You're here because you are working on a task involving porting legacy TH code into ATen. This document provides some useful resources for going about doing ports.

Preliminaries

- **Check that the code is live.** Some functions in THNN are dead. An easy check is to make sure the function shows up in the documentation (temporal = 1d, spatial = 2d, volumetric = 3d). A more thorough check is to stick a purposeful exception in the kernel in question, and then push your change to a PR to let CI run on it. If CI is all green even after you broke the code, it's a pretty strong indication it's dead.

The general recipe

Let us take `max_unpool1d` as an example. max_unpool1d, max_unpool2d, and max_unpool3d are neural network functions that are currently implemented in our legacy THNN (CPU) / THCUNN (CUDA) libraries. It is generally better if these live in our new library ATen, since it is more feature complete and reduces cognitive overhead. Here is the documentation for one of these functions: https://pytorch.org/docs/master/nn.html?highlight=adaptive#torch.nn.MaxUnpool1d You can find how the function is binded to ATen in nn.yaml (https://github.com/pytorch/pytorch/blob/b30c803662d4c980588b087beebf98982b3b653c/aten/src/ATen/nn.yaml#L185-L195) and how its frontend API is defined in native_functions.yaml (https://github.com/pytorch/pytorch/blob/b30c803662d4c980588b087beebf98982b3b653c/aten/src/ATen/native/native_functions.yaml#L3334-L3356).

The goal here is to remove the entry in nn.yaml, so the implementation is done complete in ATen. There are a few reasonable steps you could take to do this (these could all be separate PRs):

- Implement a CPU-only replacement. You can still dispatch to THCUNN for the CUDA implementation by specifying backend-specific dispatch in native_functions.yaml (search for "dispatch:" for examples). The implementation of FractionalMaxPool3d (https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/native/FractionalMaxPool3d.cpp) can give you some inspiration here.
- Implement a CUDA-only replacement. As before, the CUDA implementation of FractionalMaxPool3d (https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/native/cuda/FractionalMaxPool3d.cu) can give you some inspiration.
- Implement a CPU-only replacement for the backwards functions (these are suffixed with _backward and _backward_out).
- Implement a CUDA-only replacement for the backwards functions

NOTE: no new tests should have to be written. If you want to test manually, run "python test/test_nn.py" and look for tests related to max unpooling, but the PR will run all the tests for you in any case.

Good PRs to refer to

A diff is worth a thousand words. Here are some existing ports which you can use to get some basic orientation:

- https://github.com/pytorch/pytorch/pull/14714/files

To read these diffs, you should pull up both the old and new code (they're in separate files) and compare them line by line.

Refactor philosophy

The golden rule of refactoring is *change one thing at a time*. If you change one thing, and the code breaks, you know that one thing is to blame. If you change many things, and the code breaks, you have no idea what you did wrong. **If you are doing a port, just do a port; don't try to "fix" things that you notice are wrong.** We would be ecstatic if you did a follow up PR to fix the problems you noticed, or submitted a preparatory PR that refactors the old code in place, before you actually do the port. Just don't do both at once!

Caveat: Small, low risk, improvements

This being said, there are some small changes which are unlikely to cause problems, which are nice to see when reviewing porting PRs. Here are some of them:

- Change C-style casts into C++ static_cast. Anywhere you see (scalar_t)x replace it with static_cast(x)
- Sink variable declarations to their first use site. A lot of TH declares variables as "int colWidth;" and then have a single def-site at some later point in time. Better to sink the declaration to the first point the variable is actually defined, to help avoid bugs where a variable is used before it is initialized.
- Change uses of "int" as booleans to actual "bool".

Dispatch and scalar_t

In code in TH, you will see references to scalar_t. In fact, all code in TH is implicitly "templated", and instantiated multiple times for various values of scalar_t; e.g., float, int, etc. This is done by a macro monstrosity that includes non-header files multiple times with different settings of scalar_t.

This is obviously horrible, and in ATen we do things more normally using templates. However, this doesn't mean you should go out and template everything. Instead, there is usually a critical loop inside the kernel which actually needs to be templated, and then everything else can be compiled once generically. At the point this occurs, you should use `AT_DISPATCH_FLOATING_TYPES_AND_HALF` to perform this dispatch from code which doesn't at compile time what scalar_t is, to code that knows what scalar_t. For usage guidance, search for some examples.

