性操作，涵盖了算术运算、比较运算、统计运算、数学函数、线性代数操作以及谱操作等内容。在这些运算中，张量的形状（$Size$）会影响运算的方式。对于不同形状的张量，LibTorch 提供了广播机制来确保不同维度的张量能够进行操作。

3.2 算术运算

算术运算是张量计算中最基本的操作，通常包括加法、减法、乘法和除法等运算。

运算	API 函数	功能描述	代码示例
加法	`torch::add()`	逐元素加法	`auto result = torch::add(a, b);`
减法	`torch::sub()`	逐元素减法	`auto result = torch::sub(a, b);`
乘法	`torch::mul()`	逐元素乘法	`auto result = torch::mul(a, b);`
除法	`torch::div()`	逐元素除法	`auto result = torch::div(a, b);`
取余	`torch::remainder()`	逐元素取余	`auto result = torch::remainder(a, b);`
求幂	`torch::pow()`	逐元素计算幂	`auto result = torch::pow(a, 2);`

表 1: 常见算术运算

3.3 比较运算

比较运算用于对两个张量进行元素级别的比较，返回一个布尔类型的张量，表示每个元素的比较结果。

3.4 统计运算

统计运算用于计算张量的各种统计信息，常见的包括最大值、最小值、均值、方差等。

运算	API 函数	功能描述	代码示例
相等	torch::eq()	逐元素相等	auto result = torch::eq(a, b);
不等	torch::ne()	逐元素不相等	auto result = torch::ne(a, b);
大于	torch::gt()	逐元素大于	auto result = torch::gt(a, b);
小于	torch::lt()	逐元素小于	auto result = torch::lt(a, b);
大于等于	torch::ge()	逐元素大于等于	auto result = torch::ge(a, b);
小于等于	torch::le()	逐元素小于等于	auto result = torch::le(a, b);

表 2: 常见比较运算

运算	API 函数	功能描述	代码示例
求和	torch::sum()	所有元素的总和	auto result = torch::sum(a);
平均值	torch::mean()	所有元素平均值	auto result = torch::mean(a);
最大值	torch::max()	所有元素的最大值	auto result = torch::max(a);
最小值	torch::min()	所有元素的最小值	auto result = torch::min(a);
标准差	torch::std()	所有元素的标准差	auto result = torch::std(a);
方差	torch::var()	所有元素的方差	auto result = torch::var(a);
中位数	torch::median()	所有元素的中位数	auto result = torch::median(a);

表 3: 常见统计运算

3.5 数学函数

数学函数广泛用于深度学习中的各种计算，包括对数、指数、平方根等。

运算	API 函数	功能描述	代码示例
指数	torch::exp()	逐元素指数	auto result = torch::exp(a);
对数	torch::log()	逐元素自然对数	auto result = torch::log(a);
平方根	torch::sqrt()	逐元素平方根	auto result = torch::sqrt(a);
绝对值	torch::abs()	逐元素绝对值	auto result = torch::abs(a);
三角函数	torch::sin()	逐元素正弦值	auto result = torch::sin(a);
	torch::cos()	逐元素余弦值	auto result = torch::cos(a);
	torch::tan()	逐元素正切值	auto result = torch::tan(a);
反三角函数	torch::asin()	逐元素反正弦值	auto result = torch::asin(a);
	torch::acos()	逐元素反余弦值	auto result = torch::acos(a);

表 4: 常见数学函数

3.6 线性代数操作

线性代数操作是深度学习中常见的基础运算，主要包括矩阵乘法、转置、行列式、逆矩阵等。

运算	API 函数	功能描述	代码示例
矩阵乘法	torch::matmul()		auto result = torch::matmul(A, B);
点积	torch::dot()	向量点积	auto result = torch::dot(a, b);
转置	torch::t()		auto result = torch::t(mat);
行列式	torch::det()		auto result = torch::det(mat);
逆矩阵	torch::inverse()		auto result = torch::inverse(mat);
范数	torch::norm()	张量的 Lp 范数	auto result = torch::norm(a, 2);
LU 分解	torch::lu()	列主元置换 LU 分解	auto [P,L,U] = torch::lu(a);
QR 分解	torch::qr()		auto [Q,R] = torch::qr(a);

表 5: 常见线性代数操作

3.7 谱操作

谱操作通常用于频域分析等高级应用，如傅里叶变换和特征值分解。

运算	API 函数	功能描述	代码示例
快速傅里叶变换	torch::fft::fft()	一维或多维 fft	auto ret = torch::fft::fft(a, 2);
逆快速傅里叶变换	torch::fft::ifft()		auto ret = torch::fft::ifft(a, 2);
特征值分解	torch::eig()		auto [val,vec] = torch::eig(a);
奇异值分解	torch::svd()	SVD	auto [U,S,V] = torch::svd(a);

表 6: 谱操作

3.8 广播机制

广播机制允许形状不同的张量进行运算，自动扩展较小张量的形状以匹配较大的张量。广播使得不同大小的张量能够在数学运算中配合使用，而无需手动调整其形状。

操作	API 函数	功能描述
广播加法	torch::add()	在不同形状的张量间执行加法操作
广播乘法	torch::mul()	在不同形状的张量间执行乘法操作

表 7: 广播机制示例

4 张量形状操作

张量的形状操作主要是针对其 Size 属性进行的，这些操作对于张量的维度、大小及形状的调整和变换至关重要。常见的形状操作包括张量索引、形状变换和张量拼接拆分操作，具体内容如下。

