libtensor

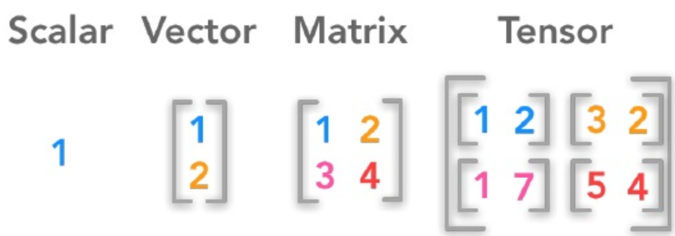
Designer: Zean He, Ziyang Leng

1. 概述

在这个项目中，你需要在C/ c++中实现一个张量库，支持基本的张量操作和一些高级功能，如序列化计算加速。

2. 定义

张量是一个包含单一数据类型元素的多维矩阵。在这个项目中，我们只考虑实数的张量。



大小为[2,2,2]的张量

3. 要求

> 由于张量可以是高维的，我们强烈建议您实现张量的基本属性函数，例如size()、type()和data_ptr()，以方便地演示您实现的正确性。这对于判断内存管理的正确性尤为重要。
>
> ```
> // Example
> ts::Tensor t = ts::tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]]);
> std::cout << t.size() << std::endl << t.type() << std::endl << t.data_ptr() <<
> std::endl;
> // Output here is just an example, just make it easy to understand.
> [3, 2]
> float
> 0x7f8b1c000000
> ```

3.1 基本要求(90分)

- **1. 张量的创建和初始化(15分)**
  从给定的数组或形状实现具有不同维度和数据类型(例如float, double, int, bool等)的张量的创建操作。你至少应该实现以下功能:
  - 1.1通过将数据复制到你的内存中，从给定的数组中创建一个张量(2分)

```
ts::张量t = ts::张量(t data[]);
```

```
// Example
ts::Tensor t = ts::tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]]);
std::cout << t << std::endl;
// Output
[[ 0.1000,  1.2000],
 [ 2.2000,  3.1000],
 [ 4.9000,  5.2000]]
```

- 1.2用给定的形状和数据类型创建一个张量并随机初始化(3 pts)

```
ts::张量t = ts::rand< t >(int size[]);
```

```
/ /实例
ts::张量t = ts::rand([2,3]);
Std::cout << t << Std::endl;
/ /输出
[[0.1000,      1.2000),
  [2.2000,     3.1000),
  [4.9000,     5.2000]]
```

- 1.3用给定的形状和数据类型创建一个张量，并用给定的值初始化它(6 pts)

```
ts::Tensor t = ts::zeros<T>(int size[]);
ts::Tensor t = ts::ones<T>(int size[]);
ts::Tensor t = ts::full(int size[], T value);
```

```
// Example
ts::Tensor t1 = ts::zeros([2, 3]);
ts::Tensor t2 = ts::ones([2, 3]);
ts::Tensor t3 = ts::full([2, 3], 0.6);
std::cout << t1 << std::endl << t2 << std::endl << t3 << std::endl;
// Output
[[ 0.0000,  0.0000,  0.0000],
 [ 0.0000,  0.0000,  0.0000]]

[[ 1.0000,  1.0000,  1.0000],
 [ 1.0000,  1.0000,  1.0000]]

[[ 0.6000,  0.6000,  0.6000],
 [ 0.6000,  0.6000,  0.6000]]
```

- 1.4用给定的形状和数据类型创建一个张量，并将其初始化为特定的模式(4 pts) ts::张量t = ts::eye< t >(int size[]);//这创建了一个单位矩阵。

```
// Example
ts::Tensor t = ts::eye([3, 3]);
std::cout << t << std::endl;
// Output
[[ 1.0000,   0.0000,   0.0000],
 [ 0.0000,   1.0000,   0.0000],
 [ 0.0000,   0.0000,   1.0000]]
```

- **2. 张量操作(40分)**
实现张量的索引、切片、连接、变异操作。**对于索引、切片、mutating、permutating和查看操作，你应该在不显式复制底层存储的情况下实现，但以不同的形状返回对相同数据的引用**。你至少应该实现以下功能:

