libtensor

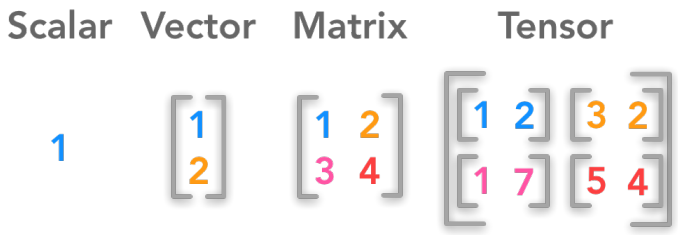
Designer: Zean He, Ziyang Leng

1. Overview

In this project, you need to implement a tensor library in C/C++ that supports basic tensor operations and some advanced features like serialization computing acceleration.

2. Definition

A tensor is a multi-dimensional matrix containing elements of a single data type. In this project, we only consider tensors of real numbers.



Tensor of size [2, 2, 2]

3. Requirements

> As tensors can be high-dimensional, we strongly recommend you implement basic attribute functions for tensors, such as `size()`, `type()`, and `data_ptr()` to demonstrate the correctness of your implementation conveniently. This is particularly important for judging the correctness of memory management.

```
// 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 Basic Requirements (90 pts)

- **1. Tensor Creation and Initialization (15 pts)**
  Implement `creation operations` of tensors with different dimensions and data types (e.g. float, double, int, bool, etc.) from a given array or shape. You should at least implement the following functions:
  - 1.1 Create a tensor from a given array by copying data to your memory (2 pts)

```
ts::Tensor t = ts::tensor(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 Create a tensor with a given shape and data type and initialize it randomly (3 pts)

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

```
// Example
ts::Tensor t = ts::rand([2, 3]);
std::cout << t << std::endl;
// Output
[[ 0.1000,  1.2000],
 [ 2.2000,  3.1000],
 [ 4.9000,  5.2000]]
```

- 1.3 Create a tensor with a given shape and data type, and initialize it with a given value (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 Create a tensor with a given shape and data type, and initialize it to a specific pattern (4 pts)

```
ts::Tensor t = ts::eye<T>(int size[]); // This creates an identity matrix.
```

```
// 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. Tensor Operations (40 pts)**

  Implement `indexing`, `slicing`, `joining`, `mutating` operations for tensors. **For the `indexing`, `slicing`, `mutating`, `permuting`, and `viewing` operations, you should implement without explicitly copying the underlying storage but return the reference to the same data with a different shape.** You should at ~~least~~ implement the following functions:

  - 2.1 `Indexing` and `slicing` operations (5 pts)

```
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 `Joining` operations (10 pts)

```
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]]);
```

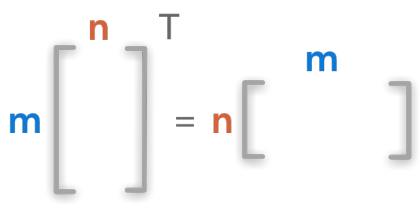
```
std::cout << ts::cat({t1, t2}, 0) << std::endl << ts::cat({t1, t2}, 1) << std::endl
<< ts::tile(t1, {2,2}) << std::endl;
// Output
[[ 0.1000,  1.2000],
 [ 2.2000,  3.1000],
 [ 4.9000,  5.2000],
 [ 0.2000,  1.3000],
 [ 2.3000,  3.2000],
 [ 4.8000,  5.1000]]

[[ 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 `Mutating` operations (5 pts)

```
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 `Transpose` and `permute` operations (10 pts)



