ATen "native" functions are the modern mechanism for adding operators and functions to ATen. Native functions are declared in `native_functions.yaml` and have implementations defined in one of the `cpp` files in this directory.

Like all ATen methods/functions, native functions are made available from both ATen's C++ and Python APIs. In C++, they are made available either as methods on `Tensor` (`t.mymeth()`) and functions in the ATen namespace (`at::myfunc()`). In PyTorch, they are made available as methods on `Variable` or as functions on `torch._C._FunctionBase`. (It is the user's responsibility to re-export these functions in a more user-facing module.)

The rest of this document describes how to implement an ATen function.

Registering a function in `native_functions.yaml`

Every native function must have an entry in `native_functions.yaml`. The format can be summarized as:

```
- func: func_name(ArgType arg0[=default], ArgType arg1[=default], ...) -> Return
  variants: function, method
  dispatch:
    CPU: func_cpu
    CUDA: func_cuda
```

Each component is described in more detail below:

`func`

```
- func: func_name[.overload_name](ArgType arg0[=default], ArgType arg1[=default], ...) -> Re
```

The `func` entry is a string describing the name of the function and its type signature.

Argument types. These types are permissible as ArgType:

- `Tensor`. A `Tensor` argument translates into a C++ argument of type `const Tensor&` (except when the argument is "inplace"; in this case, it is simply `Tensor&`). A trailing `?`, as in `Tensor?`, indicates that the tensor argument is optional and may be omitted by passing c10::nullopt. When a function takes multiple `Tensor` arguments, these tensors are assumed to be the same type (e.g., if one argument is a `FloatTensor`, all other arguments are checked to be `FloatTensor`s). `Tensor` or `Tensor?` must sometimes be annotated to indicate aliasing and mutability. In general annotations can be defined via the following four situations:
  - `Tensor(a)` - `a` is a set of Tensors that may alias to the same data.
  - `Tensor(a!)` - `a` members of a may be written to thus mutating the underlying data.
  - `Tensor!` - shorthand for Tensor(fresh_identifier!)

  - `Tensor(a! -> a|b)` - Tensor is in set `a`, written to, and after the write is in set `a` AND `b`. For more details on when and why this needs to happen, please see the section on annotations.
- `Tensor[]`. A `Tensor[]` argument translates into a C++ argument of type `ArrayRef<Tensor>` (a.k.a. `TensorList`)
- `int[]`. `int[]` accepts an optional length specifier, e.g., `int[2]`, which has no effect in C++ but extends our Python bindings to accept a bare number, which will be expanded into an appropriately sized list by repeating the number.
- `int`. Think about this like a Python int. This is translated into a C++ argument of type `int64_t`.
- `float`. Think about this like a Python `float`. It is translated into a C++ argument of type `double`.
- `bool`
- `str`. It is translated into a C++ argument of non-owning type `c10::string_view`
- `Scalar`. `Scalar` supports binding to any numerical types from Python, including integral types, floating point types, and zero dimensional tensors. `int` and `float` bind to the corresponding Python numerical types. However, you probably don't want to use `Scalar`; `float` and `int` argument types should suffice for most algorithms (you should only use `Scalar` if the operator truly may accept either type).
- `Generator?`, the state for a random number generator,
- `bool[N]` (where N is `1-4`).
- `*` is a special sentinel argument, which doesn't translate into an actual argument, but indicates that in the Python bindings, any subsequent arguments must be specified as keyword arguments (and cannot be provided positionally).
- `?` is trailing question mark that annotates an argument to be an optional type. Grep for `optional` to find some example usages. In general, most functions will not need to use this, but there are some cases that we want to use optional for the different types:
  - You want to pass a `None` to an ATen function/method from Python and handle the None type on the C++ side. For example, `clamp(Tensor self, Scalar? min=None, Scalar? max=None)` can take `None` for its `min` and `max` parameter, but does not dispatch to different backends if one of the parameters is `None`. Optional type can accept a `None` type (`nullopt` in C++) from Python and use the C++ Optional class to interact with the parameters.
  - You want a default value, which is fine in Python, but would cause ambiguity in C++. For example, `norm(Tensor self, Scalar p=2, int dim, bool keepdim=False)` would cause ambiguity in C++ since its default args must be adjacent (`p` could not have a default value when `dim` does not). Therefore, we need to make `p` as a optional Scalar, and make `p=2` when `p` is not passed in (nullopt).
  - You want a value to default to the same value as another argument

(this cannot be expressed in C++ default arguments).

