

## [ezyang's blog](#)

the arc of software bends towards understanding

- [About](#)
- [Archives](#)
- [Subscribe](#)

### PyTorch internals

This post is a long form essay version of a talk about PyTorch internals, that I gave at the PyTorch NYC meetup on May 14, 2019.

# PyTorch Internals

Edward Z. Yang

Hi everyone! Today I want to talk about the internals of [PyTorch](#).

# Who's this talk for?

You want to contribute to PyTorch,  
but the codebase seems daunting

Notable omissions: JIT, distributed

This talk is for those of you who have used PyTorch, and thought to yourself, "It would be great if I could contribute to PyTorch," but were scared by PyTorch's behemoth of a C++ codebase. I'm not going to lie: the PyTorch codebase can be a bit overwhelming at times. The purpose of this talk is to put a map in your hands: to tell you about the basic conceptual structure of a "tensor library that supports automatic differentiation", and give you some tools and tricks for finding your way around the codebase. I'm going to assume that you've written some PyTorch before, but haven't necessarily delved deeper into how a machine learning library is written.

# Concepts

Tensor/Storage/Strides  
Layout/Device/Dtype  
Autograd

# Mechanics

Operator call stack  
Tools for writing kernels  
Legacy code  
Efficient workflow

The talk is in two parts: in the first part, I'm going to first introduce you to the conceptual universe of a tensor library. I'll start by talking about the tensor data type you know and love, and give a more detailed discussion about what exactly this data type provides, which will lead us to a better understanding of how it is actually implemented under the hood. If you're an advanced user of PyTorch, you'll be familiar with most of this material. We'll also talk about the trinity of "extension points", layout, device and dtype, which guide how we think about extensions to the tensor class. In the live talk at PyTorch NYC, I skipped the slides about autograd, but I'll talk a little bit about them in these notes as well.

The second part grapples with the actual nitty gritty details involved with actually coding in PyTorch. I'll tell you how to cut your way through swaths of autograd code, what code actually matters and what is legacy, and also all of the cool tools that PyTorch gives you for writing kernels.

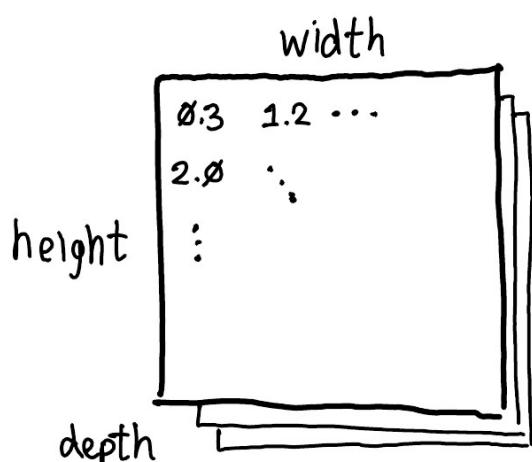
---

# Concepts

# Tensor

The tensor is the central data structure in PyTorch. You probably have a pretty good idea about what a tensor intuitively represents: it's an n-dimensional data structure containing some sort of scalar type, e.g., floats, ints, et cetera. We can think of a tensor as consisting of some data, and then some metadata describing the size of the tensor, the type of the elements it contains (dtype), what device the tensor lives on (CPU memory? CUDA memory?)

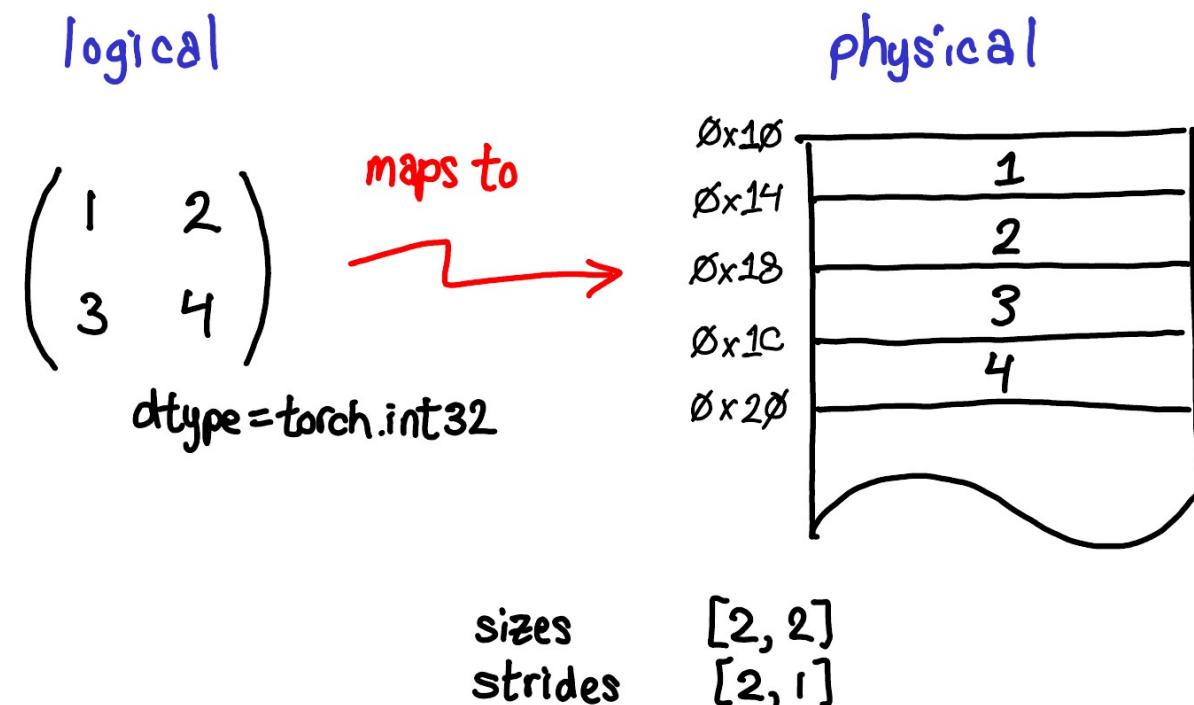
# Tensor



sizes	$(D, H, W)$	contiguous
strides	$(H*W, W, 1)$	←
dtype	float	
device	cuda:0	
layout	strided	

There's also a little piece of metadata you might be less familiar with: the stride. Strides are actually one of the distinctive features of PyTorch, so it's worth discussing them a little more.

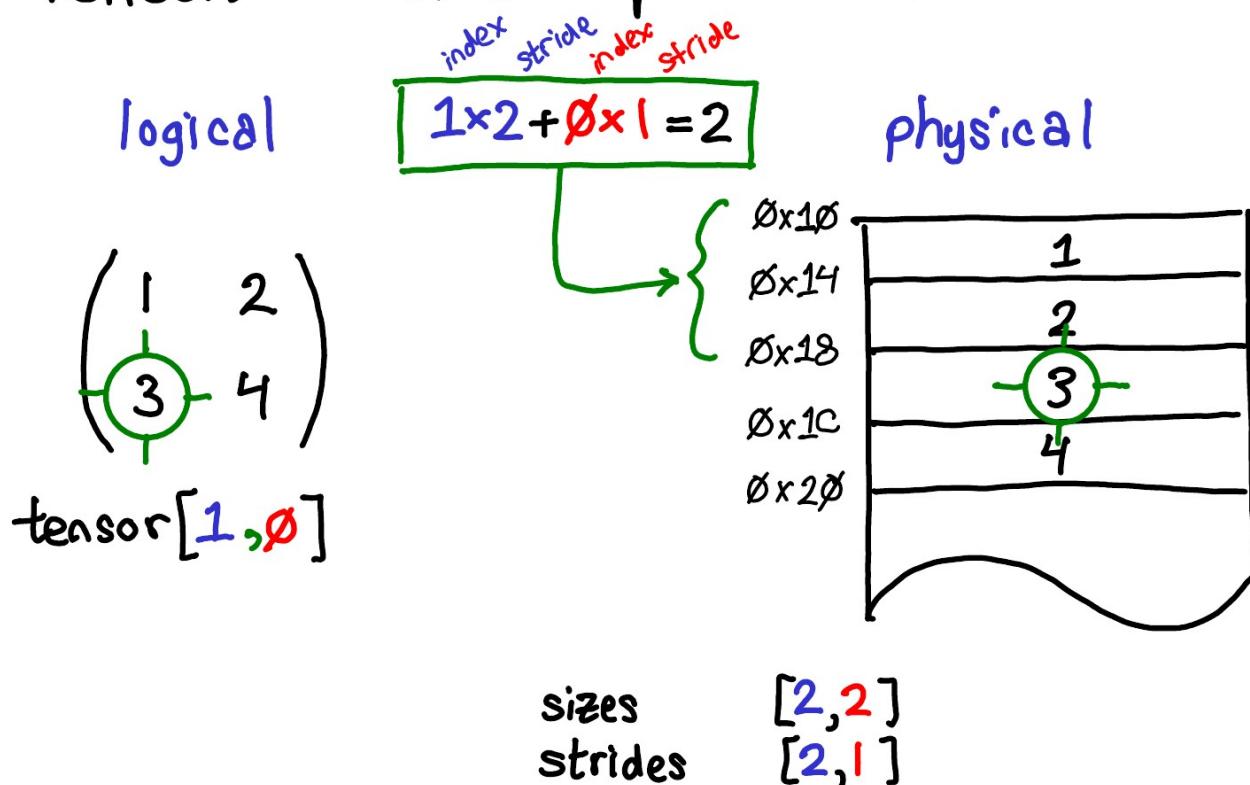
# Tensor: Strided Representation



A tensor is a mathematical concept. But to represent it on our computers, we have to define some sort of physical representation for them. The most common representation is to lay out each element of the tensor contiguously in memory (that's where the term contiguous comes from), writing out each row to memory, as you see above. In the example above, I've specified that the tensor contains 32-bit integers, so you can see that each integer lies in a physical address, each offset four bytes from each other. To remember what the actual dimensions of the tensor are, we have to also record what the sizes are as extra metadata.

So, what do strides have to do with this picture?

# Tensor: Strided Representation



Upcoming: TensorAccessor

rev2

Suppose that I want to access the element at position `tensor[0, 1]` in my logical representation. How do I translate this logical position into a location in physical memory? Strides tell me how to do this: to find out where any element for a tensor lives, I multiply each index with the respective stride for that dimension, and sum them all together. In the picture above, I've color coded the first dimension blue and the second dimension red, so you can follow the index and stride in the stride calculation. Doing this sum, I get two (zero-indexed), and indeed, the number three lives two below the beginning of the contiguous array.

(Later in the talk, I'll talk about `TensorAccessor`, a convenience class that handles the indexing calculation. When you use `TensorAccessor`, rather than raw pointers, this calculation is handled under the covers for you.)

