Extending PyTorch

In this note we'll cover ways of extending :mod:`torch.nn`, :mod:`torch.autograd`, :mod:`torch`, and writing custom C extensions utilizing our C libraries.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 4); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 4); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 4); *backlink*

Unknown interpreted text role "mod".

Extending :mod:`torch.autograd`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 10); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 13)

Unknown directive type "currentmodule".

```
.. currentmodule:: torch.autograd
```

Adding operations to :mod:`~torch.autograd` requires implementing a new :class:`Function` subclass for each operation. Recall that Functions are what :mod:`~torch.autograd` uses to encode the operation history and compute gradients.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 15); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 15); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 15); *backlink*

Unknown interpreted text role "mod".

The first part of this doc is focused on backward mode AD as it is the most widely used feature. A section at the end discusses the extensions for forward mode AD.

When to use

In general, implement a custom function if you want to perform computations in your model that are not differentiable or rely on non-Pytorch libraries (e.g., NumPy), but still wish for your operation to chain with other ops and work with the autograd engine.

In some situations, custom functions can also be used to improve performance and memory usage: If you implemented your forward and backward passes using a C++ extension, you can wrap them in :class:`~Function` to interface with the autograd engine. If you'd like to reduce the number of buffers saved for the backward pass, custom functions can be used to combine ops together.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 29); *backlink*

Unknown interpreted text role "class".

When not to use

If you can already write your function in terms of PyTorch's built-in ops, its backward graph is (most likely) already able to be recorded by autograd. In this case, you do not need to implement the backward function yourself. Consider using a plain old Python function.

If you need to maintain state, i.e., trainable parameters, you should (also) use a custom module. See the section below for more information on extending :mod:`torch.nn`.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 43); *backlink*
>
> Unknown interpreted text role "mod".

If you'd like to alter the gradients during the backward pass or perform a side effect, consider registering a tensor or Module hook.

How to use

Take the following steps: 1. Subclass :class:`~Function` and implement the :meth:`~Function.forward` and :meth:`~Function.backward` methods. 2. Call the proper methods on the *ctx* argument. 3. Declare whether your function supports double backward. 4. Validate whether your gradients are correct using gradcheck.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 53); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 53); *backlink*
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 53); *backlink*
>
> Unknown interpreted text role "meth".

Step 1: After subclassing :class:`Function`, you'll need to define 2 methods:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 61); *backlink*
>
> Unknown interpreted text role "class".

- :meth:`~Function.forward` is the code that performs the operation. It can take as many arguments as you want, with some of them being optional, if you specify the default values. All kinds of Python objects are accepted here. :class:`Tensor` arguments that track history (i.e., with `requires_grad=True`) will be converted to ones that don't track history before the call, and their use will be registered in the graph. Note that this logic won't traverse lists/dicts/any other data structures and will only consider tensors that are direct arguments to the call. You can return either a single :class:`Tensor` output, or a :class:`tuple` of tensors if there are multiple outputs. Also, please refer to the docs of :class:`Function` to find descriptions of useful methods that can be called only from :meth:`~Function.forward`.

  > **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 63); *backlink*
  >
  > Unknown interpreted text role "meth".

  > **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 63); *backlink*
  >
  > Unknown interpreted text role "class".

  > **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 63); *backlink*
  >
  > Unknown interpreted text role "class".

  > **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source]

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 63); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 63); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 63); backlink

Unknown interpreted text role "meth".

- :meth:`~Function.backward` (or :meth:`~Function.vjp`) defines the gradient formula. It will be given as many :class:`Tensor` arguments as there were outputs, with each of them representing gradient w.r.t. that output. It is important NEVER to modify these in-place. It should return as many tensors as there were inputs, with each of them containing the gradient w.r.t. its corresponding input. If your inputs didn't require gradient (:attr:`~ctx.needs_input_grad` is a tuple of booleans indicating whether each input needs gradient computation), or were non-:class:`Tensor` objects, you can return :class:`python:None`. Also, if you have optional arguments to :meth:`~Function.forward` you can return more gradients than there were inputs, as long as they're all :any:`python:None`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 75); backlink

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 75); backlink

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 75); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 75); backlink

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 75); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 75); backlink

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 75); backlink

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 75); backlink

Unknown interpreted text role "any".

