Notes on libtorch

Alexandre Foley

July 30, 2020

Contents

1 Linear Algebra operations **1**

2 Tensor class structure **2**
2.1 Sparse tensors 2

3 Complex numbers **2**

1 Linear Algebra operations

For tensors of real numbers, the number of decomposition and operations available is extensive. SVD,Eigenvalues,QR,LU and more are present. Of note: all matrix operation act on the last 2 indices of a tensor, the other index are treated as labeling independent matrices. For exemple:

```
auto A = torch::rand({4,3,4,4});
autop [E,U] = A.eig();
```

will treat this as solving the eigenvalue problem of $4 \times 3 = 12$ size 4 matrices. Therefor E is of dimension $\{4, 3, 4\}$ and U of dimension $\{4, 3, 4, 4\}$.

The equivalent of dmrjulia's contract is torch::tensordot. It differ in two major way: order of arguments, and no overload to permute the output tensor.

```
auto A = torch::rand({5,3,4});
auto B = torch::rand({4,3,5});
auto i_A = {1,2};
auto i_B = {1,0};
auto C = torch::tensordot(A,B,i_A,i_B); // 5x5 matrix
```

The arguments are the 2 input tensor, and then the lists of index to contract in each tensor.

An arbitrary permutation can be done with torch::Tensor::permute, and tensors can be reshaped with torch::Tensor::reshape.

2 Tensor class structure

The tensors of pytorch are build in a somewhat complicated way to offer a simple and uniform interface no matter what is the actual backend on which the computations are done. This is accomplished by using the "pointer to implementation" idiom, "pimpl" for short. The torch::Tensor class is only a thin wrapper to a polymorphic pointer on the actual tensor, that tensor type depends on the actual backend but are all derived from the same base class. Those different tensor types therefor must offer the same interface. Done this way, the end user gets all the advantages of the multiple backends with none of the additionnal difficulty associated with the differents type, not even polymorphic pointers.

While hiding a quantum tensor behind this interface is not impossible, any specific properties of quantum tensors would be innacessible. This means that constructing a quantum tensor would have to be done through a factory function, and from then on the user cannot do any direct manipulation to a property specific to quantum tensor.

2.1 Sparse tensors

Torch implement a sparse tensor class. The format is similar to the implementation of the quantum tensor of DMRjulia: The non-zero coefficient and their indices are stored densely.

Sparse tensor, much like any potential quantum tensor implementation, require some differences from dense tensor in their interface. Emulating pytorch's implementers approach to these class for our quantum tensor is almost certainly the best way forward.

Also of note: augmenting this sparse tensor class for quantum tensor would closely emulate DMRjulia implementation.

3 Complex numbers

At the moment, complex numbers are badly supported by libtorch and not supported in the current release of pytorch.

Expression that contains both real numbers and complex numbers will cause type compatibility issues, explicit conversion to complex is currently necessary.

The number of supported linear algebra operations is limited. Currently SVD is supported on both CPU and GPU. Eigen values is unavaible at the time of writing this.

The current status of Complex number support is tracked in issue 33152