C++ torch API guidance

For the most part, the at::Tensor API matches the Python Torch tensor API directly; so you can check the official docs to find out what is callable: https://pytorch.org/docs/stable/tensors.html

There are a few notable differences, which are listed here:

- To get a vector of sizes of a tensor, write `x.sizes()` instead of `x.size()` (you can still get the size of an individual dimension with `x.size(d)`

Height-width argument order

Some kernels in THNN have very strange argument order; for example:

```
void THNN_(SpatialFractionalMaxPooling_updateOutput)(
           THCState *state,
           THCTensor *input,
           THCTensor *output,
           int outputW, int outputH,
           int poolSizeW, int poolSizeH,
           THCIndexTensor *indices,
           THCTensor *randomSamples)
```

outputW comes before outputH, even though conventional NCHW ordering says that height comes before width. **There is code generation logic specifically for THNN that reorders output_sizes to match the THNN kernel; you should manually reorder when you port.** So for example, the above arguments become:

```
void fractional_max_pool2d_out(
  Tensor& output,
  Tensor& indices,
  const Tensor& input,
  IntList pool_size,
  IntList output_size,
  const Tensor& randomSamples) {
  ...
  auto outputH = output[0]; // Notice they are reordered
  auto outputW = output[1];
  auto poolSizeH = pool_size[0];
  auto poolSizeW = pool_size[1];
}
```

The tests are supposed to catch if you've gotten it wrong, but we may have missed some cases, so be vigilant. In general, height comes before width. Here is a full list of dimension offsets (from aten/src/ATen/nn_parse.py), where negative numbers mean that you index from the end of the list:

```
DIMENSION_OFFSET = {
    'width': -1,
    'height': -2,
    'B': 0,
    'C': 1,
    'W': -1,
    'H': -2,
    'T': -3,
    'left': 0,
    'right': 1,
    'top': 2,
    'bottom': 3,
    'front': 4,
```

```
    'back': 5,
}
```

TH API guidance

General guidance: maybe someone has ported something similar before! You can use "pickaxe" to search for diffs that mention the function you are interested in, e.g., "git log -S THCTensor_canUse32BitIndexMath" to see if anyone edited a line mentioning this, and then see if it was a porting commit or not.

- `THCUNN_assertSameGPU` : use `checkAllSameGPU`
- `THTensor_(data)(input)` and `THCTensor_(data)(state, input)` : use `input.data<T>()` (where T is the type of the pointer you want)
- `THCTensor_canUse32BitIndexMath(state, input)` : use at::cuda::detail::canUse32BitIndexMath
- `THCTensor_(nDimensionLegacyNoScalars)(state, input)`
  - This one is a bit interesting. See [NOTE: nDimension vs nDimensionLegacyNoScalars vs nDimensionLegacyAll] in **aten/src/TH/THTensor.hpp** for the gory details
  - **Usually**, it's the case that the dimension of the tensor is guaranteed not to be zero. In this case, you can replace this with a regular input.dim() call. However, if you think that the input might be scalar, some more involved changes are necessary. Get in touch with your Bootcamp mentor if you think this is the case.
- `toDeviceTensor<scalar_t, 2>(state, input):` use PackedTensorAccessor. Example diff 14004cbef67c8cb933938bd61b75fd46857bd903
  - Unfortunately, PackedTensorAccessor has less features than DeviceTensor. A very common idiom is to call upcastOuter<5>() after creating a DeviceTensor. This can be replaced by calling view() on the tensor to change its shape before calling PackedTensorAccessor.
- `x->is_empty()` ==> `x.numel() == 0`
- `THNN_ARGCHECK(condition, argument position, error message)` : use `AT_CHECK(condition, message)`
- `TH_INDEX_BASE=0.`
- `THCNumerics<Dtype>::min()` ==> `at::numeric_limits<scalar_t>::lowest()`
- The `Acctype` template type can be translate from `scalar_t` by `at::acc_type<scalar_t, /*is_cuda=*/[true, false]>.`
- `THCCeilDiv(m, n)` ==> `(m + n - 1) / n # ceiling division.`

Caveats for `native_functions.yaml`

- The argument order in native_functions.yaml does not match the order in the generated file `torch/include/ATen/NativeFunctions.h` . Example:

  a) Signature: `adaptive_max_pool2d(Tensor self, int[2] output_size, *, Tensor(a!) out, Tensor(b!) indices) -> (Tensor(a!), Tensor(b!))` b) Function prototype: `std::tuple<Tensor &,Tensor &> adaptive_max_pool2d_out_cpu(Tensor & out, Tensor & indices, const Tensor & self, IntArrayRef output_size);`

- Argument names matter, the convention is to use `out` for output arguments. See: https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/native/README.md

What's the deal buffers and forward/non-forward functions?