Strides are the fundamental basis of how we provide views to PyTorch users. For example, suppose that I want to extract out a tensor that represents the second row of the tensor above:

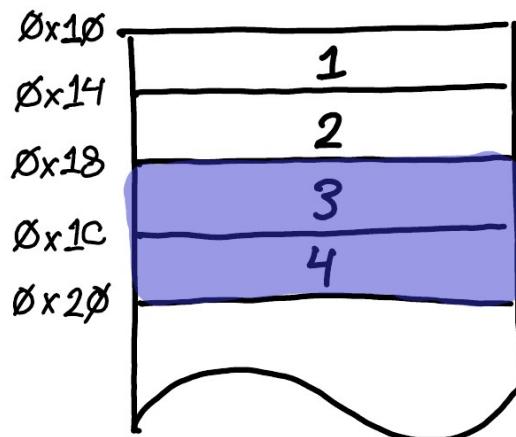
# Tensor: Strided Representation

logical

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

`tensor[1, :]`

physical



sizes  
strides  
offset

[ 2 ]  
[ 1 ]  
2

Using advanced indexing support, I can just write `tensor[1, :]` to get this row. Here's the important thing: when I do this, I don't create a new tensor; instead, I just return a tensor which is a different view on the underlying data. This means that if I, for example, edit the data in that view, it will be reflected in the original tensor. In this case, it's not too hard to see how to do this: three and four live in contiguous memory, and all we need to do is record an offset saying that the data of this (logical) tensor lives two down from the top. (Every tensor records an offset, but most of the time it's zero, and I'll omit it from my diagrams when that's the case.)

Question from the talk: If I take a view on a tensor, how do I free the memory of the underlying tensor?

Answer: You have to make a copy of the view, thus disconnecting it from the original physical memory. There's really not much else you can do. By the way, if you have written Java in the old days, taking substrings of strings has a similar problem, because by default no copy is made, so the substring retains the (possibly very large string). Apparently, they [fixed this in Java 7u6](#).

A more interesting case is if I want to take the first column:

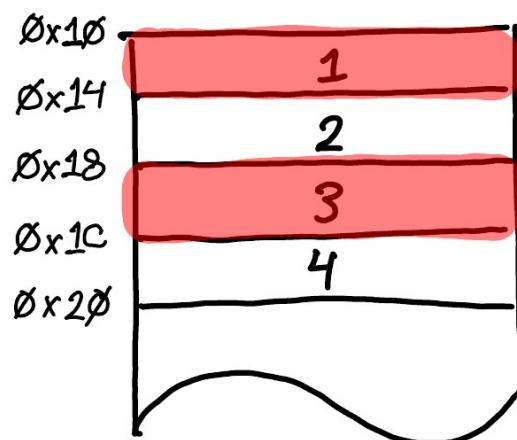
# Tensor: Strided Representation

logical

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

`tensor[:, :, 0]`

physical



sizes  
strides

[2]  
[2]

stride ≠ 1 means we skip elements

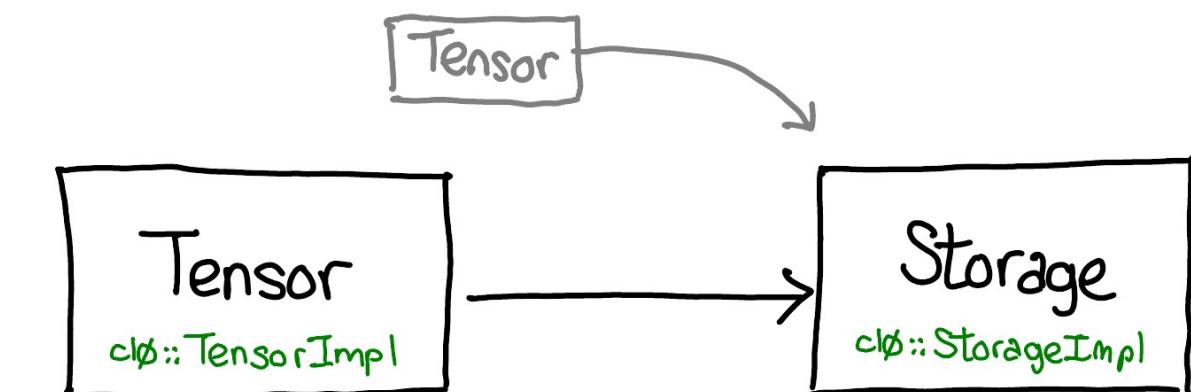
When we look at the physical memory, we see that the elements of the column are not contiguous: there's a gap of one element between each one. Here, strides come to the rescue: instead of specifying a stride of one, we specify a stride of two, saying that between one element and the next, you need to jump two slots. (By the way, this is why it's called a "stride": if we think of an index as walking across the layout, the stride says how many locations we stride forward every time we take a step.)

The stride representation can actually let you represent all sorts of interesting views on tensors; if you want to play around with the possibilities, check out the [Stride Visualizer](#).

Let's step back for a moment, and think about how we would actually implement this functionality (after all, this is an internals talk.) If we can have views on tensor, this means we have to decouple the notion of the tensor (the user-visible concept that you know and love), and the actual physical data that stores the data of the tensor (called storage):

<https://ezyang.github.io/stride-visualizer>

# Tensor: Strided Representation



the logical view  
sizes, strides, offset

the actual physical data  
physical size, dtype

there is always a `Tensor+Storage`, even for "simple" cases

There may be multiple tensors which share the same storage. Storage defines the dtype and physical size of the tensor, while each tensor records the sizes, strides and offset, defining the logical interpretation of the physical memory.

One thing to realize is that there is always a pair of Tensor-Storage, even for "simple" cases where you don't really need a storage (e.g., you just allocated a contiguous tensor with `torch.zeros(2, 2)`).

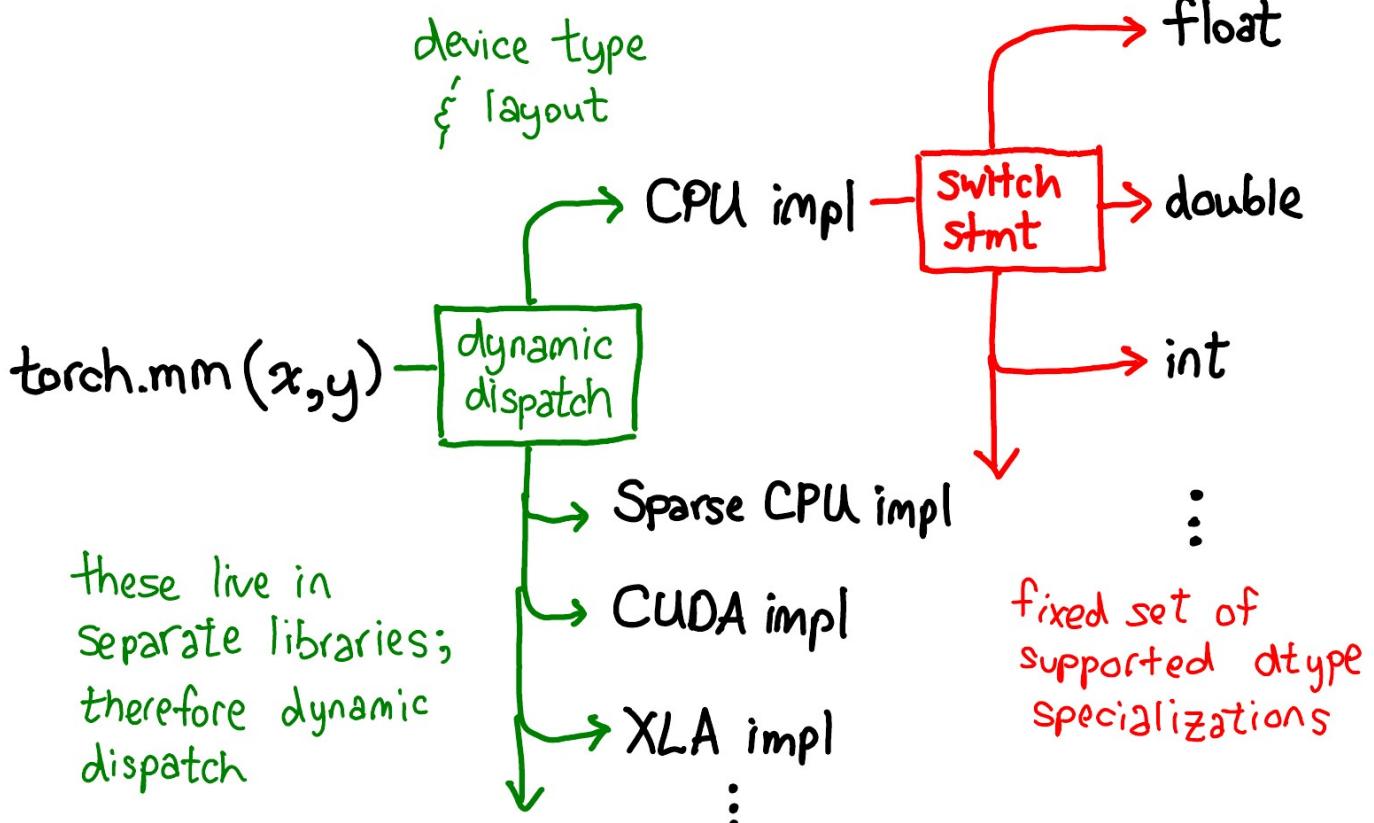
By the way, we're interested in making this picture not true; instead of having a separate concept of storage, just define a view to be a tensor that is backed by a base tensor. This is a little more complicated, but it has the benefit that contiguous tensors get a much more direct representation without the Storage indirection. A change like this would make PyTorch's internal representation a bit more like Numpy's.

---

We've talked quite a bit about the data layout of tensor (some might say, if you get the data representation right, everything else falls in place). But it's also worth briefly talking about how operations on the tensor are implemented. At the very most abstract level, when you call `torch.mm`, two dispatches happen:

# Tensor operations

(slight simplification)  
dtype



The first dispatch is based on the device type and layout of a tensor: e.g., whether or not it is a CPU tensor or a CUDA tensor (and also, e.g., whether or not it is a strided tensor or a sparse one). This is a dynamic dispatch: it's a virtual function call (exactly where that virtual function call occurs will be the subject of the second half of this talk). It should make sense that you need to do a dispatch here: the implementation of CPU matrix multiply is quite different from a CUDA implementation. It is a *dynamic* dispatch because these kernels may live in separate libraries (e.g., `libcaffee2.so` versus `libcaffee2_gpu.so`), and so you have no choice: if you want to get into a library that you don't have a direct dependency on, you have to dynamic dispatch your way there.