  - 2.1索引和切片操作(5分)

```
ts::Tensor t = ts::tensor(T data[]);
ts::Tensor t1 = t(1); // This indexes the second element of t.
ts::Tensor t2 = t(2,{2,4}); // This slices the third to fifth (excluded) elements of the third dimension of t.
```

```
// Example
ts::Tensor t = ts::tensor([[0.1, 1.2, 3.4, 5.6, 7.8], [2.2, 3.1, 4.5, 6.7, 8.9], [4.9, 5.2, 6.3, 7.4, 8.5]]);
std::cout << t(1) << std::endl << t(2,{2,4}) << std::endl;
// Output
[ 2.2000,   3.1000,   4.5000,   6.7000,   8.9000]

[ 6.3000,   7.4000]
```

  - 2.2加入操作(10分)

```
ts::Tensor t1 = ts::tensor(T data1[]);
ts::Tensor t2 = ts::tensor(T data2[]);
ts::Tensor t3 = ts::cat({t1, t2}, int dim); // This joins t1 and t2 along the given dimension.
ts::Tensor t4 = ts::tile(t1, int dims[]); // This construct t4 by repeating the elements of t1
```

```
// Example
ts::Tensor t1 = ts::tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]]);
ts::Tensor t2 = ts::tensor([[0.2, 1.3], [2.3, 3.2], [4.8, 5.1]]);
```

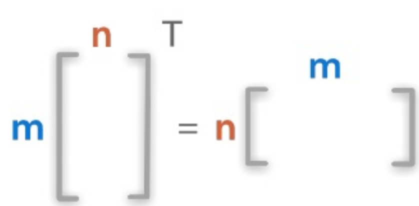
```
std::cout << ts::cat({t1, t2}, 0) << std::endl << ts::cat({t1, t2}, 1) << std::endl
<< ts::tile(t1, {2,2}) << std::endl;
// Output
[[ 0.1000,  1.2000],
 [ 2.2000,  3.1000],
 [ 4.9000,  5.2000],
 [ 0.2000,  1.3000],
 [ 2.3000,  3.2000],
 [ 4.8000,  5.1000]]

[[ 0.1000,  1.2000,  0.2000,  1.3000],
 [ 2.2000,  3.1000,  2.3000,  3.2000],
 [ 4.9000,  5.2000,  4.8000,  5.1000]]

[[ 0.1000,  1.2000,  0.1000,  1.2000],
 [ 2.2000,  3.1000,  2.2000,  3.1000],
 [ 4.9000,  5.2000,  4.9000,  5.2000],
 [ 0.1000,  1.2000,  0.1000,  1.2000],
 [ 2.2000,  3.1000,  2.2000,  3.1000],
 [ 4.9000,  5.2000,  4.9000,  5.2000]]
```

- 2.3变异操作(5分)

```
ts::Tensor t = ts::tensor(T data[]);
t(1) = 1; // This sets the second element of t to 1.
t(2,{2,4}) = [1,2]; // This sets the third to fifth (excluded) elements of the third
dimension of t to [1,2].
```

```
// Example
ts::Tensor t = ts::tensor([[0.1, 1.2, 3.4, 5.6, 7.8], [2.2, 3.1, 4.5, 6.7, 8.9],
[4.9, 5.2, 6.3, 7.4, 8.5]]);
t(1) = 1;
t(2,{2,4}) = {1,2};
std::cout << t << std::endl;
// Output
[[ 0.1000,  1.2000,  3.4000,  5.6000,  7.8000],
 [ 1.0000,  1.0000,  1.0000,  1.0000,  1.0000],
 [ 4.9000,  5.2000,  1.0000,  2.0000,  8.5000]]
```

- 2.4转置和置换操作(10分)



转置

```
ts::Tensor t = ts::tensor(T data[]);
ts::Tensor t1 = ts::transpose(t, int dim1, int dim2);
// This transposes the tensor t along the given dimensions.
ts::Tensor t2 = t.transpose(int dim1, int dim2);
// Another way to transpose the tensor t.
ts::Tensor t3 = ts::permute(t, int dims[]);
// This permutes the tensor t according to the given dimensions.
ts::Tensor t4 = t.permute(int dims[]);
// Another way to permute the tensor t.
```