Step 2: It is your responsibility to use the functions in the forward's *ctx* properly in order to ensure that the new :class:`Function` works properly with the autograd engine.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 87);

backlink

Unknown interpreted text role "class".

- :meth:`~torch.autograd.function.FunctionCtx.save_for_backward` must be used when saving input or output tensors of the forward to be used later in the backward. Anything else, i.e., non-tensors and tensors that are neither input nor output should be stored directly on *ctx*.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 91); *backlink*

  Unknown interpreted text role "meth".

- :meth:`~torch.autograd.function.FunctionCtx.mark_dirty` must be used to mark any input that is modified inplace by the forward function.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 95); *backlink*

  Unknown interpreted text role "meth".

- :meth:`~torch.autograd.function.FunctionCtx.mark_non_differentiable` must be used to tell the engine if an output is not differentiable. By default all output tensors that are of differentiable type will be set to require gradient. Tensors of non-differentiable type (i.e., integral types) are never marked as requiring gradients.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 97); *backlink*

  Unknown interpreted text role "meth".

- :meth:`~torch.autograd.function.FunctionCtx.set_materialize_grads` can be used to tell the autograd engine to optimize gradient computations in the cases where the output does not depend on the input by not materializing grad tensors given to backward function. That is, if set to False, None object in python or "undefined tensor" (tensor x for which x.defined() is False) in C++ will not be converted to a tensor filled with zeros prior to calling backward, and so your code will need to handle such objects as if they were tensors filled with zeros. The default value of this setting is True.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 102); *backlink*

  Unknown interpreted text role "meth".

Step 3: If your :class:`~Function` does not support double backward you should explicitly declare this by decorating backward with the :func:`~function.once_differentiable`. With this decorator, attempts to perform double backward through your function will produce an error. See our double backward tutorial for more information on double backward.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 110); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 110); *backlink*

Unknown interpreted text role "func".

Step 4: It is recommended that you use :func:`torch.autograd.gradcheck` to check whether your backward function correctly computes gradients of the forward by computing the Jacobian matrix using your backward function and comparing the value element-wise with the Jacobian computed numerically using finite-differencing.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 116); *backlink*

Unknown interpreted text role "func".

Example

Below you can find code for a `Linear` function from :mod:`torch.nn`, with additional comments:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 125); *backlink*

Unknown interpreted text role "mod".

```
# Inherit from Function
class LinearFunction(Function):

    # Note that both forward and backward are @staticmethods
    @staticmethod
    # bias is an optional argument
    def forward(ctx, input, weight, bias=None):
        ctx.save_for_backward(input, weight, bias)
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    # This function has only a single output, so it gets only one gradient
    @staticmethod
    def backward(ctx, grad_output):
        # This is a pattern that is very convenient - at the top of backward
        # unpack saved_tensors and initialize all gradients w.r.t. inputs to
        # None. Thanks to the fact that additional trailing Nones are
        # ignored, the return statement is simple even when the function has
        # optional inputs.
        input, weight, bias = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = None

        # These needs_input_grad checks are optional and there only to
        # improve efficiency. If you want to make your code simpler, you can
        # skip them. Returning gradients for inputs that don't require it is
        # not an error.
        if ctx.needs_input_grad[0]:
            grad_input = grad_output.mm(weight)
        if ctx.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)
        if bias is not None and ctx.needs_input_grad[2]:
            grad_bias = grad_output.sum(0)

        return grad_input, grad_weight, grad_bias
```

Now, to make it easier to use these custom ops, we recommend aliasing their `apply` method:

```
linear = LinearFunction.apply
```

Here, we give an additional example of a function that is parametrized by non-Tensor arguments:

```
class MulConstant(Function):
    @staticmethod
    def forward(ctx, tensor, constant):
        # ctx is a context object that can be used to stash information
        # for backward computation
        ctx.constant = constant
        return tensor * constant

    @staticmethod
    def backward(ctx, grad_output):
        # We return as many input gradients as there were arguments.
        # Gradients of non-Tensor arguments to forward must be None.
        return grad_output * ctx.constant, None
```