The second dispatch is a dispatch on the dtype in question. This dispatch is just a simple switch-statement for whatever dtypes a kernel chooses to support. Upon reflection, it should also make sense that we need to a dispatch here: the CPU code (or CUDA code, as it may) that implements multiplication on `float` is different from the code for `int`. It stands to reason you need separate kernels for each dtype.

This is probably the most important mental picture to have in your head, if you're trying to understand the way operators in PyTorch are invoked. We'll return to this picture when it's time to look more at code.

# Tensor extensions

sparse tensors

quantized tensors

encrypted tensors

There's more to life than strided tensors

MKLDNN tensors

TPU tensors

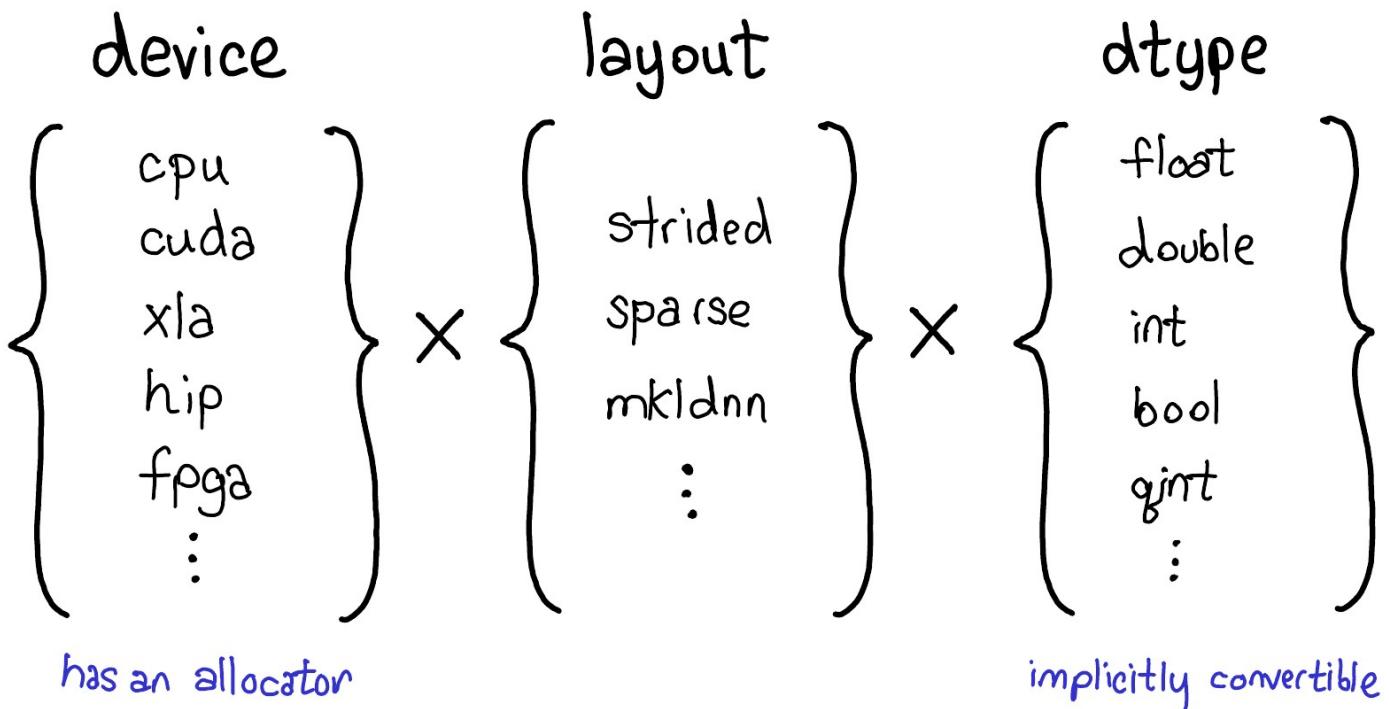
batch tensors

Since we have been talking about Tensor, I also want to take a little time to the world of tensor extensions. After all, there's more to life than dense, CPU float tensors. There's all sorts of interesting extensions going on, like XLA tensors, or quantized tensors, or MKL-DNN tensors, and one of the things we have to think about, as a tensor library, is how to accommodate these extensions.

# tensor wrapper

e.g., batch tensors

(things that  
don't flow  
through autograd)



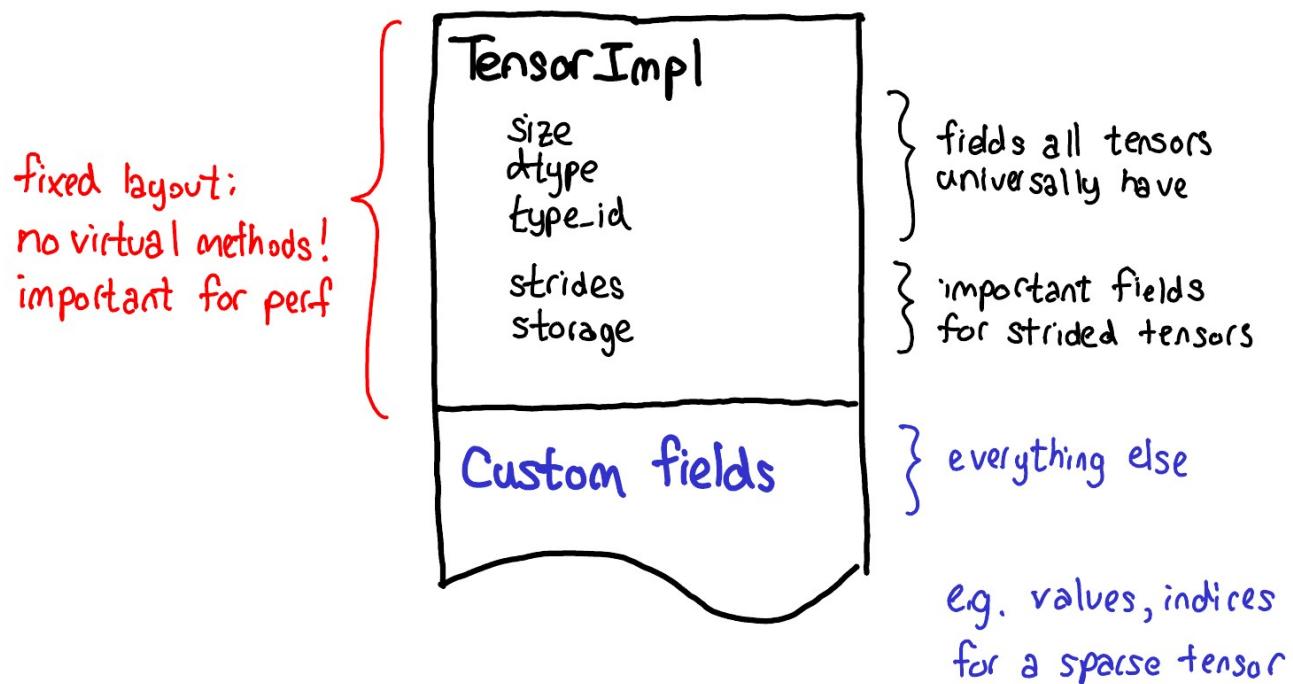
Our current model for extensions offers four extension points on tensors. First, there is the trinity three parameters which uniquely determine what a tensor is:

- The **device**, the description of where the tensor's physical memory is actually stored, e.g., on a CPU, on an NVIDIA GPU (cuda), or perhaps on an AMD GPU (hip) or a TPU (xla). The distinguishing characteristic of a device is that it has its own allocator, that doesn't work with any other device.
- The **layout**, which describes how we logically interpret this physical memory. The most common layout is a strided tensor, but sparse tensors have a different layout involving a pair of tensors, one for indices, and one for data; MKL-DNN tensors may have even more exotic layout, like blocked layout, which can't be represented using merely strides.
- The **dtype**, which describes what it is that is actually stored in each element of the tensor. This could be floats or integers, or it could be, for example, quantized integers.

If you want to add an extension to PyTorch tensors (by the way, if that's what you want to do, please talk to us! None of these things can be done out-of-tree at the moment), you should think about which of these parameters you would extend. The Cartesian product of these parameters define all of the possible tensors you can make. Now, not all of these combinations may actually have kernels (who's got kernels for sparse, quantized tensors on FPGA?) but in principle the combination could make sense, and thus we support expressing it, at the very least.

There's one last way you can make an "extension" to Tensor functionality, and that's write a wrapper class around PyTorch tensors that implements your object type. This perhaps sounds obvious, but sometimes people reach for extending one of the three parameters when they should have just made a wrapper class instead. One notable merit of wrapper classes is they can be developed entirely out of tree.

When should you write a tensor wrapper, versus extending PyTorch itself? The key test is whether or not you need to pass this tensor along during the autograd backwards pass. This test, for example, tells us that sparse tensor should be a true tensor extension, and not just a Python object that contains an indices and values tensor: when doing optimization on networks involving embeddings, we want the gradient generated by the embedding to be sparse.



Our philosophy on extensions also has an impact of the data layout of tensor itself. One thing we really want out of our tensor struct is for it to have a fixed layout: we don't want fundamental (and very frequently called) operations like "What's the size of a tensor?" to require virtual dispatches. So when you look at the actual layout of a Tensor (defined in the [TensorImpl struct](#)), what we see is a common prefix of all fields that we consider all "tensor"-like things to universally have, plus a few fields that are only really applicable for strided tensors, but are *so* important that we've kept them in the main struct, and then a suffix of custom fields that can be done on a per-Tensor basis. Sparse tensors, for example, store their indices and values in this suffix.

# Autograd

I told you all about tensors, but if that was the only thing PyTorch provided, we'd basically just be a Numpy clone. The distinguishing characteristic of PyTorch when it was originally released was that it provided automatic differentiation on tensors (these days, we have other cool features like TorchScript; but back then, this was it!)

What does automatic differentiation do? It's the machinery that's responsible for taking a neural network:

```
i2h = torch.mm(W-x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
loss = next_h.sum()
loss.backward()
```

W\_h, W-x, x, prev-h  
requires grad

...and fill in the missing code that actually computes the gradients of your network:

$i2h = \text{torch.mm}(W\_x, x.t())$ 
 $h2h = \text{torch.mm}(W\_h, \text{prev\_h}.t())$ 
 $\text{next\_h} = i2h + h2h$ 
 $\text{next\_h2} = \text{next\_h.tanh}()$ 
 $\text{loss} = \text{next\_h2.sum}()$ 

$W\_h$ ,  $W\_x$ ,  $x$ , prev-h  
requires grad

swap inputs and outputs!