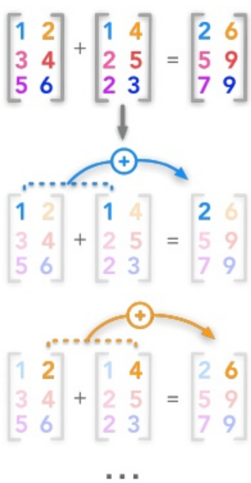
```
// Example
ts::Tensor t = ts::tensor([[0.1, 1.2, 3.4, 5.6, 7.8], [2.2, 3.1, 4.5, 6.7, 8.9],
[4.9, 5.2, 6.3, 7.4, 8.5]]);
std::cout << ts::transpose(t, 0, 1) << std::endl << ts::permute(t, [1, 0]) <<
std::endl;
// Output
[[ 0.1000,  2.2000,  4.9000],
 [ 1.2000,  3.1000,  5.2000],
 [ 3.4000,  4.5000,  6.3000],
 [ 5.6000,  6.7000,  7.4000],
 [ 7.8000,  8.9000,  8.5000]]

[[ 0.1000,  2.2000,  4.9000],
 [ 1.2000,  3.1000,  5.2000],
 [ 3.4000,  4.5000,  6.3000],
 [ 5.6000,  6.7000,  7.4000],
 [ 7.8000,  8.9000,  8.5000]]
```

- 2.5视图操作(10分)

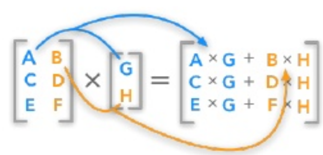
```
ts::Tensor t = ts::tensor(T data[]);
ts::Tensor t3 = ts::view(t, int shape[]); // This views the tensor t according to
the given shape.
ts::Tensor t4 = t.view(int shape[]); // Another way to view the tensor t.
```

```
// Example
ts::Tensor t = ts::tensor([[0.1, 1.2, 3.4, 5.6, 7.8], [2.2, 3.1, 4.5, 6.7, 8.9],
[4.9, 5.2, 6.3, 7.4, 8.5]]);
std::cout << ts::view(t, [5, 3]) << std::endl << t.view([1, 15]) << std::endl;
// Output
[[ 0.1000,  1.2000,  3.4000],
 [ 5.6000,  7.8000,  2.2000],
 [ 3.1000,  4.5000,  6.7000],
 [ 8.9000,  4.9000,  5.2000],
 [ 6.3000,  7.4000,  8.5000]]

[[ 0.1000,  1.2000,  3.4000,  5.6000,  7.8000,  2.2000,  3.1000,  4.5000,  6.7000,
 8.9000,  4.9000,  5.2000,  6.3000,  7.4000,  8.5000]]
```

- **3. 数学运算(35分)**
  实现张量的数学运算。你至少应该实现以下函数:
  - 3.1逐点操作包括add, sub, mul, div, log(5分)



除了



点积

```
ts::Tensor t1 = ts::tensor(T data1[]);
ts::Tensor t2 = ts::tensor(T data2[]);
ts::Tensor t3 = ts::add(t1, t2); // This adds t1 and t2 element-wise.
ts::Tensor t4 = t1.add(t2); // Another way to add t1 and t2 element-wise.
ts::Tensor t5 = t1 + t2; // Another way to add t1 and t2 element-wise.
ts::Tensor t6 = ts::add(t1, T value); // This adds t1 and a scalar value element-wise.
ts::Tensor t7 = t1.add(T value); // Another way to add t1 and a scalar value element-wise.
// ... Similar for sub, mul, div, log.
```

```
// Example
ts::Tensor t1 = ts::tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]]);
ts::Tensor t2 = ts::tensor([[0.2, 1.3], [2.3, 3.2], [4.8, 5.1]]);
std::cout << t1 + t2 << std::endl << ts::add(t1, 1) << std::endl;
// Output
[[ 0.3000,  2.5000],
 [ 4.5000,  6.3000],
 [ 9.7000, 10.3000]]

[[ 1.1000,  2.2000],
 [ 3.2000,  4.1000],
 [ 5.9000,  6.2000]]
```

- 3.2还原操作包括求和、均值、最大值、最小值(5分)

```
ts::Tensor t = ts::tensor(T data[]);
ts::Tensor t1 = ts::sum(t, int dim);// This sums the tensor t along the given dimension.
ts::Tensor t2 = t.sum(int dim);// Another way to sum the tensor t along the given dimension.
// ... Similar for mean, max, min.
```

```
// Example
ts::Tensor t = ts::tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]]);
std::cout << ts::sum(t, 0) << std::endl << t.sum(1) << std::endl;
// Output
[ 7.2000,  9.5000]

[ 1.3000,  5.3000, 10.1000]
```