Functions with no tensor inputs are called *factory functions*, and are handled specially by code generation. If your function is behaving differently than another example, check first and see if one is a factory while another is not. In some rare cases, factory function might have a tensor argument. In this case mark it with `category_override: factory` explicitly.

Argument names. Argument names are meaningful; downstream binding code may make use of the specific argument name you provide, and a rename of an argument name is considered a BC-breaking change (e.g., you will probably need to update `tools/autograd/derivatives.yaml` at least, and it may affect Python keyword arguments). For more details please see the section on `variants`.

As a convention we use 'out' to indicate an output argument. This aligns with the Python bindings. Even if a function might not be used in the Python bindings, we still advise to follow this convention. Check the generated code when making a change to make sure you're not breaking the API when renaming an argument name of an existing function.

Defaults. Any suffix of arguments can have a default value defined; these default values translate into C++/Python default values which are applied when those positional arguments are not specified.

Here are the supported default values:

- Numbers (e.g., `0` or `5.0` for `int`, `float` and `int[]` with an explicit length (e.g., `int[2]`)–in the case of `int[]` a number is replicated to fill the length (e.g., `int[2] x=2` is equivalent to `int[2] x=[2,2]`).
- Lists of numbers (e.g., `[0, 0]`) for `IntList`.
- Booleans (e.g., `True`) for `bool`.
- Empty initializer lists (e.g., `[]`) for `Tensor` (this implicitly changes a `Tensor` argument to accept undefined tensors).
- `None` for pointer types (e.g., `Generator?`)

Returns. The following are permissible on Return:

Non-tuple return:

```
ReturnType [retarg0]
```

Tuple return:

```
(ReturnType [retarg0], ReturnType [retarg1], ...)
```

The following are permissible on ReturnType: - `Tensor` and `Tensor[]`, which translate into the C++ types `Tensor` and `std::vector<Tensor>`, respectively (unless the operation is in-place, in which case the return type is `Tensor&`. - A tuple of any number of `Tensor`, e.g., `(Tensor, Tensor)`, translating into the C++ `std::tuple<Tensor, Tensor>`.

If you need a type that is not listed in this list, it may be possible to extend ATen's code generation to support it. ATen's philosophy on types to support is that it supports only simple, universal types, as well as a handful of fundamental Tensor structures (e.g., `Tensor` and `Generator?`), because these types can be easily ported to any language bound to ATen (in practice, C++ and Python.)

Return also supports specifying (optional) return argument names. These serve two functions:

- They let you easily write derivatives in terms of return arguments in `tools/autograd/derivatives.yaml`
- They correspond to the named field the output can be referred to from Python. (This means that changing a return argument name is BC-breaking, be careful!)

Note that argument type modifiers such as defaults and optional are not currently supported on Return.

Overloads. You can register multiple functions with the same name and different function signatures if you give them unique overload names. An overload name is specified after the function name, separated by a dot.

Overload names do not have to be globally unique, but must be unique in the set of all overloads for the same function. Overload names cannot be changed for backwards compatibility reasons. Please try to make overload names semantically meaningful. An overload name that just enumerates all the argument types isn't helpful. In many cases, a semantic name is clear from what the overload is doing differently. As a fallback, you can use the name or type of the first differing argument as an overload name.

If you add a new overload to an existing function, please leave the existing overload names as they are (for backwards compatibility), but give the new overload a new, unique name. Although overload names are not directly used by the Python or C++ APIs, they are public API surface for external backends (who register to specific overload names) and deployed mobile models (which use overload names as part of the serialization format.)

Not specifying an overload name is equivalent to specifying an empty overload name. If you add a new function with multiple overloads, give them unique overload names, at most one overload is allowed to have an empty overload name.

The declarations also support the following attributes.

`variants`

`variants: function, method`

Controls whether Tensor method (`t.foo()`) or namespace Function (`at::foo()`) is generated as a result of this declaration. If the declaration is a method,

you must have an argument `Tensor self` at some position in the method; in the method variant this argument will be elided from the argument list. For example, given the declaration `where(BoolTensor cond, Tensor self, Tensor other)`, this generates the function `at::where(cond, self, other)` and the method `self.where(cond, other)`.