 $\text{grad\_loss} = \text{torch.tensor}(1, \text{dtype=loss.dtype})$ 
 $\text{grad\_next\_h2} += \text{grad\_loss.expand}(\text{next\_h2.size}())$ 

original inputs  
may be reused

 $\text{grad\_next\_h} += \text{tanh\_backward}(\text{grad\_next\_h2}, \text{next\_h2})$ 
 $\text{grad\_i2h}, \text{grad\_h2h} += \text{grad\_next\_h}, \text{grad\_next\_h}$ 
 $W\_h.\text{grad} += \text{mm\_mat1\_backward}(\text{grad\_h2h}, \text{prev\_h.t}(), W\_h, 1)$ 
 $W\_x.\text{grad} += \text{mm\_mat1\_backward}(\text{grad\_i2h}, x.t(), W\_x, 1)$ 

how do we get here?

All grad variables are zero to start:

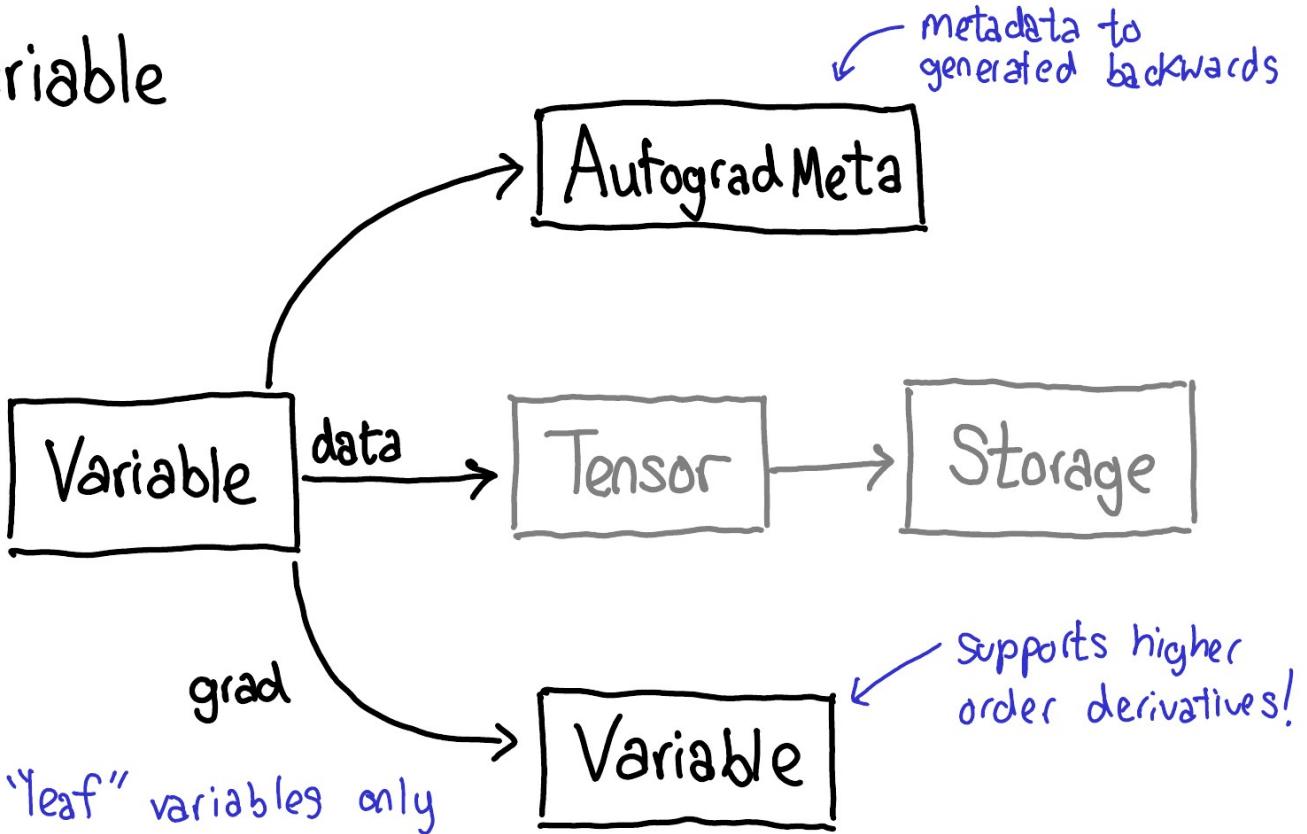
 $\text{grad\_i2h} = \text{torch.zeros\_like}(i2h)$ 

Take a moment to study this diagram. There's a lot to unpack; here's what to look at:

1. First, rest your eyes on the variables in red and blue. PyTorch implements [reverse-mode automatic differentiation](#), which means that we effectively walk the forward computations "backward" to compute the gradients. You can see this if you look at the variable names: at the bottom of the red, we compute loss; then, the first thing we do in the blue part of the program is compute grad\_loss. loss was computed from next\_h2, so we compute grad\_next\_h2. Technically, these variables which we call grad\_ are not really gradients; they're really Jacobians left-multiplied by a vector, but in PyTorch we just call them grad and mostly everyone knows what we mean.
2. If the structure of the code stays the same, the behavior doesn't: each line from forwards is replaced with a different computation, that represents the derivative of the forward operation. For example, the tanh operation is translated into a tanh\_backward operation (these two lines are connected via a grey line on the left hand side of the diagram). The inputs and outputs of the forward and backward operations are swapped: if the forward operation produced next\_h2, the backward operation takes grad\_next\_h2 as an input.

The whole point of autograd is to do the computation that is described by this diagram, but without actually ever generating this source. PyTorch autograd doesn't do a source-to-source transformation (though PyTorch JIT does know how to do symbolic differentiation).

# Variable



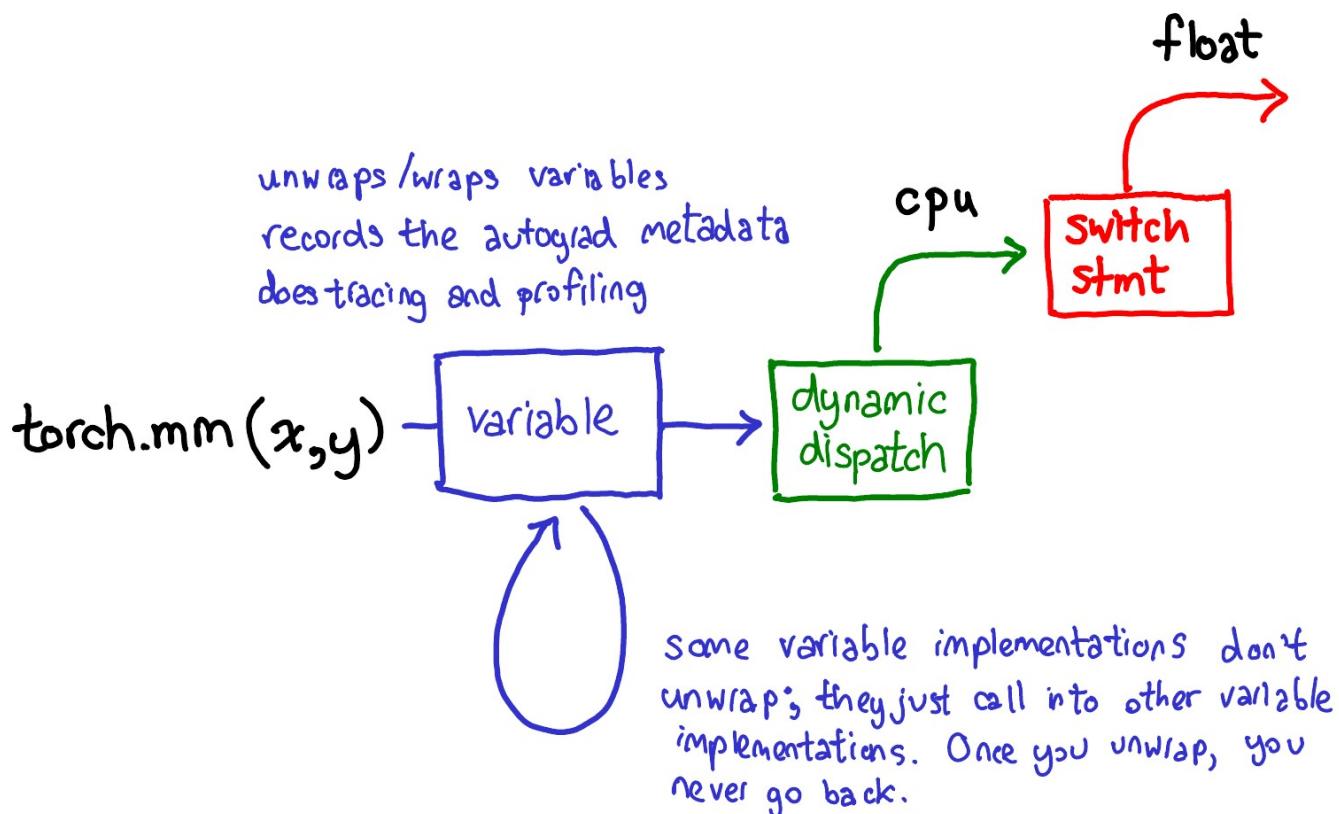
NB: This changing soon!

To do this, we need to store more metadata when we carry out operations on tensors. Let's adjust our picture of the tensor data structure: now instead of just a tensor which points to a storage, we now have a variable which wraps this tensor, and also stores more information (AutogradMeta), which is needed for performing autograd when a user calls `loss.backward()` in their PyTorch script.

This is yet another slide which will hopefully be out of date in the near future. Will Feng is working on a [Variable-Tensor merge in C++](#), following a simple merge which happened to PyTorch's frontend interface.

We also have to update our picture about dispatch:

# Dispatch with autograd



Before we dispatch to CPU or CUDA implementations, there is another dispatch on variables, which is responsible for unwrapping variables, calling the underlying implementation (in green), and then rewrapping the results into variables and recording the necessary autograd metadata for backwards.

Some implementations don't unwrap; they just call into other variable implementations. So you might spend a while in the Variable universe. However, once you unwrap and go into the non-Variable Tensor universe, that's it; you never go back to Variable (except by returning from your function.)

---

In my NY meetup talk, I skipped the following seven slides. I'm also going to delay writeup for them; you'll have to wait for the sequel for some text.