And here, we optimize the above example by calling set_materialize_grads(False):

```
class MulConstant(Function):
    @staticmethod
    def forward(ctx, tensor, constant):
        ctx.set_materialize_grads(False)
        ctx.constant = constant
        return tensor * constant

    @staticmethod
    def backward(ctx, grad_output):
        # Here we must handle None grad_output tensor. In this case we
        # can skip unnecessary computations and just return None.
        if grad_output is None:
            return None, None

        # We return as many input gradients as there were arguments.
        # Gradients of non-Tensor arguments to forward must be None.
        return grad_output * ctx.constant, None
```

Note

Inputs to `backward`, i.e., :attr:`grad_output`, can also be tensors that track history. So if `backward` is implemented with differentiable operations, (e.g., invocation of another custom :class:`~torch.autograd.Function`), higher order derivatives will work. In this case, the tensors saved with `save_for_backward` can also be used in the backward and have gradients flowing back but tensors saved in the `ctx` won't have gradients flowing back for them. If you need gradients to flow back for a Tensor saved in the `ctx`, you should make it an output of the custom `Function` and save it with `save_for_backward`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 208); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 208); *backlink*

Unknown interpreted text role "class".

You probably want to check if the backward method you implemented actually computes the derivatives of your function. It is possible by comparing with numerical approximations using small finite differences:

```
from torch.autograd import gradcheck

# gradcheck takes a tuple of tensors as input, check if your gradient
# evaluated with these tensors are close enough to numerical
# approximations and returns True if they all verify this condition.
input = (torch.randn(20,20,dtype=torch.double,requires_grad=True), torch.randn(30,20,dtype=torch.double,requires_g
test = gradcheck(linear, input, eps=1e-6, atol=1e-4)
print(test)
```

See :ref:`grad-check` for more details on finite-difference gradient comparisons. If your function is used in higher order derivatives (differentiating the backward pass) you can use the `gradgradcheck` function from the same package to check higher order derivatives.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 231); *backlink*

Unknown interpreted text role "ref".

Forward mode AD

Overriding the forward mode AD formula has a very similar API with some different subtleties. You can implement the :meth:`~Function.jvp` function.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 238); *backlink*

Unknown interpreted text role "meth".

It will be given as many :class:`Tensor` arguments as there were inputs, with each of them representing gradient w.r.t. that input. It should return as many tensors as there were outputs, with each of them containing the gradient w.r.t. its corresponding output. The :meth:`~Function.jvp` will be called just after the :meth:`~Function.forward` method, before the :meth:`~Function.apply` returns.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 241); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 241); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 241); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 241); *backlink*

Unknown interpreted text role "meth".

:meth:`~Function.jvp` has a few subtle differences with the :meth:`~Function.backward` function:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 247); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 247); *backlink*

Unknown interpreted text role "meth".

- You can use the *ctx* to pass any data from the :meth:`~Function.forward` to the :meth:`~Function.jvp` function. If that state will not be needed for the :meth:`~Function.backward`, you can explicitly free it by doing `del ctx.foo` at the end of the :meth:`~Function.jvp` function.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 249); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 249); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 249); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 249); *backlink*

Unknown interpreted text role "meth".

- The implementation of :meth:`~Function.jvp` must be backward differentiable or explicitly check that none of the given forward mode gradient has `requires_grad` set.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 252); *backlink*

Unknown interpreted text role "meth".

- The :meth:`~Function.jvp` function must match the view/inplace behavior of :meth:`~Function.forward`. For example, if the `i` th input is modified inplace, then the `i` th gradient must be updated inplace. Similarly, if the `j` th output is a view of the `k` th input. Then the returned `j` th output gradient must be a view of the given `k` th input gradient.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 254); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 254); *backlink*

Unknown interpreted text role "meth".

- Because the user cannot specify which gradient needs to be computed, the :meth:`~Function.jvp` function should always compute gradients for all the outputs.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 258); *backlink*

Unknown interpreted text role "meth".