By default, ATen generates only the function variant for a native function. When should you also generate a method variant? Tensor operations as methods are appropriate for "core" Tensor operations (e.g., add, sub, etc.), but not for more complicated neural network layers (e.g., `conv2d`) and internal functions designed specifically for binding (e.g., `cudnn_convolution`).

As we progress along our schema unification of the `func` schema with the JIT signature schema, we must introduce features that allow us to increase compliance. One of these features are Tensor annotations. As of now we use naming conventions to indicate whether an argument of a function is going to be mutated and returned.

`annotations`

There are two typical situations in which we mutate the memory of an argument in the Python frontend: a) For an inplace operations such as `self.abs_()` b) for a function with an output keyword argument such as `torch.abs(input, out=None)`.

In order to provide implementations for these Python functions the legacy schema requires C++ implementations for three situations `abs(Tensor self) -> Tensor`, `abs_(Tensor self) -> Tensor` and `abs_out(Tensor out, Tensor self) -> Tensor`.

Now, as we move towards the unification, we start to use a different syntax to represent this by using annotations. In the end we still translate to the legacy schema for the downstream consumers such as the C++ code generation, but this will soon change.

If two Tensors carry the same annotation, they both *may* represent the same memory. A write annotation, as indicated by an exclamation mark, indicates that they both *may* also be written to.

Let's revisit the previous native function declarations and see the conventions of adding annotations. - `abs(Tensor self) -> Tensor` stays the same as it will always allocate new memory. - `abs_(Tensor(a!) self) -> Tensor(a!)` `self` may be written to and returned. Further, the annotation indicates that the return value may alias the input. This indicates an inplace function and by convention ends in a single '_'. - `abs(Tensor self, *, Tensor(a!) out) -> Tensor(a!)` In the Python frontend `out` can be passed as a keyword argument and may be written to. In this case it indicates the schema for a function that must accept `out` as this does not provide a default argument. The idea behind representing this as a optional argument is to document the intended usage. This

maps to the legacy `abs_out(Tensor out, Tensor self) -> Tensor`. As with the legacy `_out` function you must call the argument `Tensor out` or `Tensor out0`, `Tensor out1` in the context of multiple arguments.

There is also another situation in which we use annotations, namely views. - `transpose(Tensor(a) self, int dim0, int dim1) -> Tensor(a)` An alias to the memory represented by `self` may be also returned, however it is not mutated.

When a Tensor views are contained in a Tensor list, we need to represent that the output list contains Tensors that alias the input. - `func: chunk(Tensor(a -> *) self, int chunks, int dim=0) -> Tensor(a)[]` We assume lists contain memory which aliases the heap, so in order to correctly set up the aliasing relationship between the output and input, we annotate that the input Tensor enters the wildcard set `(a -> *)`. For more details, see the JIT README.

We have some asserts to check whether a developer uses these annotations correctly and throw asserts if she doesn't. For example, any out function must use the `(a!)` annotation as described above. If this causes a lot of confusion please add @cpuhrsch to your PR.

`dispatch`

```
dispatch:
    CPU: func_cpu
    CUDA: func_cuda
```

This specifies the actual name of the function you want to dispatch to, so you can dispatch to different functions depending on which backend the passed tensors belong to. If the dispatch table is omitted, we assume a default dispatch table:

```
# overload is ignored
func: func.overload(...) -> ...
dispatch:
    CompositeImplicitAutograd: func

# overload is ignored, but out functions get suffixed with _out in their name
# (NB: no out functions in PyTorch today actually support autograd, but if they
# did, you could call them here and autograd would be inferred)
func: func.out_overload(...) -> ...
dispatch:
    CompositeImplicitAutograd: func_out
```

If two backends have the same dispatch function, you can write `CPU, CUDA: func` to reuse the same function name in both cases.

