Intro

Testing is an important part of developing PyTorch. Good tests ensure both that:

- operations in PyTorch correctly implement their behavior and that
- changes to PyTorch do not change accidentally change these behaviors.

This article explains PyTorch's existing testing tools to help you write effective and efficient tests.

If you're interested in helping to improve PyTorch's tests then see our tracking issue here.

Test file organization

PyTorch's test suites are located under pytorch/test.

Code for generating tests and testing helper functions are located under pytorch/torch/testing/_internal.

Running PyTorch's test suites

Most PyTorch test suites can be run using unittest (the default) or pytest.

To run a test suite, like test_torch.py, using unittest:

```
python test_torch.py
```

Unittest-specific arguments can be appended to this command. For example, to run only a specific test:

```
python test_torch.py <TestClass>.<TestName>
```

Other commonly useful options are `-k`, which specifies a string to filter the tests, and `-v`, which runs the test suite in "verbose" mode. Unfortunately `-k` works only with `pytest`. For example,

```
pytest test_torch.py -k cpu
```

Will run all the tests in test_torch.py with "cpu" in their name.

By default, the test suite will not run tests marked with the `@slowTest` decorator. To run the "slow tests" set the `PYTORCH_TEST_WITH_SLOW` to one like so:

```
PYTORCH_TEST_WITH_SLOW=1 python test_torch.py
```

The device-generic test framework

PyTorch extends Python test frameworks like unittest and pytest with its own test framework referred to as the "device-generic test framework." The device-generic test framework instantiates variants of test templates with new arguments, making it easier to test multiple device types, dtypes, and operations. It is defined in common_device_type.py.

"Instantiating," or, equivalently, "generating" tests programmatically has pros and cons. A pro is that it makes writing tests simpler. A con is that it's harder to understand what tests actually cover and where generated tests come from. The next section will elaborate more on PyTorch's test coverage.

Before we look at the device-generic test framework, note that using it is not a requirement for all tests. Tests in PyTorch not using the framework are simply `TestCase` class methods whose names start with "test" and that accept only `self` as an argument, like this:

```
# a simple test
def test_foo(self):
  ...
```

Many of PyTorch's test classes use the device-generic framework, however. These classes and the tests written in them are actually templates. The simplest version of these test templates just accept an additional device argument:

```
class TestFoo(TestCase):
  # a device generic test
  # the device is an argument
  def test_foo(self, device):
    ...
```

When this test class is run it will typically generate a version of this template for each available device type. For example, on a machine with a CUDA device this template will create two two classes: `TestFooCPU` and `TestFooCUDA`, with tests `test_foo_cpu` and `test_foo_cuda`, respectively. The `device` argument passed to `test_foo_cpu` will be "cpu", and the `device` argument passed to `test_foo_cuda` will be a string representing a CUDA device like "cuda:0" or "cuda:1". To elaborate, we can think of the template being translated at runtime to:

```
class TestFooCPU(TestCase):
  def test_coo_cpu(self, device='cpu'):
    ...

class TestFooCUDA(TestCase):
  def test_foo_cuda(self, device='cuda'):
    ...
```

These tests can be run directly with a command like `python test_foo.py TestFooCPU.test_foo_cpu`. Even though these tests are generated they will also work with the `-k` flag, so `pytest test_foo.py -k test_foo` will run both `test_foo_cpu` and `test_foo_cuda`.

Tensors created in these tests should typically be placed on the device, like this:

```
torch.randn(shape, device=device)
```

And if the test has device-specific behavior it can query for its device type using:

```
# acquire the device type
torch.device(device).type
```

Tests should not assume, however, that the only possible device types are the CPU or CUDA. XLA, for example, is another device type that PyTorch's CI includes.