- The forward mode gradients do respect the flag set by :meth:`~torch.autograd.function.FunctionCtx.set_materialize_grads` and you can get *None* input gradients when this is disabled.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 260); *backlink*

Unknown interpreted text role "meth".

Extending :mod:`torch.nn`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 264); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 267)

Unknown directive type "currentmodule".

```
.. currentmodule:: torch.nn
```

:mod:`~torch.nn` exports two kinds of interfaces - modules and their functional versions. You can extend it in both ways, but we recommend using modules for all kinds of layers, that hold any parameters or buffers, and recommend using a functional form parameter-less operations like activation functions, pooling, etc.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 269); *backlink*

Unknown interpreted text role "mod".

Adding a functional version of an operation is already fully covered in the section above.

Adding a :class:`Module`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 278); *backlink*

Unknown interpreted text role "class".

Since :mod:`~torch.nn` heavily utilizes :mod:`~torch.autograd`, adding a new :class:`Module` requires implementing a :class:`~torch.autograd.Function` that performs the operation and can compute the gradient. From now on let's assume that we want to implement a `Linear` module and we have the function implemented as in the listing above. There's very little code required to add this. Now, there are two functions that need to be implemented:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 281); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 281); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 281); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 281); *backlink*

Unknown interpreted text role "class".

- `__init__` (*optional*) - takes in arguments such as kernel sizes, numbers of features, etc. and initializes parameters and buffers.
- :meth:`~Module.forward` - instantiates a :class:`~torch.autograd.Function` and uses it to perform the operation. It's very similar to a functional wrapper shown above.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 290); *backlink*

  Unknown interpreted text role "meth".

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 290); *backlink*

  Unknown interpreted text role "class".

This is how a `Linear` module can be implemented:

```
class Linear(nn.Module):
    def __init__(self, input_features, output_features, bias=True):
        super(Linear, self).__init__()
        self.input_features = input_features
        self.output_features = output_features

        # nn.Parameter is a special kind of Tensor, that will get
        # automatically registered as Module's parameter once it's assigned
        # as an attribute. Parameters and buffers need to be registered, or
        # they won't appear in .parameters() (doesn't apply to buffers), and
        # won't be converted when e.g. .cuda() is called. You can use
        # .register_buffer() to register buffers.
        # nn.Parameters require gradients by default.
        self.weight = nn.Parameter(torch.empty(output_features, input_features))
        if bias:
            self.bias = nn.Parameter(torch.empty(output_features))
        else:
            # You should always register all possible parameters, but the
            # optional ones can be None if you want.
            self.register_parameter('bias', None)

        # Not a very smart way to initialize weights
        nn.init.uniform_(self.weight, -0.1, 0.1)
        if self.bias is not None:
            nn.init.uniform_(self.bias, -0.1, 0.1)

    def forward(self, input):
        # See the autograd section for explanation of what happens here.
        return LinearFunction.apply(input, self.weight, self.bias)

    def extra_repr(self):
        # (Optional)Set the extra information about this module. You can test
        # it by printing an object of this class.
        return 'input_features={}, output_features={}, bias={}'.format(
            self.input_features, self.output_features, self.bias is not None
        )
```

Extending :mod:`torch`

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 335);** *backlink*
>
> Unknown interpreted text role "mod".

You can create custom types that emulate :class:`Tensor` by defining a custom class with methods that match :class:`Tensor`. But what if you want to be able to pass these types to functions like :func:`torch.add` in the top-level :mod:`torch` namespace that accept :class:`Tensor` operands?

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 338);** *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 338);** *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 338);** *backlink*
>
> Unknown interpreted text role "func".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 338);** *backlink*
>
> Unknown interpreted text role "mod".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 338);** *backlink*
>
> Unknown interpreted text role "class".