Available backend options can be found by searching `dispatch_keys` in codegen. There are also two special "generic" backends:

- `CompositeExplicitAutograd` (previously known as `DefaultBackend`): implementations of kernels that work for all backends, but require an explicit definition of backward function in `derivatives.yaml` to support autograd. The most typical use of this key are for delegating functions; i.e., functions that do a very small amount of work and then delegate to another operator to do the actual heavy lifting. Under the hood, registering a kernel to `CompositeExplicitAutograd` is equivalent to registering that kernel to every backend (e.g., `CPU, CUDA`). Note: kernels which call DispatchStub should NOT be registered as CompositeExplicitAutograd, as DispatchStub only works for `CPU, CUDA`)
- `CompositeImplicitAutograd` (previously known as `Math`): implementations of kernels that work for all backends, and also can implicitly support autograd, because all of the operations it calls support autograd. Direct use of this key should be rare: if you provide no dispatch table, we default to registering your kernel as `CompositeImplicitAutograd`. Explicitly adding this key to an existing dispatch table may be useful if you have specialized CPU and CUDA implementations, but you might want to provide a fallback lowering for external backends that may not have a specialized implementation.

Functions registered to composite backends should work for any backend, if the nested functions they call work for those backends.

For example, suppose `my_op` can be implemented in the following way:

```
at::Tensor my_op(const Tensor& self, const Tensor& other) {
  return self + 2 * other;
}
```

If we already know inference kernels and derivative formulas for operators `+` and `*` in our system, you can just register `my_op` to `CompositeImplicitAutograd` and both inference & autograd will just work. Although it seems we only write down the inference formula here, PyTorch autograd system would correctly set up the backward for `my_op` using the chain formula and derivatives of `+` & `*` operators. In other words `d_out/d_self = 1; d_out/d_other = 2` can be derived automatically from the `my_op` inference kernel. Of course if we don't have derivative formula defined for either `+` or `*`, backward of `my_op` can no longer be derived automatically.

Whether to use implicit or explicit autograd for your kernel can be decided by the following steps: 1. If you can, always start with a `CompositeImplicitAutograd` kernel that's composable from existing operators. 2. If you don't want to use the derived gradient formula from `CompositeImplicitAutograd` kernel for autograd, either to get better performance or better numerical stability, you should register the kernel with `CompositeExplicitAutograd` so that it's only used in inference. Later for autograd, depending on whether your autograd kernel works for all backends or not, you can put them in alias `Autograd` or

specific keys like `AutogradCPU`. 3. If you prefer to write backend-specific kernels, use reserved dispatch keys for your backend instead, e.g. `CPU/AutogradCPU`.

Important: because a `CompositeImplicitAutograd` kernel is implicitly registered for ops with no `dispatch:` section, when you add a backend-specific kernel (and hence a `dispatch:` section) to one of these, you **must** also add a `CompositeImplicitAutograd:` entry that names the old kernel implementation (it's named after the op, with _ added if applicable), so that it's still available for other backends to use.

If you implemented a native function in C++ and want to find out which dispatch keyword should be used in native_functions.yaml, please follow steps in dispatch keywords

CompositeImplicitAutograd Compliance

Functions registered as CompositeImplicitAutograd MUST work for most, if not all, backends. This means that we impose a set of constraints that make it more difficult to write a CompositeImplicitAutograd function than writing regular PyTorch code.

If you wish to do something that is banned (you may wish to do this for perf reasons), please write a backwards formula for your operator so it is no longer CompositeImplicitAutograd or hide parts of the operator in a new operator that is not CompositeImplicitAutograd.

CompositeImplicitAutograd operators must not: - call `resize_` or moral equivalents. These are tricky to handle for many backends, like vmap and meta. - call `out=` operations. These are impossible to handle for vmap and can cause dispatch-to-python objects to lose their subclassing. - Change the metadata of a Tensor without performing dispatches. Examples of these operations are directly accessing the TensorImpl API to modify the sizes/strides/metadata of a Tensor. - In the same vein as the last point, `data_ptr` access or `item` access are not allowed. These operations do not go through the dispatcher. - `copy_` is a marginal case. If you're able to rewrite your operation without `copy_` you should definitely do so; this should be trivial if you're not copy-ing into a view. Otherwise, it is fine to leave the code as-is.

We have CompositeImplicitAutograd compliance tests in `test/test_ops.py`. These tests aren't perfect (it's pretty difficult to check for all of the above) so if something looks wrong please shout.