The device-generic framework also offers a variety of decorators to control how templates are instantiated. Some of these decorators are:

```
@onlyOnCPUAndCUDA   # skips variants that do not use the CPU or CUDA device type
@onlyCPU            # skips non-CPU variants for this template
@onlyCUDA           # skips non-CUDA variants for this template
@skipCPUIfNoLapack  # Skips CPU variants of the test if LAPACK is not installed
@skipCPUIfNoMkl     # Skips CPU variants of the test if MKL is not installed
@skipCUDAIfNoMagma  # Skips CUDA variants of the test if MAGMA is not installed
@skipCUDAIfRocm     # Skips CUDA variants of the test if ROCm is being used
```

Two additional decorators will cause the device-generic framework to instantiate additional versions of the test. Each of these decorators requires a modification of the test template's signature. This section describes the `@dtypes` decorator, and the next section describes the `@ops` decorator.

The `@dtypes` decorator tells the device-generic framework to instantiate variants of the test template for each of the dtypes, or iterables containing dtypes, passed to the decorator. These arguments become the third input to the test. For example:

```
@dtypes(torch.float, torch.double)
def test_foo(self, device, dtype):
  ...
```

Would instantiate the following tests on a machine with a CUDA device:

```
test_foo_cpu_float32
test_foo_cpu_float64
test_foo_cuda_float32
test_foo_cuda_float64
```

All of the device-generic framework decorators are composable, so a template like this:

```
@onlyCPU
@dtypes(torch.float, torch.double)
def test_foo(self, device, dtype):
  ...
```

Would only instantiate two tests:

```
test_foo_cpu_float32
test_foo_cpu_float64
```

Tensors created in these tests should typically be placed on the given device and use the given dtype, like so:

```
torch.randn(shape, device=device, dtype=dtype)
```

Using the device-generic test framework is essential when testing new tensor operations, and becoming familiar with it will help you understand how PyTorch's tests are generated. The latest addition to the device-generic test framework, OpInfos, is described in the next section.

OpInfos and the future of testing tensor operations

The OpInfo pattern, described in this section, is the future of testing tensor operations in PyTorch. This pattern is intended to make testing tensor operations simpler, since tensor operations have so much in common that writing a test from scratch for each operation would be redundant and exhausting. Instead, OpInfos contain metadata that test templates can use to test properties of many operators at once.

The OpInfo class and every OpInfo is defined in common_methods_invocations.py. The OpInfo class has tensor operation-related metadata like

- which dtypes the operator supports
- whether it supports inplace gradients or not.

One of the most important properties of each OpInfo is its `sample_inputs()` method, used to acquire "sample inputs" to the function with different properties.

In addition to the OpInfo base class, there are several derived classes with additional structure. For example, UnaryUfuncInfo is a specialization of OpInfos for unary universal functions. The details of each of these classes is beyond the scope of this section. See the documentation in common_methods_invocations.py for details.

Let's look at an example OpInfo:

```
UnaryUfuncInfo(
  'asinh',
  ref=np.arcsinh,
  dtypes=all_types_and(torch.bool),
  dtypesIfCPU=all_types_and(torch.bool),
  dtypesIfCUDA=all_types_and(torch.bool, torch.half, torch.bfloat16),
  promotes_integers_to_float=True,
  decorators=(precisionOverride({torch.bfloat16: 5e-2}),),
  test_inplace_grad=False,
  skips=(
    # RuntimeError: "rsqrt_cuda" not implemented for 'BFloat16'
    SkipInfo('TestCommon', 'test_variant_consistency_jit',
             device_type='cuda', dtypes=[torch.bfloat16])
```

This OpInfo, or more precisely this UnaryUfuncInfo, is for torch.asinh(). It has metadata, like that NumPy's `np.arcsinh` is a "reference function" with comparable behavior, and that `torch.asinh()` promotes integer inputs to floating point inputs. It also specifies which dtypes it supports on both CPU and CUDA, and it defines a `SkipInfo` that tells the device generic framework to not instantiate the `test_variant_consistency_jit` template for this operator with the CUDA device type and the bfloat16 dtype.

This is a lot to review at once. In practice writing an OpInfo is typically an iterative process where a simple OpInfo is written, and then the test suites will identify issues with the operator's specification and suggest changes. For example, if an OpInfo incorrectly states that an operator supports a dtype, like `torch.half`, then a test will fail and point out that the operator does not, in fact, support that dtype. The OpInfo can then be updated using this information. If you need help implementing an OpInfo then file an issue.