# AutogradMeta

```
Variable grad_;  
shared_ptr<Function> grad_fn_;  
weak_ptr<Function> grad_accumulator_;  
bool requires_grad_ // leaf only  
uint32_t output_nr_;
```

## Function

```
edge_list next_edges_;
```

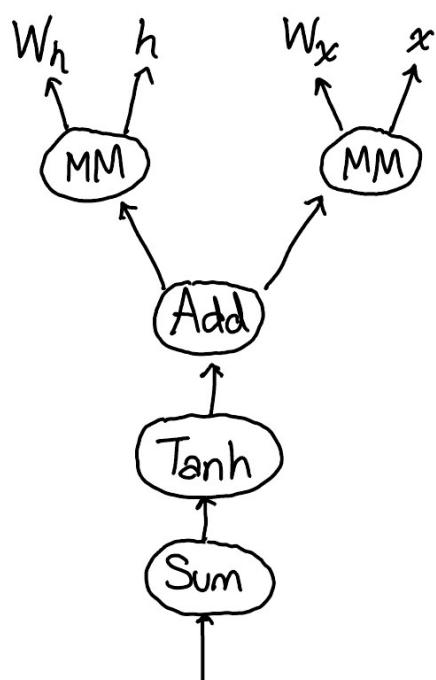
(e.g.,) *SavedVariable x*;

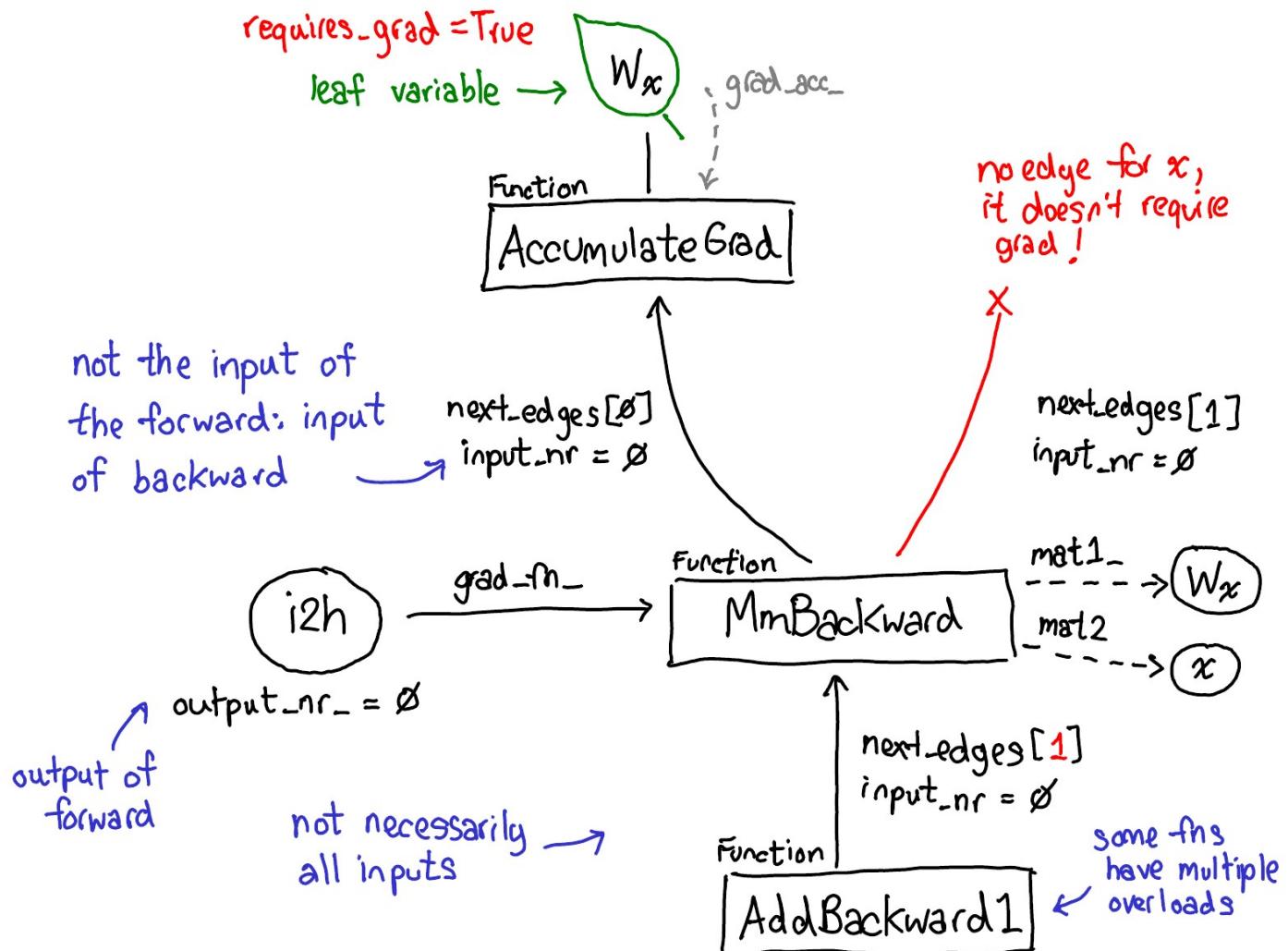
class hierarchy

## Edge

```
shared_ptr<Function> function;  
uint32_t input_nr_;
```

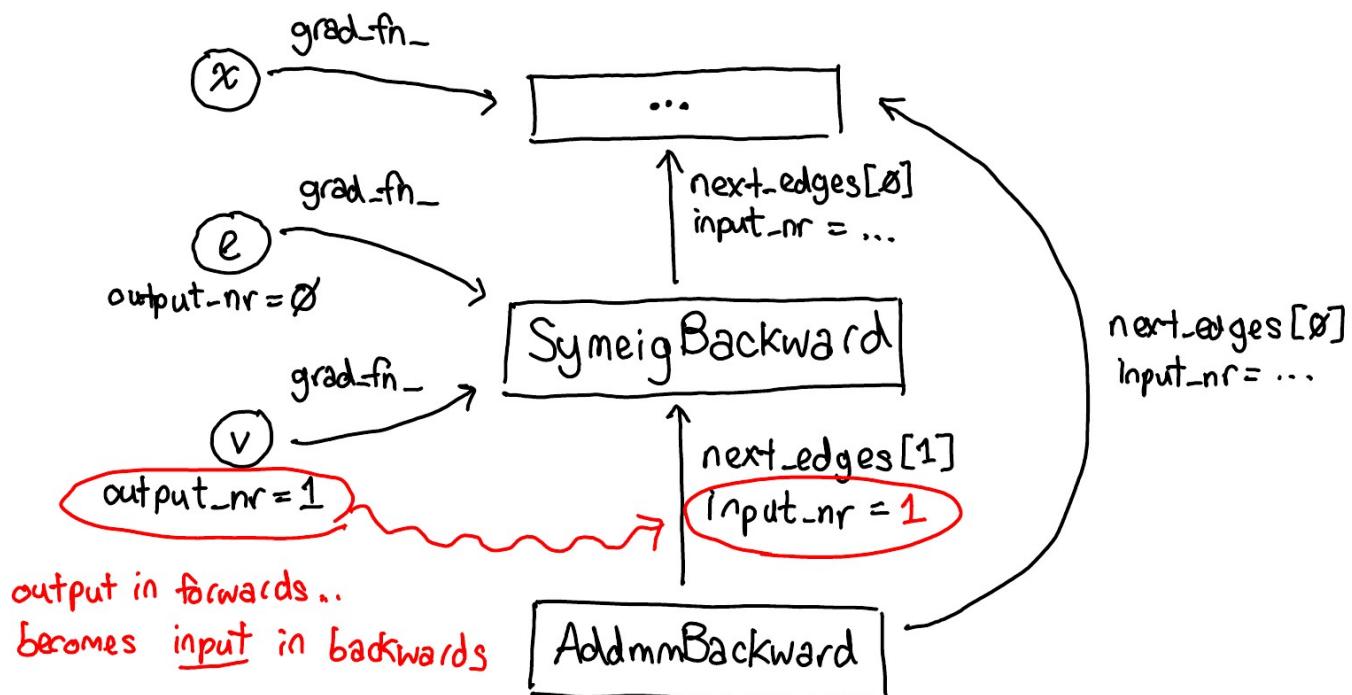
```
i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
loss = next_h.sum()
loss.backward()
```





# Multiple outputs

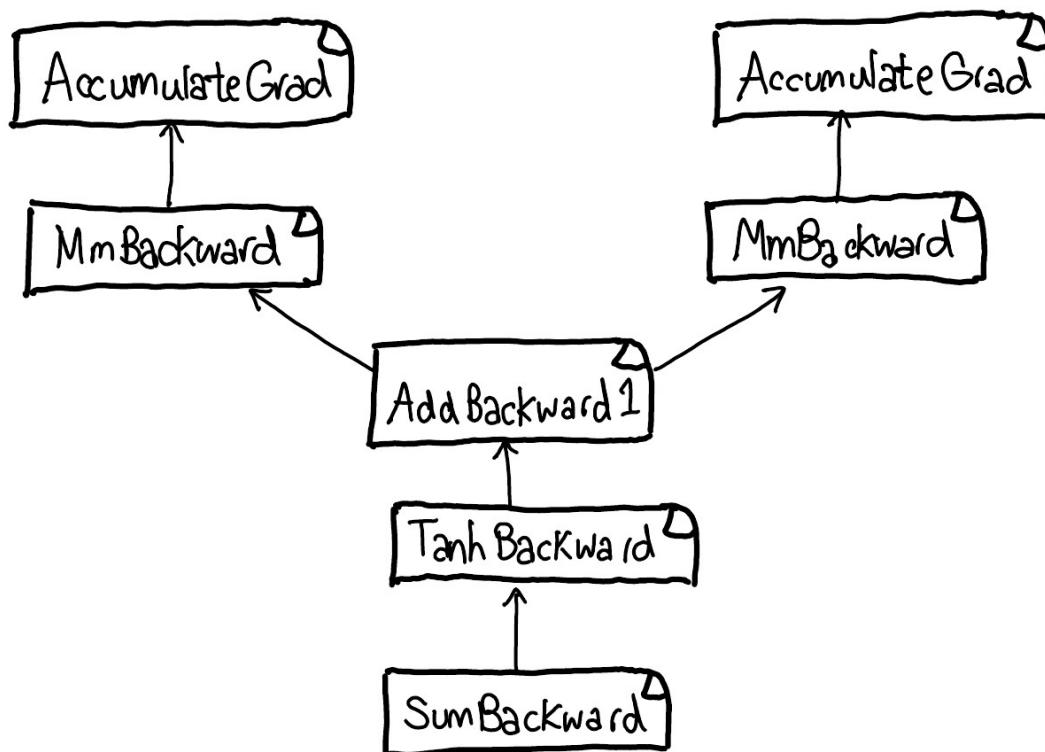
$e, v = \text{torch.symeig}(x, \text{eigenvectors}=\text{True})$   
 $d = \text{torch.mm}(x, v)$