4.1 张量索引

张量索引操作允许我们根据指定位置访问张量中的特定元素或切片。索引操作是处理和选择数据的基础。

操作	函数	功能描述	示例
元素访问	`Tensor::operator[]()`	数组随机访问	`auto elem = a[0][1];`
切片操作	`Tensor::slice()`	选取连续的元素子集	`auto s = a.slice(0, 0, 2);`
getter	`Tensor::index()`	索引列表访问值	`auto s = a.index({0, 2}, 0);`
setter	`Tensor::index_put_()`	索引列表修改值	`auto b = a.index_put_({None}, 1);`

表 8: 张量索引操作

4.2 形状变换

张量的形状变换操作用于改变张量的维度或结构。LibTorch 提供了多种方法来调整张量的形状，常见的操作包括重塑形状、添加和删除维度等。

操作	函数	功能描述	示例
重塑形状	`torch::view()`	变形，元素个数一致	`auto r = a.view({3, 2});`
安全重塑	`torch::reshape()`	连续内存版变形	`auto r = a.reshape({3, 2});`
添加维度	`torch::unsqueeze()`	指定维度插入 $Size = 1$	`auto e = a.unsqueeze(0);`
删除维度	`torch::squeeze()`	删除 $Size = 1$ 的轴	`auto s = a.squeeze();`
重排维度	`torch::permute()`	按索引排列全部维度	`auto d = torch::permute(A,{0,2,1});`
交换维度	`torch::transpose()`	交换任意两个维度	`auto t = a.transpose(1, 2);`

表 9: 形状变换操作

4.3 张量拼接拆分

张量拼接和拆分操作允许我们将多个张量沿指定维度进行组合或分割。常用于数据的合并或划分。

操作	函数	功能描述	示例
堆叠张量	`torch::stack()`	多个张量沿新维度堆叠	`auto s = torch::stack({a, b}, 0);`
拼接张量	`torch::cat()`	多个张量沿指定维度拼接	`auto c = torch::cat({a, b}, 0);`
拆分张量	`torch::split()`	沿指定维度拆分	`auto s = a.split(2, 0);`
均匀拆分	`torch::chunk()`	沿指定维度拆分为指定数量的均匀块	`auto ck = a.chunk(3, 0);`

表 10: 张量拼接拆分操作

5 张量选项操作

选项操作是指设置 *TensorOptions* 属性的 API。torch::TensorOptions 是一个class类型，有 4 个成员函数：dtype() layout() device() requires_grad()。这些成员函数会返回调用对象的引用，仍然是 torch::TensorOptions类型，它们的参数可以设定调用对象的属性。这些属性都具有默认值，分别是dtype=kFloat32 layout=kStrided device=kCPU requires_grad=false。

5.1 创建前指定

共有三种方式指定属性。手动指定全部属性，使用这样的语法。

```
auto options = torch::TensorOptions()
    .dtype(torch::kFloat32)
    .layout(torch::kStrided)
    .device(torch::kCUDA, 1)
    .requires_grad(true);
```

也可以使用torch::命名空间下的属性函数，传递参数指定某个属性。函数原型分别是

torch::TensorOptions torch::dtype() torch::TensorOptions torch::layout()

torch::TensorOptions torch::device() torch::TensorOptions torch::requires_grad()

比如torch::ones(10, torch::TensorOptions().dtype(torch::kFloat32).layout(torch::kStrided))

$\Longleftrightarrow$ torch::ones(10, torch::dtype(torch::kFloat32).layout(torch::kStrided))

当只需要指定单个属性时，可以直接赋值属性，会有隐式转换。比如torch::ones(10, torch::kFloat32)

$\Longleftrightarrow$ torch::ones(10, torch::dtype(torch::kFloat32))

$\Longleftrightarrow$ torch::ones(10, torch::TensorOptions().dtype(torch::kFloat32))

5.2 现有张量转换

以上是创建前设定选项，还可以从现有张量转换选项得到新张量。新张量会另外开辟空间存储数据。使用Tensor::to()函数指定新张量的选项。

- 改变 *dtype*

  torch::Tensor source_tensor = torch::randn(2, 3, torch::kInt64);

  torch::Tensor float_tensor = source_tensor.to(torch::kFloat32);

- 改变 *device*

  torch::Tensor gpu_two_tensor = float_tensor.to(torch::Device(torch::kCUDA, 1));

如果有多个 GPU，可以创建一个torch::DeviceGuard对象指定默认 GPU 设备。代码如下：

```
#include <torch/torch.h>
#include <iostream>

int main() {
    // Check the number of available GPUs
    int device_count = torch::cuda::device_count();
```

```
    std::cout << "Number of CUDA devices available: " << device_count << std::endl;

    if (device_count > 1) {
        // Use DeviceGuard to set the current GPU to GPU 1 (if you have multiple GPUs)
        {
            torch::DeviceGuard guard(torch::Device(torch::kCUDA, 1 ));  // Set active device to GPU 1

            // Create a tensor on GPU 1
            torch::Tensor tensor_on_gpu1 = torch::rand({ 3, 3 }, torch::device({torch::kCUDA, 1}));
            std::cout << "Tensor on GPU 1:\n" << tensor_on_gpu1 << std::endl;
        }

        // Now GPU 0 is active again, no need to manually change back
        torch::Tensor tensor_on_gpu0 = torch::rand({ 3, 3 }, torch::device({torch::kCUDA, 0}));
        std::cout << "Tensor on GPU 0:\n" << tensor_on_gpu0 << std::endl;
    }

    return 0;
}
```

Listing 2: { 全局或局部默认 GPU }

6 张量小结