The way forward and non-forward in THNN bindings works is that some THNN functions take extra "buffers", which are scratch space used by the kernels and then reused by the backwards. `_thnn_conv_transpose2d` is an example of a function that has buffers: if you look at its nn.yaml declaration it says:

```
- name: _thnn_conv_transpose2d(Tensor self, Tensor weight, IntArrayRef[2] kernel_size, Tensor? bias={},
IntArrayRef[2] stride=1, IntArrayRef[2] padding=0, IntArrayRef[2] output_padding=0, IntArrayRef[2]
dilation=1)
  cname: SpatialFullDilatedConvolution
  buffers: [columns, ones]
```

That is the "publicly visible" function signature. If you, however, compare the C++ signatures of the forward and non-forward variants (I picked these out of `TypeExtendedInterface.h` , a generated file), you'll see:

```
  virtual std::tuple<Tensor,Tensor,Tensor> _thnn_conv_transpose2d_forward(const Tensor & self, const
Tensor & weight, IntArrayRef kernel_size, const Tensor & bias, IntArrayRef stride, IntArrayRef padding,
IntArrayRef output_padding, IntArrayRef dilation) const = 0;
  virtual Tensor thnn_conv_transpose2d(const Tensor & self, const Tensor & weight, IntArrayRef
kernel_size, const Tensor & bias, IntArrayRef stride, IntArrayRef padding, IntArrayRef output_padding,
IntArrayRef dilation) const = 0;
```

The forward version returns three Tensors; the first is the normal result, but the latter two are the columns and ones buffers. `thnn_conv_transpose2d` wraps `_forward` so that only the result is returned. In derivatives.yaml, we only define a derivative on the `_forward` variant (and in fact, we must, because the backwards formula requires the columns/ones buffer).

If we treat the kernel as a black box, the discussion stops here. But what do ones and columns mean, really? It turns out, we don't *really* rely on the previous data in an interesting way, these buffers are just scratch space used by the kernels. Here is the initialization of `columns` in `SpatialDilatedConvolution_accGradParameters` which dominates all uses of `columns`:

```
      // Extract columns:
      THNN_(im2col)(
        input_n->data<scalar_t>(),
        nInputPlane, inputHeight, inputWidth,
        outputHeight, outputWidth,
        kH, kW, padH, padW, dH, dW,
        dilationH, dilationW,
        columns->data<scalar_t>()
      );
```

And here is the (possibly dominating) initialization of `ones`:

```
      // Do GEMV (note: this is a bit confusing because gemv assumes column-major matrices)
      // Define a buffer of ones, for bias accumulation
      if (ones->dim() != 2 || ones->size(0)*ones->size(1) < outputHeight*outputWidth) {
        // Resize plane and fill with ones...
        THTensor_(resize2d)(ones, outputHeight, outputWidth);
        THTensor_(fill)(ones, 1);
      }
```

The passing around of columns and ones might seem a bit strange. Why go through all that trouble, just to share these buffers? To answer this question, you need to know a little bit of history about LuaTorch, from whence these kernels come. In LuaTorch, we didn't have a CUDA caching allocator; memory allocations were very expensive. To amortize the cost of allocating temporary buffers, LuaTorch would frequently do an allocation once in forwards, and keep that memory around to reuse in backwards.

This consideration is no longer an issue in PyTorch, since we have a caching allocator, which means that allocating memory on the fly is cheap. So in an ideal world, we wouldn't have these columns/ones buffers at all, in which case you also don't need the forward or non-forward variants either.

In any case, if this is making your head hurt, I recommend doing a slavish first, and then refactoring the code to remove the buffers afterwards.