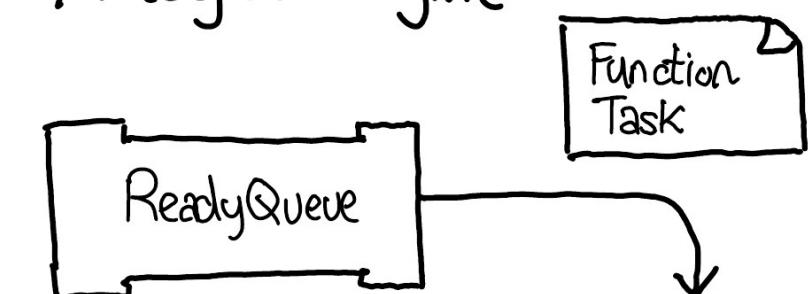
# Autograd engine

"just a parallel graph executor"

FunctionTasks:



# Autograd engine

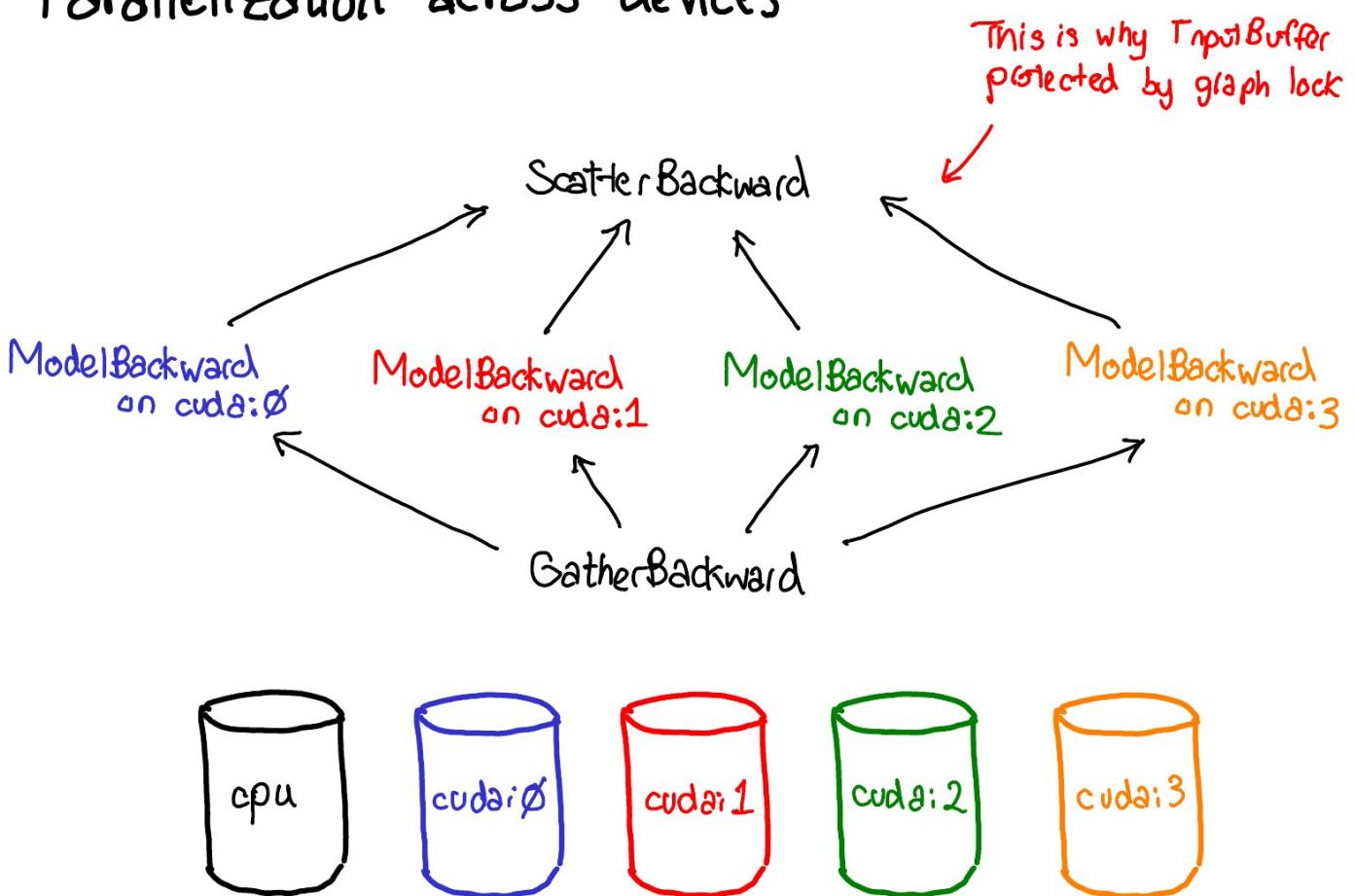


There's a preprocessing scan of the graph to determine how many dependencies each FunctionTask has

1. Evaluate function on InputBuffer
  - a. Call pre-hooks on inputs
  - b. Call the function for real
  - c. Call post-hooks on outputs
2. Release saved variables
3. With graph lock, accumulate grad into InputBuffers of next functions, and add newly ready FunctionTasks to Ready Queue



# Parallelization across devices



# Mechanics

Enough about concepts, let's look at some code.

# Finding your way around

PyTorch has a lot of folders, and there is a very detailed description of what they are in the [CONTRIBUTING](#) document, but really, there are only four directories you really need to know about:

# Source structure

`torch/`

Python; the frontend!

`torch/csrc/`

Python bindings, this functionality

`aten/src/ATen/`

Tensor operator implementations

`c10/`

Core abstractions (e.g. `TensorImpl`)



`api/` C++ API

`autograd/`

`jit/` TorchScript

`native/` Modern

../`TH, THC, ...`

Legacy

- First, `torch/` contains what you are most familiar with: the actual Python modules that you import and use. This stuff is Python code and easy to hack on (just make a change and see what happens). However, lurking not too deep below the surface is...
- `torch/csrc/`, the C++ code that implements what you might call the frontend of PyTorch. In more descriptive terms, it implements the binding code that translates between the Python and C++ universe, and also some pretty important pieces of PyTorch, like the autograd engine and the JIT compiler. It also contains the C++ frontend code.
- `aten/`, short for "A Tensor Library" (coined by Zachary DeVito), is a C++ library that implements the operations of Tensors. If you're looking for where some kernel code lives, chances are it's in ATen. ATen itself bifurcates into two neighborhoods of operators: the "native" operators, which are modern, C++ implementations of operators, and the "legacy" operators (TH, THC, THNN, THCUNN), which are legacy, C implementations. The legacy operators are the bad part of town; try not to spend too much time there if you can.
- `c10/`, which is a pun on Caffe2 and A"Ten" (get it? Caffe 10) contains the core abstractions of PyTorch, including the actual implementations of the Tensor and Storage data structures.

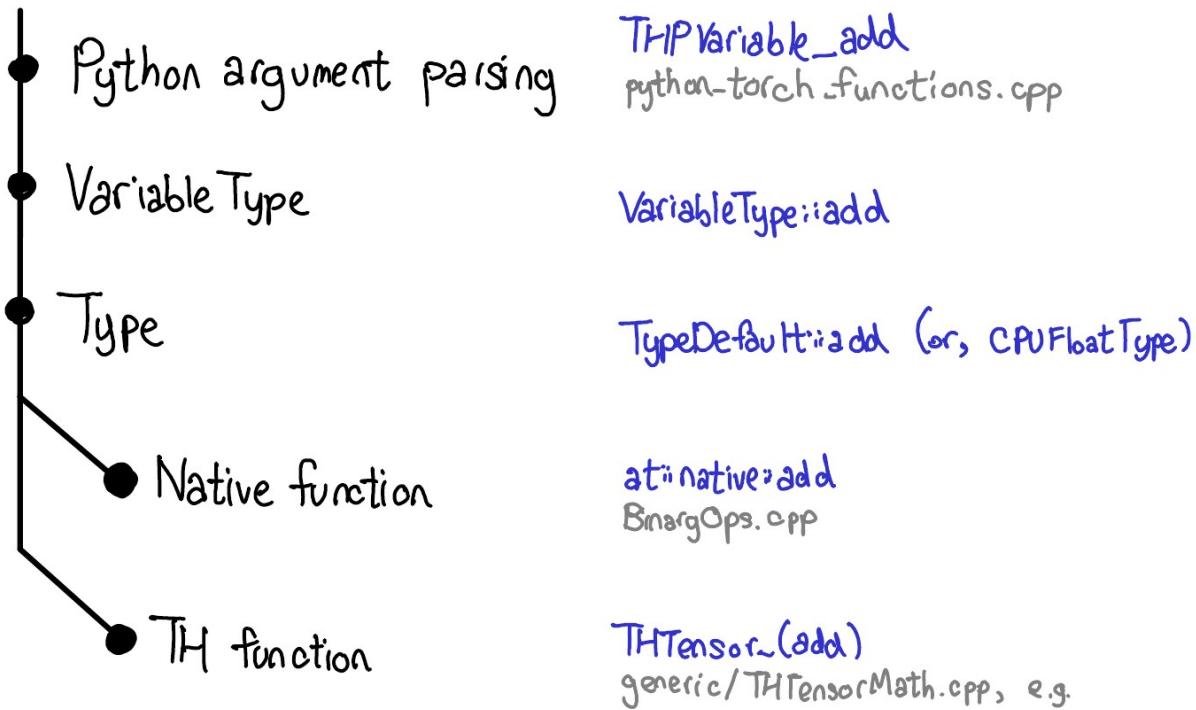
That's a lot of places to look for code; we should probably simplify the directory structure, but that's how it is. If you're trying to work on operators, you'll spend most of your time in `aten`.

Let's see how this separation of code breaks down in practice:

# Anatomy of an operator call

How to figure this out?  
 Build PyTorch with DEBUG=1,  
 set a breakpoint on  
 at::native::add, and look  
 at the backtrace!

`torch.add(x, y)`



rev 2

When you call a function like `torch.add`, what actually happens? If you remember the discussion we had about dispatching, you already have the basic picture in your head:

1. We have to translate from Python realm to the C++ realm (Python argument parsing)
2. We handle **variable** dispatch (VariableType->Type, by the way, doesn't really have anything to do with programming language types, and is just a gadget for doing dispatch.)
3. We handle **device type / layout** dispatch (Type)
4. We have the actual kernel, which is either a modern native function, or a legacy TH function.

Each of these steps corresponds concretely to some code. Let's cut our way through the jungle.

```

static PyObject * THPVariable_add(PyObject* self_, PyObject* args, PyObject* kwargs)
{
    HANDLE_TH_ERRORS
    static PythonArgParser parser({
        "add(Tensor input, Scalar alpha, Tensor other, *, Tensor out=None) |deprecated",
        "add(Tensor input, Tensor other, *, Scalar alpha=1, Tensor out=None)",
    }, /*traceable=*/true);

    ParsedArgs<4> parsed_args;
    auto r = parser.parse(args, kwargs, parsed_args);

    if (r.idx == 0) {
        if (r.isNone(3)) {
            return wrap(dispatch_add(r.tensor(0), r.scalar(1), r.tensor(2)));
        } else {
            return wrap(dispatch_add(r.tensor(0), r.scalar(1), r.tensor(2), r.tensor(3)));
        }
    } else if (r.idx == 1) {
        if (r.isNone(3)) {
            return self.add(other, alpha);
        } else {
            return wrap(dispatch_add(r.tensor(0), r.tensor(1), r.scalar(2), r.tensor(3)));
        }
    }
    Py_RETURN_NONE;
    END_HANDLE_TH_ERRORS
}

static PyMethodDef torch_functions[] = {
    ...
    {"add", (PyCFunction)THPVariable_add, METH_VARARGS | METH_KEYWORDS | METH_STATIC, N
    ...