OpInfos are used in two ways. First, test_ops.py tests common properties of every tensor operation. Second, tests like test_unary_ufuncs.py can use classes derived from OpInfo to create exemplar Tensors to test properties of a subset of the tensor operations. Understanding what test_ops.py tests is important when implementing a new tensor operator or updating an existing one so you know what additional tests, if any, are needed. Knowing how to write a test template that consumes OpInfos is interesting, but only directly useful if you plan to write a test template that runs on multiple tensor operators.

The `@ops` decorator

test_ops.py has tests that use the `@ops` decorator. Like the `@dtypes` decorator, the `@ops` decorator is part of the device-generic test framework and instantiates variants of test templates based on an iterable of `OpInfo`s. For example:

```
@ops(op_db)
def test_foo(self, device, dtype, op):
  ...
```

This will instantiate a variant of test_foo for every OpInfo (all OpInfos are included in the `op_db` iterable), for every dtype that OpInfo's operator supports, and on each device type. As of this writing this is hundreds of tests. Test templates designed to work on every tensor operation are inherently restricted to testing fundamental properties of these operators. For example, that their derivatives are properly implemented. In particular, test_ops.py tests the following:

- that the operation's dtypes are correctly enumerated
- that the operation's function, method, and inplace variants are equivalent
- that the operation's eager behavior is the same as its jitted behavior
- that the operation's derivative and second derivative are properly implemented

Notably it DOES NOT test:

- that the operation is implemented properly

And this may seem odd. How can, for example, test_ops.py verify that the operation's derivative and second derivative are implemented properly but not that the operation itself is!? It's because the tests for the derivative and second derivative assume the operation's "forward" is correctly implemented, but test_ops.py has no way of knowing if that's true. In the future it may be expanded with additional support for verifying that a function's "forward" behavior is correct, too.

This means that, typically, when implementing a new tensor operation you will need to write an OpInfo and one or more device-generic test templates for it. These templates should verify the operation's forward is properly implemented, but they do not need to be redundant with test_ops.py. In particular, they usually don't need to test the operation's jitted behavior or that its derivative is properly implemented. Some derived classes of OpInfo, like the UnaryUfuncInfos, actually do have the structure to test the forward behavior of their operators, however. See test_unary_ufuncs.py for details on how the UnaryUfuncInfos work.

NN Module Testing

TODO

Deprecated Test Patterns (Do Not Use)

Please do not use older methods such as:

1. `use_cuda` variants:

```
    def _test_random_neg_values(self, use_cuda=False):
        signed_types = ['torch.DoubleTensor', 'torch.FloatTensor',
'torch.LongTensor',
                        'torch.IntTensor', 'torch.ShortTensor']
        for tname in signed_types:
```

```
            res = torch.rand(SIZE, SIZE).type(tname)
            if use_cuda:
                res = res.cuda()
            res.random_(-10, -1)
            self.assertLessEqual(res.max().item(), 9)
            self.assertGreaterEqual(res.min().item(), -10)

    def test_random_neg_values(self):
        self._test_random_neg_values(self)
```

As it requires 3 function declarations and weird `if` statements.

2. `get_all_device_types` variants:

```
    def test_add(self):
        for device in torch.testing.get_all_device_types():
            # [res] torch.add([res,] tensor1, tensor2)
            m1 = torch.randn(100, 100, device=device)
            v1 = torch.randn(100, device=device)
# ......
```

Generally good, but when it fails, you spent time to figure out if fail is CPU error or CUDA error. Also is impossible to filter tests by the name.

3. `_cuda` and `_cpu` variants:

```


    def test_all_any_empty(self):
        x = torch.ByteTensor()
# .....

    @unittest.skipIf(not torch.cuda.is_available(), 'no CUDA')
    def test_all_any_empty_cuda(self):
# .......
```

This solution has lots of code duplication and is prone to errors.

4. Casting lambdas variants:

```
def _test_dim_reduction(self, cast):
    # bla bla bla
    return

def test_dim_reduction(self):
    self._test_dim_reduction(self, lambda t: t)
```

Casting is slower, also it is hard to read such code.

5. Decorator `@torchtest.test_all_device_types()`, which would only instantiate CPU and CUDA variants of a test.