This directory contains the low-level tensor libraries for PyTorch, as well as the new ATen C++ bindings.

The low-level libraries trace their lineage from the original Torch. There are multiple variants of the library, summarized here:

- TH = TorcH
- THC = TorcH Cuda
- THCS = TorcH Cuda Sparse (now defunct)
- THNN = TorcH Neural Network (now defunct)
- THS = TorcH Sparse (now defunct)

(You'll also see these abbreviations show up in symbol names.)

Reference counting

PyTorch employs reference counting in order to permit tensors to provide differing views on a common underlying storage. For example, when you call view() on a Tensor, a new THTensor is allocated with differing dimensions, but it shares the same c10::StorageImpl with the original tensor.

Unfortunately, this means we are in the business of manually tracking reference counts inside our C library code. Fortunately, for most of our library code implementing tensor operations, there is only one rule you have to remember:

> ***Golden Rule of Reference Counting:*** *You must either FREE or RETURN a pointer which was returned by a function whose name begins with `new` or which you called `retain` on. If you return this pointer, your function name must begin with `new`.*

In a long function, there may be many invocations of functions with `new` in their name. Your responsibility is to go through each of them and ensure that there is a matching `free` for it for EACH exit point of the function.

Examples

Suppose you want to get a reference to the indices of a sparse tensor. This function is called `newIndices`. The `new` means you MUST free it when you're done (usually at the end of your function.) (It's worth noting that `newIndices` doesn't actually allocate a fresh indices tensor; it just gives you a pointer to the existing one.) DO NOT directly access the member variables of the struct.

```
THIndexTensor *indices = THSTensor_(newIndices)(state, sparse);
// ... do some stuff ...
THIndexTensor_(free)(state, indices);
```

Let's take a look at the implementation of `newIndices`. This doesn't free the return result of `newNarrow`, but returns it. This justifies the `new` in its name.

```
THIndexTensor *THSTensor_(newIndices)(const THSTensor *self) {
  // ...
  return THIndexTensor_(newNarrow)(self->indices, 1, 0, self->nnz);
}
```

Passing an object to another function does NOT absolve you of responsibility of freeing it. If that function holds on to a pointer to the object, it will `retain` it itself.

```
  THByteStorage *inferred_size = THByteStorage_newInferSize(size, numel);
  THTensor_(setStorage)(self, tensor->storage, tensor->storageOffset, inferred_size,
NULL);
  c10::raw::intrusive_ptr::decref(inferred_size);
```

Sometimes, you have a tensor in hand which you'd like to use directly, but under some conditions you have to have to call, e.g., `newContiguous`, to get it into the correct form:

```
  if (!(k_->stride(3) == 1) || !(k_->stride[2] == k_->size(3))) {
    kernel = THTensor_(newContiguous)(k_);
  } else {
    THTensor_(retain)(k_);
    kernel = k_;
  }
  ...
  c10::raw::intrusive_ptr::decref(kernel);
```

In this case, we have (redundantly) called `retain` on `k_`, so that we can unconditionally free `kernel` at the end of the function; intuitively, you want it to be possible to replace the conditional expression with an equivalent function call, e.g., `kernel = THTensor_(newContiguous2D)(k_)`.

Tips

- If you have an early exit in a function (via a `return`), don't forget to `free` any pointers which you allocated up to this point. If at all possible, move early exits prior to these allocations, so that you don't have to clean up.
- Very occasionally, you may be able to implement an algorithm more efficiently if you "destroy" its input. This is a `move`; after moving an object away, you must NOT `free` it. This is the one exception to the rule, and at the moment there is only one instance of `move` in the code base.
- We use `THError` to signal error cases, and fortunately, you do NOT need to make sure you've freed everything before calling `THError`, because by default, it aborts the entire process. However, it's good style to call `THError` before performing any allocations, since in some cases we sketchily throw a C++ exception and try to recover (in particular, the test suite does this.)