```

auto-generated! just skim

argument parser

release the GIL

AutoNoGIL no\_gil;

return self.add(other, alpha);

rewrap into PyObject

actual binding  
on torch.\_C.VariableFunctions

File: torch/csrc/autograd/generated/python\_torch\_functions\_dispatch.h

Our initial landing point in the C++ code is the C implementation of a Python function, which we've exposed to the Python side as something like `torch._C.VariableFunctions.add`. `THPVariable_add` is the implementation of one such implementation.

One important thing to know about this code is that it is auto-generated. If you search in the GitHub repository, you won't find it, because you have to actually build PyTorch to see it. Another important thing is, you don't have to really deeply understand what this code is doing; the idea is to skim over it and get a sense for what it is doing. Above, I've annotated some of the most important bits in blue: you can see that there is a use of a class `PythonArgParser` to actually pull out C++ objects out of the Python args and kwargs; we then call a `dispatch_add` function (which I've inlined in red); this releases the global interpreter lock and then calls a plain old method on the C++ Tensor `self`. On its way back, we rewrap the returned Tensor back into a `PyObject`.

(At this point, there's an error in the slides: I'm supposed to tell you about the Variable dispatch code. I haven't fixed it here yet. Some magic happens, then...)

```
inline Tensor Tensor::add(const Tensor & other, Scalar alpha) const {
    return type().add(*this, other, alpha);
}
```

*CPUFloatType* ↑      ↗ *virtual method*

File: aten/src/ATen/core/TensorMethods.h

*might be overridden by subclass*



*Switch (e.g. CUDA) device*



```
Tensor TypeDefault::add(const Tensor & self, const Tensor & other, Scalar alpha) const {
    const OptionalDeviceGuard device_guard(device_of(self));
    return at::native::add(/* native_actuals */ self, other, alpha);
}
```

File: build/aten/src/ATen/TypeDefault.cpp

```
Tensor add(const Tensor& self, const Tensor& other, Scalar alpha) {
    Tensor result;
    if (other.is_sparse()) {
        result = at::empty({0}, self.options());
        return native::add_out(result, self, other, alpha);
    }
    auto iter = TensorIterator::binary_op(result, self, other);
    add_stub(iter->device_type(), *iter, alpha);
    return iter->output();
}
```

File: aten/src/ATen/native/BinaryOps.cpp

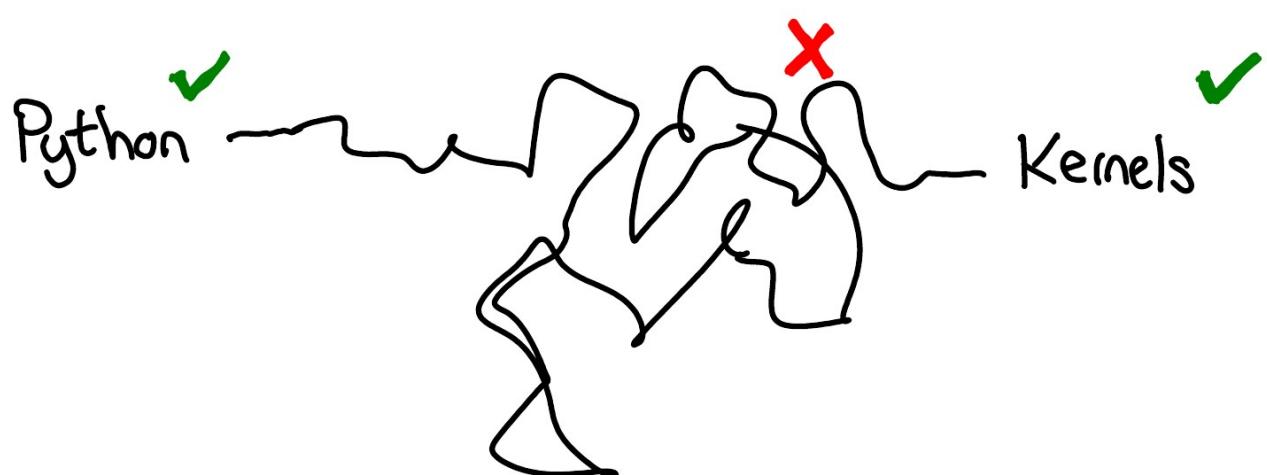
*the kernel  
proper*

When we call the `add` method on the `Tensor` class, no virtual dispatch happens yet. Instead, we have an inline method which calls a virtual method on a "Type" object. This method is the actual virtual method (this is why I say Type is just a "gadget" that gets you dynamic dispatch.) In the particular case of this example, this virtual call dispatches to an implementation of `add` on a class named `TypeDefault`. This happens to be because we have an implementation of `add` that is the same for every device type (both CPU and CUDA); if we had happened to have different implementations, we might have instead landed on something like `CPUFloatType::add`. It is this implementation of the virtual method that finally gets us to the actual kernel code.

Hopefully, this slide will be out-of-date very soon too; Roy Li is working on replacing Type dispatch with another mechanism which will help us better support PyTorch on mobile.

It's worth reemphasizing that all of the code, until we got to the kernel, is automatically generated.

I don't recommend spending too much time mucking around backtraces



It's a bit twisty and turny, so once you have some basic orientation about what's going on, I recommend just jumping straight to the kernels.

---

# Writing kernels

PyTorch offers a lot of useful tools for prospective kernel writers. In this section, we'll walk through a few of them. But first of all, what do you need to write a kernel?

# Anatomy of a kernel (+ Infra wiring up!)

```
Tensor my_op_out_cpu(Tensor& result, const Tensor& self, const Tensor& other) {
    TORCH_CHECK(result.is_cpu() && self.is_cpu() && other.is_cpu());
    TORCH_CHECK(self.dim() == 1);
    TORCH_CHECK(self.sizes() == other.sizes());
```

*Error checking*

```
    result.resize_(self.sizes());
```

*Output allocation*

```
    AT_DISPATCH_FORALL_TYPES(
        self.scalar_type(), "my_op_cpu", [&] {
            my_op_cpu_kernel<scalar_t>(result, self, other);
        }
    );
}
```

*Dtype dispatch*

```
template <typename scalar_t>
void my_op_cpu_kernel(Tensor& result, const Tensor& self, const Tensor& other) {
    auto result_accessor = result.accessor<scalar_t, 1>();
    auto self_accessor = self.accessor<scalar_t, 1>();
    auto other_accessor = other.accessor<scalar_t, 1>();

    parallel_for(0, self.size(0), 0, [&](int64_t start, int64_t end) {
        ... self_accessor[i] ...
    });
}
```

*Parallelization*

*Data access (vectorized? !)*

We generally think of a kernel in PyTorch consisting of the following parts:

1. First, there's some metadata which we write about the kernel, which powers the code generation and lets you get all the bindings to Python, without having to write a single line of code.
2. Once you've gotten to the kernel, you're past the device type / layout dispatch. The first thing you need to write is error checking, to make sure the input tensors are the correct dimensions. (Error checking is really important! Don't skimp on it!)
3. Next, we generally have to allocate the result tensor which we are going to write the output into.
4. Time for the kernel proper. At this point, you now should do the second, dtype dispatch, to jump into a kernel which is specialized per dtype it operates on. (You don't want to do this too early, because then you will be uselessly duplicating code that looks the same in any case.)
5. Most performant kernels need some sort of parallelization, so that you can take advantage of multi-CPU systems. (CUDA kernels are "implicitly" parallelized, since their programming model is built on top of massive parallelization).
6. Finally, you need to access the data and do the computation you wanted to do!

In the subsequent slides, we'll walk through some of the tools PyTorch has for helping you implementing these steps.

# Scaffolding

Every operator has a schema

- func: add(Tensor self, Tensor other, \*, Scalar alpha=1) -> Tensor  
variants: function, method  
dispatch:  
    CPU: add  
    CUDA: add  
    SparseCPU: add  
    SparseCUDA: add  
    MkldnnCPU: mkldnn\_add

Signature

control Python/C++ API

dispatch

aten/src/ATen/native/native\_functions.yaml

This schema controls our code generation

<https://github.com/pytorch/pytorch/blob/master/aten/src/ATen/native/README.md>

To take advantage of all of the code generation which PyTorch brings, you need to write a *schema* for your operator. The schema gives a mypy-esque type of your function, and also controls whether or not we generate bindings for methods or functions on Tensor. You also tell the schema what implementations of your operator should be called for given device-layout combinations. Check out the [README in native](#) for more information about this format.

# Scaffolding

Also a derivative:

Signature

```
- name: add(Tensor self, Tensor other, *, Scalar alpha)
  self: grad
  other: maybe_multiply(grad, alpha) } one grad per Tensor argument
```

grad\_self = grad\_add

the gradient flowing into add

tools/autograd/derivatives.yaml

You also may need to define a derivative for your operation in [derivatives.yaml](#).

# Error checking

## Low level

```
#include <c10/util/Exception.h>
TORCH_CHECK(self.dim() == 1, "Expected self to be 1-D tensor, but "
           "was ", self.dim(), "-D tensor")
```

Condition

Hand-write your error message.



## Higher level

```
#include <ATen/TensorUtils.h>
```

```
TensorArg result_arg{result, "result", 0}
    self_arg{self, "self", 1},
    other_arg{other, "other", 2}; }
```

Metadata about each argument

```
CheckedFrom c = "my_op_cpu";
checkDim(c, self_arg, 1);
```

↑  
Not complete!

↑ Pre-canned error message

Error checking can be done by way of either a low level or a high level API. The low level API is just a macro, `TORCH_CHECK`, which takes a boolean, and then any number of arguments to make up the error string to render if the boolean is not true. One nice thing about this macro is that you can intermix strings with non-string data; everything is formatted using their implementation of `operator<<`, and most important data types in PyTorch have `operator<<` implementations.

The high level API saves you from having to write up repetitive error messages over and over again. The way it works is you first wrap each Tensor into a `TensorArg`, which contains information about where the tensor came from (e.g., its argument name). It then provides a number of pre-canned functions for checking various properties; e.g., `checkDim()` tests if the tensor's dimensionality is a fixed number. If it's not, the function provides a user-friendly error message based on the `TensorArg` metadata.

