Tensor Basics

The ATen tensor library backing PyTorch is a simple tensor library thats exposes the Tensor operations in Torch directly in C++14. ATen's API is auto-generated from the same declarations PyTorch uses so the two APIs will track each other over time.

Tensor types are resolved dynamically, such that the API is generic and does not include templates. That is, there is one `Tensor` type. It can hold a CPU or CUDA Tensor, and the tensor may have Doubles, Float, Ints, etc. This design makes it easy to write generic code without templating everything.

See https://pytorch.org/cppdocs/api/namespace_at.html#functions for the provided API. Excerpt:

```
Tensor atan2(const Tensor & other) const;
Tensor & atan2_(const Tensor & other);
Tensor pow(Scalar exponent) const;
Tensor pow(const Tensor & exponent) const;
Tensor & pow_(Scalar exponent);
Tensor & pow_(const Tensor & exponent);
Tensor lerp(const Tensor & end, Scalar weight) const;
Tensor & lerp_(const Tensor & end, Scalar weight);
Tensor histc() const;
Tensor histc(int64_t bins) const;
Tensor histc(int64_t bins, Scalar min) const;
Tensor histc(int64_t bins, Scalar min, Scalar max) const;
```

In place operations are also provided, and always suffixed by _ to indicate they will modify the Tensor.

Efficient Access to Tensor Elements

When using Tensor-wide operations, the relative cost of dynamic dispatch is very small. However, there are cases, especially in your own kernels, where efficient element-wise access is needed, and the cost of dynamic dispatch inside the element-wise loop is very high. ATen provides *accessors* that are created with a single dynamic check that a Tensor is the type and number of dimensions. Accessors then expose an API for accessing the Tensor elements efficiently.

Accessors are temporary views of a Tensor. They are only valid for the lifetime of the tensor that they view and hence should only be used locally in a function, like iterators.

Note that accessors are not compatible with CUDA tensors inside kernel functions. Instead, you will have to use a *packed accessor* which behaves the same way but copies tensor metadata instead of pointing to it.

It is thus recommended to use *accessors* for CPU tensors and *packed accessors* for CUDA tensors.

CPU accessors

```
torch::Tensor foo = torch::rand({12, 12});

// assert foo is 2-dimensional and holds floats.
auto foo_a = foo.accessor<float,2>();
float trace = 0;

for(int i = 0; i < foo_a.size(0); i++) {
  // use the accessor foo_a to get tensor data.
  trace += foo_a[i][i];
}
```

CUDA accessors

```
__global__ void packed_accessor_kernel(
    torch::PackedTensorAccessor64<float, 2> foo,
    float* trace) {
  int i = threadIdx.x;
  gpuAtomicAdd(trace, foo[i][i]);
}

torch::Tensor foo = torch::rand({12, 12});

// assert foo is 2-dimensional and holds floats.
auto foo_a = foo.packed_accessor64<float,2>();
float trace = 0;

packed_accessor_kernel<<<1, 12>>>(foo_a, &trace);
```

In addition to `PackedTensorAccessor64` and `packed_accessor64` there are also the corresponding `PackedTensorAccessor32` and `packed_accessor32` which use 32-bit integers for indexing. This can be quite a bit faster on CUDA but may lead to overflows in the indexing calculations.

Note that the template can hold other parameters such as the pointer restriction and the integer type for indexing. See documentation for a thorough template description of *accessors* and *packed accessors*.

Using Externally Created Data

If you already have your tensor data allocated in memory (CPU or CUDA), you can view that memory as a `Tensor` in ATen:

```
float data[] = { 1, 2, 3,
                 4, 5, 6 };
torch::Tensor f = torch::from_blob(data, {2, 3});
```

These tensors cannot be resized because ATen does not own the memory, but otherwise behave as normal tensors.

Scalars and zero-dimensional tensors

In addition to the `Tensor` objects, ATen also includes `Scalar`s that represent a single number. Like a Tensor, Scalars are dynamically typed and can hold any one of ATen's number types. Scalars can be implicitly constructed from C++ number types. Scalars are needed because some functions like `addmm` take numbers along with Tensors and expect these numbers to be the same dynamic type as the tensor. They are also used in the API to indicate places where a function will *always* return a Scalar value, like `sum`.

```
namespace torch {
Tensor addmm(Scalar beta, const Tensor & self,
             Scalar alpha, const Tensor & mat1,
             const Tensor & mat2);
Scalar sum(const Tensor & self);
} // namespace torch

// Usage.
torch::Tensor a = ...
torch::Tensor b = ...
torch::Tensor c = ...
torch::Tensor r = torch::addmm(1.0, a, .5, b, c);
```

In addition to `Scalar`s, ATen also allows `Tensor` objects to be zero-dimensional. These Tensors hold a single value and they can be references to a single element in a larger `Tensor`. They can be used anywhere a `Tensor` is expected. They are normally created by operators like *select* which reduce the dimensions of a `Tensor`.

```
torch::Tensor two = torch::rand({10, 20});
two[1][2] = 4;
// ^^^^^^ <- zero-dimensional Tensor
```