Transpose

```
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.
```

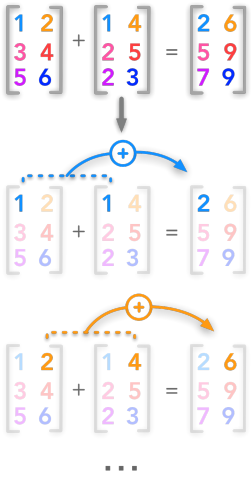
```
// Example
ts::Tensor t = ts::tensor([[0.1, 1.2, 3.4, 5.6, 7.8], [2.2, 3.1, 4.5, 6.7, 8.9],
[4.9, 5.2, 6.3, 7.4, 8.5]]);
std::cout << ts::transpose(t, 0, 1) << std::endl << ts::permute(t, [1, 0]) <<
std::endl;
// Output
[[ 0.1000,  2.2000,  4.9000],
 [ 1.2000,  3.1000,  5.2000],
 [ 3.4000,  4.5000,  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 `View` operations (10 pts)

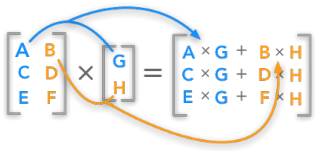
```
ts::Tensor t = ts::tensor(T data[]);
ts::Tensor t3 = ts::view(t, int shape[]); // This views the tensor t according to
the given shape.
ts::Tensor t4 = t.view(int shape[]); // Another way to view the tensor t.
```

```
// 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. Math Operations (35 pts)**
  Implement math operations for tensors. You should at least implement the following functions:
  - 3.1 Pointwise operations including `add`, `sub`, `mul`, `div`, `log` (5 pts)



Addition



Dot product

```
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 Reduction operations including `sum`, `mean`, `max`, `min` (5 pts)

```
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 Comparison operations including `eq`, `ne`, `gt`, `ge`, `lt`, `le` (10 pts)

```
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 Other operations including `einsum` (15 pts)

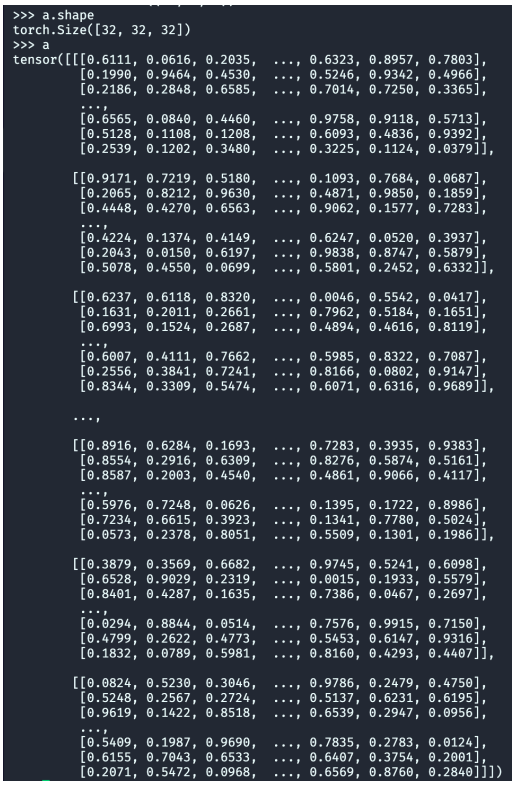
```
ts::Tensor t1 = ts::tensor(T data1[]);
ts::Tensor t2 = ts::tensor(T data2[]);
ts::Tensor t3 = ts::einsum("i,i->", t1, t2); // This computes the dot product of t1
and t2.
ts::Tensor t4 = ts::einsum("i,i->i", t1, t2); // This computes the element-wise
product of t1 and t2.
ts::Tensor t5 = ts::einsum("ii->i", 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 Advanced Requirements (up to 20 pts in total)

- **1. Serialization (5 pts)**
  Implement `serialization` operations for tensors including `save` and `load`.

```
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.
```



Pretty-print

- **2. Computing Acceleration (15 pts)**
  Implement computing acceleration for tensors. Acceleration can be achieved by using hardware (e.g. CUDA, MKL, SIMD) or software (e.g. OpenMP). Comparisons between raw implementation and accelerated implementation should be provided, or directly compared with PyTorch if you are confident enough.
- **3. Gradient Support (20 pts)**
  Implement `gradient support` for tensors.
- **4. Improvising (? pts)**
  Implement other features that you think are useful for tensors.

4. References

- PyTorch
- Image Source