`device_guard`

`device_guard: False`

By default, ATen code generation will generate a DeviceGuard invocation, which will ensure that kernel code will run with the current device set to match the device of the first Tensor argument (or first tensor of the first Tensor[] argument,

if the function takes a list of tensors). For the most part, this means kernel authors do not have to worry about setting devices.

However, in some cases, setting the device is unnecessary, because, e.g., you call a function already manages device guard setting, or you're a function that simply does not interact with any devices. In that case, code generation of the device guard can be disabled by adding `device_guard: False` to your function definition.

`device_check`

`device_check: NoCheck`

By default, ATen code generation will generate device check, which will ensure all the tensor parameters passed to kernel are on the same device.

However, in some cases, checking the device is unncessary, because, e.g., you call a function allows to work on multiple devices. In that case, code generation of the device check can be disabled by adding `device_check: NoCheck` to your function definition.

`manual_kernel_registration`

`manual_kernel_registration: True`

With this flag set, we will not generate code to automatically register the C++ operator implementation to TypeDefault (catchAll dispatch key) with the dispatcher. It doesn't make sense to have both `dispatch` section and `manual_kernel_registration: True` for the same op. You can find the manual registrations in torch/csrc/autograd/VariableTypeManual.cpp. Currently ops have this field set to True should match `MANUAL_CATCHALL` in tools/autograd/gen_variable_type.py (It can be a superset of `MANUAL_CATCHALL` but we don't have a use case for it). This field should only be used rarely.

`use_const_ref_for_mutable_tensors`

`use_const_ref_for_mutable_tensors: True`

With this flag set, we will generate arguments for Tensors whose underlying data may change as `const Tensor&` (or similar), just like we would for other Tensors. Previously, we generated these as `Tensor &`, which 1) allowed changing which `TensorImpl` the `Tensor` itself referred to and 2) was not necessary to allow the underlying data to change. (This was like using `T * const` when we wanted `const T*`.)

Writing an implementation in C++

Implementations of native functions go in an appropriate C++ file in the `native/` directory (they are organized roughly by topic, but there is no semantic meaning

to their organization aside for the `cuda` directory, which is the only place the build system knows how to build `cu` files.) To write a native function, you only need to write a C++ implementation (no header necessary) with a matching signature to the generated header from the ATen metadata. There are many simple native functions; take a look at some of them to see what to do.

Although writing an ATen function is mostly writing the algorithm you want to implement, there are some less obvious details you should also consider.

Will your function be automatically differentiable?

If you are writing a pair of functions `foo` and `foo_backward`, with the intent that `foo_backward` implements the derivative of `foo`, then your implementation of `foo` is probably not automatically differentiable: it might make use of functions like `data_ptr()` or it dispatches differently depending on if it's operating on CPU or CUDA tensors. Once you write these two functions, you will have to write an entry correlating them together in `tools/autograd/derivatives.yaml`.

However, in some situations, you can write a function in ATen and it will be automatically differentiated! This can be the case if the function implementation only calls other operations which are themselves differentiable. In this case, you don't have to write an entry in `tools/autograd/derivatives.yaml`.

Choosing the right dispatch keyword

After writing a native function in C++, it's important to think about which dispatch keyword to use in native_functions.yaml as it gives the dispatcher information about backend and autograd support of the implementation.

Here're steps to follow to decide the right dispatch keyword:

1. Think about inference: does your kernel work for all backends?

    - No: you're likely providing different kernels for different backends, e.g. backend-dependent logic is used in the implementation or it's implemented through DispatchStub. DispatchStub only support a backend if you explicitly provide a kernel through `REGISTER_DISPATCH`. Typically it only supports a few in-tree backends like CPU, CUDA, QuantizedCPU etc but not out-of-tree backends like XLA. Write a dispatch section, enumerate all supported backends and point them to the implementations.

      ```
      dispatch:
        CPU: kernel_cpu
        CUDA: kernel_cuda
        QuantizedCPU: kernel_quantized_cpu
      ```

      You're done. Now this op will be called in `CPU/CUDA/QuantizedCPU` backend inference!

Note: to support training, you're required to write a formula in derivatives.yaml since your backend implementations don't support autograd.

- Yes: you're likely calling other `at::` ops in the implemetation. Go to step 2.