# Output allocation

no-op if already the right size

```
Tensor& abs_out(Tensor& result, const Tensor& self) {  
    result.resize_(self.sizes());  
    // ... the real implementation ...  
}
```

preallocated  
output

```
Tensor abs(const Tensor& self) {  
    Tensor result = at::empty({0}, self.options());  
    abs_out(result, self);  
    return result;  
}
```

generate  
for me

```
Tensor& abs_(Tensor& self) {  
    return abs_out(self, self);  
}
```

inplace

↑ assume\_out  
will resize

↑ assumes\_out kernel handles  
aliasing inputs

One important thing to be aware about when writing operators in PyTorch, is that you are often signing up to write *three* operators: `abs_out`, which operates on a preallocated output (this implements the `out=` keyword argument), `abs_`, which operates `inplace`, and `abs`, which is the plain old functional version of an operator.

Most of the time, `abs_out` is the real workhorse, and `abs` and `abs_` are just thin wrappers around `abs_out`; but sometimes writing specialized implementations for each case are warranted.

# Dtype dispatch

not actually all types

```
AT_DISPATCH_ALL_TYPES(  
    self.scalar_type(), "my_op_cpu", [&] {  
        my_op_cpu_kernel<scalar_t>(result, self, other);  
    }  
)
```

specialize lambda for each scalar-type

```
AT_DISPATCH_ALL_TYPES(TYPE, NAME, ...)  
AT_DISPATCH_FLOATING_TYPES(TYPE, NAME, ...)  
AT_DISPATCH_INTEGRAL_TYPES(TYPE, NAME, ...)  
AT_DISPATCH_ALL_TYPES_AND(SCALARTYPE, TYPE, NAME, ...)
```

To do dtype dispatch, you should use the `AT_DISPATCH_ALL_TYPES` macro. This takes in the dtype of the tensor you want to dispatch over, and a lambda which will be specialized for each dtype that is dispatchable from the macro. Usually, this lambda just calls a templated helper function.

This macro doesn't just "do dispatch", it also decides what dtypes your kernel will support. As such, there are actually quite a few versions of this macro, which let you pick different subsets of dtypes to generate specializations for. Most of the time, you'll just want `AT_DISPATCH_ALL_TYPES`, but keep an eye out for situations when you might want to dispatch to some more types. There's guidance in [Dispatch.h](#) for how to select the correct one for your use-case.

# Parallelization (CPU only)

```
#pragma omp parallel for private(i) if (input_size > grain_size)  
for (i = 0; i < input_size; i++) {  
    ...  
}
```



```
at::parallel_for(0, input_size, grain_size, [&](int64_t begin, int64_t end) {  
    for (auto i = start; i < end; i++) {  
        // ...  
    }  
});
```



critical? implement the locks yourself!

On CPU, you frequently want to parallelize your code. In the past, this was usually done by directly sprinkling OpenMP pragmas in your code.

# Data access

## Point access

```
auto x_accessor = x.accessor<float, 3>();
float val = x_accessor[0][0][0];
                                CUDA: packed_accessor (gotcha: 32-bit!)
```

## Elementwise

```
#include <ATen/native/TensorIterator.h>
auto iter = TensorIterator::Builder()
    .add_output(output)
    .add_input(input)
    .build();
binary_kernel(iter, [](float a, float b) {
    return a + b;
});                                CUDA: gpu_binary_kernel
```

## Vectorized

```
#include <ATen/native/cpu/Loops.h>
binary_kernel_vec(iter,
    [=](scalar_t a, scalar_t b) -> scalar_t {
        return a + alpha * b; },
    [=](Vec256<scalar_t> a, Vec256<scalar_t> b) {
        return vec256::fmadd(b, alpha_vec, a);
});
```

At some point, we have to actually access the data. PyTorch offers quite a few options for doing this.

- If you just want to get a value at some specific location, you should use `TensorAccessor`. A tensor accessor is like a tensor, but it hard codes the dimensionality and dtype of the tensor as template parameters. When you retrieve an accessor like `x.accessor<float, 3>()`, we do a runtime test to make sure that the tensor really is this format; but after that, every access is unchecked. Tensor accessors handle strides correctly, so you should prefer using them over raw pointer access (which, unfortunately, some legacy kernels do.) There is also a `PackedTensorAccessor`, which is specifically useful for sending an accessor over a CUDA launch, so that you can get accessors from inside your CUDA kernel. (One notable gotcha: `TensorAccessor` defaults to 64-bit indexing, which is much slower than 32-bit indexing in CUDA!)
- If you're writing some sort of operator with very regular element access, for example, a pointwise operation, you are much better off using a higher level of abstraction, the `TensorIterator`. This helper class automatically handles broadcasting and type promotion for you, and is quite handy.
- For true speed on CPU, you may need to write your kernel using vectorized CPU instructions. We've got helpers for that too! The `Vec256` class represents a vector of scalars and provides a number of methods which perform vectorized operations on them all at once. Helpers like `binary_kernel_vec` then let you easily run vectorized operations, and then finish everything that doesn't round nicely into vector instructions using plain old instructions. The infrastructure here also manages compiling your kernel multiple times under different instruction sets, and then testing at runtime what instructions your CPU supports, and using the best kernel in those situations.

# Legacy code

TH, THC, THNN, THCUNN

- C code
- Manual refcounting
- Preprocessor shenanigans

THTensor\_wrap()

Write new code in C++; consider porting old code when you can!

<https://github.com/pytorch/pytorch/wiki/TH-to-ATen-porting-guide>

A lot of kernels in PyTorch are still written in the legacy TH style. (By the way, TH stands for TorchH. It's a pretty nice acronym, but unfortunately it is a bit poisoned; if you see TH in the name, assume that it's legacy.) What do I mean by the legacy TH style?

1. It's written in C style, no (or very little) use of C++.
2. It's manually refcounted (with manual calls to `THTensor_free` to decrease refcounts when you're done using tensors), and
3. It lives in `generic/` directory, which means that we are actually going to compile the file multiple times, but with different `#define scalar_t`.

This code is pretty crazy, and we hate reviewing it, so please don't add to it. One of the more useful tasks that you can do, if you like to code but don't know too much about kernel writing, is to port some of these TH functions to ATen.

# Workflow efficiency

- Don't edit headers
- Don't test by CI
- Do setup ccache
- Get a beefy workstation for builds

To wrap up, I want to talk a little bit about working efficiently on PyTorch. If the largeness of PyTorch's C++ codebase is the first gatekeeper that stops people from contributing to PyTorch, the efficiency of your workflow is the second gatekeeper. If you try to work on C++ with Python habits, **you will have a bad time**: it will take forever to recompile PyTorch, and it will take you forever to tell if your changes worked or not.

How to work efficiently could probably be a talk in and of itself, but this slide calls out some of the most common anti-patterns I've seen when someone complains: "It's hard to work on PyTorch."

1. If you edit a header, especially one that is included by many source files (and especially if it is included by CUDA files), expect a very long rebuild. Try to stick to editing cpp files, and edit headers sparingly!
2. Our CI is a very wonderful, zero-setup way to test if your changes worked or not. But expect to wait an hour or two before you get back signal. If you are working on a change that will require lots of experimentation, spend the time setting up a local development environment. Similarly, if you run into a hard to debug problem on a specific CI configuration, set it up locally. You can [download and run the Docker images locally](#).
3. The [CONTRIBUTING guide explains how to setup ccache](#); this is highly recommended, because sometimes it will help you get lucky and avoid a massive recompile when you edit a header. It also helps cover up bugs in our build system, when we recompile files when we shouldn't.
4. At the end of the day, we have a lot of C++ code, and you will have a much more pleasant experience if you build on a beefy server with CPUs and RAM. In particular, I don't recommend doing CUDA builds on a laptop; building CUDA is sloooooow and laptops tend to not have enough juice to turnaround quickly enough.

# How to get involved!

<https://github.com/pytorch/pytorch/issues>

high priority

module

small

Port some code from TH to ATen!

Help us improve our documentation!

Help minimize reproducers on bugs!

Help answer questions on the forums

Help us discuss design of new features!

Scratch your own itches!

So that's it for a whirlwind tour of PyTorch's internals! Many, many things have been omitted; but hopefully the descriptions and explanations here can help you get a grip on at least a substantial portion of the codebase.

Where should you go from here? What kinds of contributions can you make? A good place to start is our issue tracker. Starting earlier this year, we have been triaging issues; issues labeled **triaged** mean that at least one PyTorch developer has looked at it and made an initial assessment about the issue. You can use these labels to find out what issues we think are **high priority** or look up issues specific to some module, e.g., [autograd](#) or find issues which we think are **small** (word of warning: we're sometimes wrong!)

Even if you don't want to get started with coding right away, there are many other useful activities like improving documentation (I love merging documentation PRs, they are so great), helping us reproduce bug reports from other users, and also just helping us discuss RFCs on the issue tracker. PyTorch would not be where it is today without our open source contributors; we hope you can join us too!

- [May 16, 2019](#)
- [PyTorch](#)

## 58 Responses to “PyTorch internals”

1.  [wynne](#) says:  
[November 7, 2023 at 2:41 am](#)

wish a better doc about pytorch internal

2.  [Benjamin](#) says:  
[November 9, 2023 at 10:41 pm](#)

Hi Edward, is there any systematic docs on pytorch internals?

3.  [zuko](#) says:  
[November 29, 2023 at 6:14 pm](#)

thanks for the writeup :)

4.  [Avishake](#) says:  
[January 26, 2024 at 7:36 am](#)

Thanks for the amazing blog post that we have here.  
It was really very interesting.  
:)

5.  [dosmas](#) says:  
[March 21, 2024 at 12:28 am](#)

No one knows why c10 is c10:  
either it is Caffe TENSor, or Core TENSor, or Caffe 10 (in binary).

6. [PyTorch 内部机制: ezyang的博客 \(2019\) - 偏执的码农](#) says:  
[April 16, 2024 at 7:39 am](#)

[...] 详情参考 [...]

7. [Recreating PyTorch from Scratch \(with GPU Support and Automatic Differentiation\) | by Lucas de Lima Nogueira | May, 2024 – Play2Box](#) says:  
[May 14, 2024 at 1:24 pm](#)

[...] PyTorch internals — A information on how PyTorch is structured. [...]

8. [Recreating PyTorch from scratch \(with GPU support and automatic differentiation\) - Diss Solutions](#) says:  
[May 15, 2024 at 4:17 am](#)

[...] PyTorch internals—A guide on how PyTorch is structured. [...]

#### Leave a Comment

Name (Optional):

Comment:

[« Previous Post](#) [Next Post »](#)

© ezyang's blog. Powered by [WordPress](#), theme based off of [Ashley](#).