- 3.3比较操作包括eq, ne, gt, ge, lt, le(10分)

```
ts::Tensor t1 = ts::tensor(T data1[]);
ts::Tensor t2 = ts::tensor(T data2[]);
ts::Tensor<bool> t3 = ts::eq(t1, t2);// This compares t1 and t2 element-wise.
ts::Tensor t4<bool> = t1.eq(t2);// Another way to compare t1 and t2 element-wise.
ts::Tensor t5<bool> = t1 == t2;// Another way to compare t1 and t2 element-wise.
// ... Similar for ne, gt, ge, lt, le.
```

```
// Example
ts::Tensor t1 = ts::tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]]);
ts::Tensor t2 = ts::tensor([[0.2, 1.3], [2.2, 3.2], [4.8, 5.2]]);
std::cout << (t1 == t2) << std::endl;
// Output
[[ False,  False],
 [  True,  False],
 [ False,   True]]
```

- 3.4其他操作包括einsum(15分)

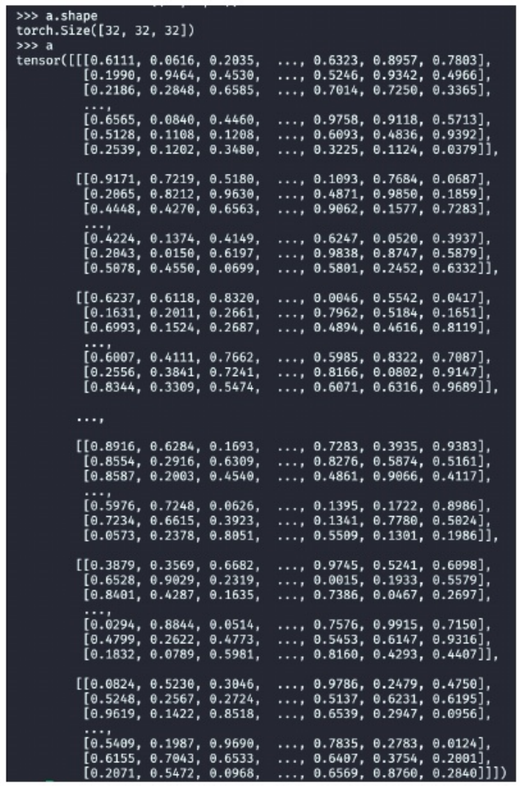
```
ts::Tensor t1 = ts::tensor(T data1[]);
ts::Tensor t2 = ts::tensor(T data2[]);
ts::Tensor t3 = ts::einsum("i,i->", t1, t2); // This computes the dot product of t1 and t2.
ts::Tensor t4 = ts::einsum("i,i->i", t1, t2); // This computes the element-wise product of t1 and t2.
ts::Tensor t5 = ts::einsum("ii->i", t1); // This computes the diagonal of t1.
ts::Tensor t6 = ts::einsum("i,j->ij", t1, t2); // This computes the outer product of t1 and t2.
ts::Tensor t7 = ts::einsum("bij,bjk->bik", t1, t2); // This computes the batch matrix multiplication of t1 and t2.
```

```
// Example
ts::Tensor t1 = ts::tensor([1, 2, 3]);
ts::Tensor t2 = ts::tensor([4, 5, 6]);
std::cout << ts::einsum("i,i->", t1, t2) << std::endl << ts::einsum("i,i->i", t1, t2) << std::endl;
// Output
32

[ 4, 10, 18]
```

3.2 高级要求(共20分)

- **1. 连载(5分)**
  实现张量的序列化操作，包括保存和加载。

```
ts::Tensor t = ts::tensor(T data[]);
ts::save(t, string filename); // This saves the tensor t to the given file.
ts::Tensor t1 = ts::load(string filename); // This loads the tensor t from the given
file.
std::cout << t << std::endl; // This should pretty-print the tensor t.
```



形式打印

- **2. 计算加速(15分)**
  实现张量的计算加速。加速可以通过使用硬件(如CUDA, MKL, SIMD)或软件(如OpenMP)来实现。应该提供原始实现和加速实现之间的比较，或者如果您有足够的信心，可以直接与PyTorch进行比较。

- **3.。梯度支持(20分)**
  实现张量的梯度支持。

- **4. 即兴创作(?分)**
  实现其他你认为对张量有用的功能。

4. 引用

- PyTorch 图 片
- 来源