2. Think about training: does your kernel support autograd? check autograd support

   - Yes: in other words, you're providing a `CompositeImplicitAutograd` kernel which supports both inference and autograd. To use autograd support for training, simply skip adding a dispatch section and you're done. This will allow this op to be correctly registered for both inference and training.

   - Yes, but you still want to provide a numerically stable gradient formula instead of using autograd, write

     ```
     dispatch:
       CompositeExplicitAutograd: kernel
     ```

     You're done. This op will be called in inference for all backends.

     Note: to support training you're required to add an autograd formula, or it'll error out in backward pass when calling with a Tensor has requires_grad=True.

   - No: ops in this category are mainly using `_out` boilerplate where its out version doesn't have a derivative formula defined. For example:

     ```
     Tensor& sign_out(Tensor& result, const Tensor& self) { return unary_op_impl_out(res
     Tensor sign(const Tensor& self) { return unary_op_impl(self, at::sign_out); }
     Tensor& sign_(Tensor& self) { return unary_op_impl_(self, at::sign_out); }
     ```

     `sign_out` uses DispatchStub so the supported backends are enumerated in its dispatch section. For `sign` and `sign_`, write

     ```
     dispatch:
       CompositeExplicitAutograd: kernel
     ```

     You're done. This op will be called in inference for all backends.

     Note: to support training you're required to add an autograd formula for `sign`, or it'll error out in backward pass when calling with a Tensor has requires_grad=True.

     Note: current plan on record for ops using this boilerplate is to replace `at::` with `at::native` in the implementations and add dispatch section with device keywords instead.

3. Validate the computed dispatch table matches what you want. You can use `PythonDispatcher` provided in torch/_python_dispatcher.py. It shows

for a certain operator, what the computed dispatch table looks like after your registrations.

```
dispatcher = PythonDispatcher()
dispatcher.register(["CPU", "XLA", "AutogradCPU", "CompositeImplicitAutograd"])
print(dispatcher.dispatchTable()) # Tells you exactly which kernel is used for certain
```

4. TODO: AutogradCPUOrCUDA

Note that in native_functions.yaml you can mix using backend keywords and alias keywords above for one op: - direct registration to backend always has higher precendence than alias - DO NOT provide multiple alias keywords to the same op: alias keywords have precedence `CompositeExplicitAutograd > CompositeImplicitAutograd`, e.g. adding both `CompositeImplicitAutograd` and `CompositeExplicitAutograd` kernels for one op will completely ignore `CompositeImplicitAutograd` kernel for both inference and training. Thus this will trigger an error when native_functions.yaml is parsed.

Will this function be exposed to python? What are the namespaces?

We don't generate python bindings for all functions. There're certain patterns in function name that we skip in python binding generation, e.g. `*_backward`. Check `tools/autograd/gen_python_functions.py` for the latest rules.

The generated bindings are either exposed as methods on python_variable or functions on the torch._C._nn (marked with `python_module: nn`), torch._C._fft (marked with `python_module: fft`), torch._C._linalg (marked with `python_module: linalg`) objects, torch._C._sparse (marked with `python_module: sparse`) objects, or torch._C._special (marked with `python_module: special`) objects.

Undefined tensor conventions

By default, `Tensor` arguments to ATen functions are always defined, unless you explicitly specified that an undefined tensor was permissible by writing `Tensor?` or `Tensor? x=[]`, the latter one is needed when you have to assign a default value in C++ (e.g. in the middle of other parameters with default values).

The rules for returning undefined Tensors are a bit more subtle, but there is only one case you have to remember:

- If the function in question is a backward function which accepts a `std::array<bool,N> output_mask` argument, you MUST return an undefined `Tensor` at every tuple position `i` for which `output_mask[i]` is false, otherwise
- You MUST NOT return an undefined tensor.

The most common situations where you might be tempted to return undefined tensors are when:

- You have a forward function that may return a buffer if training is enabled, but does not return the buffer in inference mode. In this case, just return an appropriately typed zero-size tensor.
- You have a backward function where the gradient for an input is zero. In this case, you are expected to create a zero-filled tensor of appropriate size to return for this input. To get the shape, it may be helpful to take a `TensorGeometry` of the input to use.

Debugging tips

If you build ATen and get a linker error, that probably means you copy-pasted the C++ definition of your function incorrectly. Double check your `Tensor` arguments, and make sure you wrote `const Tensor&` in your signature.