If your custom python type defines a method named `__torch_function__`, PyTorch will invoke your `__torch_function__` implementation when an instance of your custom class is passed to a function in the :mod:`torch` namespace. This makes it possible to define custom implementations for any of the functions in the :mod:`torch` namespace which your `__torch_function__`

implementation can call, allowing your users to make use of your custom type with existing PyTorch workflows that they have already written for :class:`Tensor`. This works with "duck" types that are unrelated to :class:`Tensor` as well as user-defined subclasses of :class:`Tensor`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 343); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 343); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 343); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 343); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 343); *backlink*

Unknown interpreted text role "class".

Extending :mod:`torch` with a :class:`Tensor`-like type

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 353); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 353); *backlink*

Unknown interpreted text role "class".

Note

This functionality is inspired by the NumPy `__array_function__` protocol. See the NumPy documentation and NEP-0018 for more details.

To make this concrete, let's begin with a simple example that illustrates the API dispatch mechanism. We'll create a custom type that represents a 2D scalar tensor, parametrized by the order `N` and value along the diagonal entries, `value`:

```
class ScalarTensor(object):
    def __init__(self, N, value):
        self._N = N
        self._value = value

    def __repr__(self):
        return "DiagonalTensor(N={}, value={})".format(self._N, self._value)

    def tensor(self):
        return self._value * torch.eye(self._N)
```

This first iteration of the design isn't very useful. The main functionality of `ScalarTensor` is to provide a more compact string representation of a scalar tensor than in the base tensor class:

```
>>> d = ScalarTensor(5, 2)
>>> d
ScalarTensor(N=5, value=2)
>>> d.tensor()
tensor([[2., 0., 0., 0., 0.],
        [0., 2., 0., 0., 0.],
        [0., 0., 2., 0., 0.],
        [0., 0., 0., 2., 0.],
        [0., 0., 0., 0., 2.]])
```

If we try to use this object with the :mod:`torch` API, we will run into issues:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 393); *backlink*

Unknown interpreted text role "mod".

```
>>> import torch
>>> torch.mean(d)
TypeError: mean(): argument 'input' (position 1) must be Tensor, not ScalarTensor
```

Adding a `__torch_function__` implementation to `ScalarTensor` makes it possible for the above operation to succeed. Let's redo our implementation, this time adding a `__torch_function__` implementation:

```
HANDLED_FUNCTIONS = {}
class ScalarTensor(object):
    def __init__(self, N, value):
        self._N = N
        self._value = value

    def __repr__(self):
        return "DiagonalTensor(N={}, value={})".format(self._N, self._value)

    def tensor(self):
        return self._value * torch.eye(self._N)

    @classmethod
    def __torch_function__(cls, func, types, args=(), kwargs=None):
        if kwargs is None:
            kwargs = {}
        if func not in HANDLED_FUNCTIONS or not all(
            issubclass(t, (torch.Tensor, ScalarTensor))
            for t in types
        ):
            return NotImplemented
        return HANDLED_FUNCTIONS[func](*args, **kwargs)
```

The `__torch_function__` method takes four arguments: `func`, a reference to the torch API function that is being overridden, `types`, the list of types of Tensor-likes that implement `__torch_function__`, `args`, the tuple of arguments passed to the function, and `kwargs`, the dict of keyword arguments passed to the function. It uses a global dispatch table named `HANDLED_FUNCTIONS` to store custom implementations. The keys of this dictionary are functions in the `torch` namespace and the values are implementations for `ScalarTensor`.

Note

Using a global dispatch table is not a mandated part of the `__torch_function__` API, it is just a useful design pattern for structuring your override implementations.

This class definition isn't quite enough to make `torch.mean` do the right thing when we pass it a `ScalarTensor` -- we also need to define an implementation for `torch.mean` for `ScalarTensor` operands and add the implementation to the `HANDLED_FUNCTIONS` dispatch table dictionary. One way of doing this is to define a decorator:

```
import functools
def implements(torch_function):
    """Register a torch function override for ScalarTensor"""
    @functools.wraps(torch_function)
    def decorator(func):
        HANDLED_FUNCTIONS[torch_function] = func
        return func
    return decorator
```

which can be applied to the implementation of our override:

```
@implements(torch.mean)
def mean(input):
    return float(input._value) / input._N
```

With this change we can now use `torch.mean` with `ScalarTensor`:

```
>>> d = ScalarTensor(5, 2)
>>> torch.mean(d)
0.4
```

Of course `torch.mean` is an example of the simplest kind of function to override since it only takes one operand. We can use the same machinery to override a function that takes more than one operand, any one of which might be a tensor or tensor-like that defines `__torch_function__`, for example for :func:`torch.add`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 467); *backlink*

Unknown interpreted text role "func".

```
def ensure_tensor(data):
    if isinstance(data, ScalarTensor):
        return data.tensor()
    return torch.as_tensor(data)

@implements(torch.add)
def add(input, other):
   try:
```

```
            if input._N == other._N:
                return ScalarTensor(input._N, input._value + other._value)
            else:
                raise ValueError("Shape mismatch!")
        except AttributeError:
            return torch.add(ensure_tensor(input), ensure_tensor(other))
```

This version has a fast path for when both operands are `ScalarTensor` instances and also a slower path which degrades to converting the data to tensors when either operand is not a `ScalarTensor`. That makes the override function correctly when either operand is a `ScalarTensor` or a regular :class:`Tensor`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 488); *backlink*

Unknown interpreted text role "class".

```
>>> s = ScalarTensor(2, 2)
>>> torch.add(s, s)
DiagonalTensor(N=2, value=4)
>>> t = torch.tensor([[1, 1,], [1, 1]])
>>> torch.add(s, t)
tensor([[3., 1.],
        [1., 3.]])
```

Note that our implementation of `add` does not take `alpha` or `out` as keyword arguments like :func:`torch.add` does:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 502); *backlink*

Unknown interpreted text role "func".

```
>>> torch.add(s, s, alpha=2)
TypeError: add() got an unexpected keyword argument 'alpha'
```

For speed and flexibility the `__torch_function__` dispatch mechanism does not check that the signature of an override function matches the signature of the function being overrided in the :mod:`torch` API. For some applications ignoring optional arguments would be fine but to ensure full compatibility with :class:`Tensor`, user implementations of torch API functions should take care to exactly emulate the API of the function that is being overrided.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 508); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 508); *backlink*

Unknown interpreted text role "class".

Functions in the :mod:`torch` API that do not have explicit overrides will return `NotImplemented` from `__torch_function__`. If all operands with `__torch_function__` defined on them return `NotImplemented`, PyTorch will raise a `TypeError`. This means that most of the time operations that do not have explicit overrides for a type will raise a `TypeError` when an instance of such a type is passed:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 515); *backlink*

Unknown interpreted text role "mod".

```
>>> torch.mul(s, 3)
TypeError: no implementation found for 'torch.mul' on types that
implement __torch_function__: [ScalarTensor]
```

In practice this means that if you would like to implement your overrides using a `__torch_function__` implementation along these lines, you will need to explicitly implement the full :mod:`torch` API or the entire subset of the API that you care about for your use case. This may be a tall order as the full :mod:`torch` API is quite extensive.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 526); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 526); *backlink*

Unknown interpreted text role "mod".

Another option is to not return `NotImplemented` for operations that are not handled but to instead pass a :class:`Tensor` to the original :mod:`torch` function when no override is available. For example, if we change our implementation of `__torch_function__` for `ScalarTensor` to the one below:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 532); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 532); *backlink*
>
> Unknown interpreted text role "mod".

```
@classmethod
def __torch_function__(cls, func, types, args=(), kwargs=None):
    if kwargs is None:
        kwargs = {}
    if func not in HANDLED_FUNCTIONS or not all(
            issubclass(t, (torch.Tensor, ScalarTensor))
            for t in types
        ):
        args = [a.tensor() if hasattr(a, 'tensor') else a for a in args]
        return func(*args, **kwargs)
    return HANDLED_FUNCTIONS[func](*args, **kwargs)
```

Then :func:`torch.mul` will work correctly, although the return type will always be a :class:`Tensor` rather than a :class:`ScalarTensor`, even if both operands are :class:`ScalarTensor` instances:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 549); *backlink*
>
> Unknown interpreted text role "func".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 549); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 549); *backlink*
>
> Unknown interpreted text role "class".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 549); *backlink*
>
> Unknown interpreted text role "class".

```
>>> s = ScalarTensor(2, 2)
>>> torch.mul(s, s)
tensor([[4., 0.],
        [0., 4.]])
```

Also see the `MetadataTensor` example below for another variation on this pattern but instead always returns a `MetadataTensor` to propagate metadata through operations in the :mod:`torch` API.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, line 558); *backlink*
>
> Unknown interpreted text role "mod".

The `__torch_function__` protocol is designed for full coverage of the API, partial coverage may lead to undesirable results, in particular, certain functions raising a `TypeError`. This is especially true for subclasses, where all three of *torch.add*, *torch.Tensor.__add__* and *torch.Tensor.add* must be covered, even if they return exactly the same result. Failing to do this may also lead to infinite recursion. If one requires the implementation of a function from `torch.Tensor` subclasses, they must use `super().__torch_function__` inside their implementation.

Subclassing `torch.Tensor`

As of version 1.7.0, methods on `torch.Tensor` and functions in public `torch.*` namespaces applied on `torch.Tensor` subclasses will return subclass instances instead of `torch.Tensor` instances:

```
>>> class SubTensor(torch.Tensor):
```

```
...     pass
>>> type(torch.add(SubTensor([0]), SubTensor([1]))).__name__
'SubTensor'
>>> type(torch.add(SubTensor([0]), torch.tensor([1]))).__name__
'SubTensor'
```

If multiple subclasses exist, the lowest one in the hierarchy will be chosen by default. If there is no unique way to determine such a case, then a `TypeError` is raised:

```
>>> type(torch.add(SubTensor2([0]), SubTensor([1]))).__name__
'SubTensor2'
>>> type(torch.add(SubTensor2([0]), torch.tensor([1]))).__name__
'SubTensor2'
>>> torch.add(SubTensor([0]), OtherSubTensor([1]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: no implementation found for 'torch.add' on types that implement __torch_function__: [SubTensor, OtherSu
```

If one wishes to have a global override for all tensor methods, one can use `__torch_function__`. Here is an example that logs all function/method calls:

```
class LoggingTensor(torch.Tensor):
    @classmethod
    def __torch_function__(cls, func, types, args=(), kwargs=None):
        # NOTE: Logging calls Tensor.__repr__, so we can't log __repr__ without infinite recursion
        if func is not torch.Tensor.__repr__:
            logging.info(f"func: {func.__name__}, args: {args!r}, kwargs: {kwargs!r}")
        if kwargs is None:
            kwargs = {}
        return super().__torch_function__(func, types, args, kwargs)
```

However, if one instead wishes to override a method on the Tensor subclass, there one can do so either by directly overriding the method (by defining it for a subclass), or by using `__torch_function__` and matching with `func`.

One should be careful within `__torch_function__` for subclasses to always call `super().__torch_function__(func, ...)` instead of `func` directly, as was the case before version 1.7.0. Failing to do this may cause `func` to recurse back into `__torch_function__` and therefore cause infinite recursion.

Extending :mod:`torch` with a :class:`Tensor` wrapper type

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 623); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 623); *backlink*

Unknown interpreted text role "class".

Another useful case is a type that wraps a :class:`Tensor`, either as an attribute or via subclassing. Below we implement a special case of this sort of type, a `MetadataTensor` that attaches a dictionary of metadata to a :class:`Tensor` that is propagated through :mod:`torch` operations. Since this is a generic sort of wrapping for the full :mod:`torch` API, we do not need to individually implement each override so we can make the `__torch_function__` implementation more permissive about what operations are allowed:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 626); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 626); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 626); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 626); *backlink*

Unknown interpreted text role "mod".

```
class MetadataTensor(object):
    def __init__(self, data, metadata=None, **kwargs):
```

```
            self._t = torch.as_tensor(data, **kwargs)
            self._metadata = metadata

        def __repr__(self):
            return "Metadata:\n{}\n\ndata:\n{}".format(self._metadata, self._t)

        @classmethod
        def __torch_function__(cls, func, types, args=(), kwargs=None):
            if kwargs is None:
                kwargs = {}
            args = [a._t if hasattr(a, '_t') else a for a in args]
            metadatas = tuple(a._metadata if hasattr(a, '_metadata') for a in args)
            assert len(metadatas) > 0
            ret = func(*args, **kwargs)
            return MetadataTensor(ret, metadata=metadatas[0])
```

This simple implementation won't necessarily work with every function in the :mod:`torch` API but it is good enough to capture most common operations:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst, line 652); *backlink***
>
> Unknown interpreted text role "mod".

```
>>> metadata = {'owner': 'Ministry of Silly Walks'}
>>> m = MetadataTensor([[1, 2], [3, 4]], metadata=metadata)
>>> t = torch.tensor([[1, 2], [1, 2]])
>>> torch.add(t, m)
Metadata:
{'owner': 'Ministry of Silly Walks'}

data:
tensor([[2, 4],
        [4, 6]])
>>> torch.mul(t, m)
Metadata:
{'owner': 'Ministry of Silly Walks'}

data:
tensor([[1, 4],
        [3, 8]])
```

Operations on multiple types that define `__torch_function__`

It is possible to use the torch API with multiple distinct types that each have a `__torch_function__` implementation, but special care must be taken. In such a case the rules are:

- The dispatch operation gathers all distinct implementations of `__torch_function__` for each operand and calls them in order: subclasses before superclasses, and otherwise left to right in the operator expression.
- If any value other than `NotImplemented` is returned, that value is returned as the result. Implementations can register that they do not implement an operation by returning `NotImplemented`.
- If all of the `__torch_function__` implementations return `NotImplemented`, PyTorch raises a `TypeError`.

Testing Coverage of Overrides for the PyTorch API

One troublesome aspect of implementing `__torch_function__` is that if some operations do and others do not have overrides, users will at best see an inconsistent experience, or at worst will see errors raised at runtime when they use a function that does not have an override. To ease this process, PyTorch provides a developer-facing API for ensuring full support for `__torch_function__` overrides. This API is private and may be subject to changes without warning in the future.

First, to get a listing of all overridable functions, use `torch.overrides._get_overridable_functions`. This returns a dictionary whose keys are namespaces in the `PyTorch` Python API and whose values are a list of functions in that namespace that can be overriden. For example, let's print the names of the first 5 functions in `torch.nn.functional` that can be overriden:

```
>>> from torch.overrides import get_overridable_functions
>>> func_dict = get_overridable_functions()
>>> nn_funcs = func_dict[torch.nn.functional]
>>> print([f.__name__ for f in nn_funcs[:5]])
['adaptive_avg_pool1d', 'adaptive_avg_pool2d', 'adaptive_avg_pool3d',
 'adaptive_max_pool1d', 'adaptive_max_pool1d_with_indices']
```

This listing of functions makes it possible to iterate over all overridable functions, however in practice this is not enough to write tests for all of these functions without laboriously and manually copying the signature of each function for each test. To ease this process, the `torch.overrides._get_testing_overrides` function returns a dictionary mapping overridable functions in the `PyTorch` API to dummy lambda functions that have the same signature as the original function but unconditionally return -1. These functions are most useful to use with `inspect` to analyze the function signature of the original `PyTorch` function:

```
>>> import inspect
>>> from torch.overrides import get_testing_overrides
>>> override_dict = get_testing_overrides()
>>> dummy_add = override_dict[torch.add]
>>> inspect.signature(dummy_add)
<Signature (input, other, out=None)>
```

Finally, `torch.overrides.get_ignored_functions` returns a tuple of functions that explicitly cannot be overrided by `__torch_function__`. This list can be useful to confirm that a function that isn't present in the dictionary returned by `get_overridable_functions` cannot be overriden.

Writing custom C++ extensions

See this PyTorch tutorial for a detailed explanation and examples.

Documentations are available at :doc:`../cpp_extension`.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]extending.rst`, **line 744**); ***backlink***
>
> Unknown interpreted text role "doc".

Writing custom C extensions

Example available at this GitHub repository.