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.

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 tests

Directly running python test files

Most PyTorch test files 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. For example,

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

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

Running via `test/run_test.py`

In addition to directly running python test files. PyTorch's Continuous Integeration and some specialized test cases can be launched via pytorch/test/run_test.py.

It provides some additional features that normally doesn't exist when directly running python test files. For example:

1. Run multiple python test files together via selective run.
2. Run distributed test and cpp_extension tests via custom test handler.
3. Perform other CI related optimizations such as target determination based on changed files, automatic sharding, etc. See What is CI testing and When section for more details.

One can directly run the `test/run_test.py` file and it will selectively run all tests available in your current platform:

```
python test/run_test.py
```

Alternatively you can pass in additional arguments to run specific test(s), use the help function to find out all possible test options.

```
python test/run_test.py -h
```

Using environment variables:

In addition to unittest and pytest options, PyTorch's test suite also understands the following environment variables:

- PYTORCH_TEST_WITH_SLOW, if set to 1 this will run tests marked with the @slowTest decorator (default=0)
- PYTORCH_TEST_SKIP_FAST, if set to 1 this will skip tests NOT marked with the @slowtest decorator (default=0)
- PYTORCH_TEST_WITH_SLOW_GRADCHECK, if set to ON this use PyTorch's slower (but more accurate) gradcheck mode (default=OFF)
- PYTORCH_TESTING_DEVICE_ONLY_FOR, run tests for ONLY the device types listed here (like 'cpu' and 'cuda')
- PYTORCH_TESTING_DEVICE_EXCEPT_FOR, run tests for all device types EXCEPT FOR the device types listed here
- PYTORCH_TEST_SKIP_NOARCH, if set to 1 this will all noarch tests (default=0)

For instance,

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

will run the tests in test_torch.py, including those decorated with `@slowTest`.

Using Github label to control CI behavior on PR

(last updated 2021-08-13)

PyTorch runs different sets of jobs on PR vs. on master commits.

In order to control the behavior of CI jobs on PR. We support Github Labels to control what to test on CI on PRs:

- `ci/master` : runs sets of CI jobs that normally only run on master
- `ci/binaries` : runs sets of CI jobs that builds additional binaries (normally only run on nightly)
- `ci/slow-gradcheck` : runs the slow gradcheck build

If this list becomes out of date, the current definitions should be able to be found in `.github/pytorch-circleci-labels.yml`.

Note that `ci/master` and `ci/binaries` only work for jobs running in CircleCI. We're migrating CI jobs to GitHub Actions, the new proposal of how to dynamically trigger workflows in GitHub Actions can be tracked here.

Common test utilities

Use the test case's assertEqual to compare objects for equality.

Prefer using make_tensor when generating test tensors over tensor creation ops like *torch.randn*.

PyTorch's test generation functionality

See this comment for details on writing test templates.

PyTorch's test framework lets you instantiate test templates for different operators, datatypes (dtypes), and devices to improve test coverage. It is recommended that all tests be written as templates, whether it's necessary or not, to make it easier for the test framework to inspect the test's properties.

In general, there exist three variants of instantiated tests, which adapt the names at runtime according the following scheme.

- Tests are parametrized with multiple devices: `<TestClass><DEVICE>.<test_name>_<device>`
- Tests are additionally parametrized with multiple dtypes: `<TestClass><DEVICE>.<test_name>_<device>_<dtype>`
- Test are additionally parametrized with multiple operators: `<TestClass><DEVICE>.<test_name>_<operator_name>_<device>_<dtype>`

To use the selection syntax to run only a single test class or test, be it with `unittest` or `pytest`, it is important to use instantiated name rather than the template name. For `pytest` users there is the `pytest-pytorch` plugin, that re-enables selecting individual test classes or tests by their template name.

OpInfos

See the "OpInfos" note in common_methods_invocations.py for details on adding an OpInfo and how they work.

OpInfos are used to automatically generate a variety of operator tests from metadata. If you're adding a new operator to the torch, torch.nn, torch.special, torch.fft, or torch.linalg namespaces you should write an OpInfo for it so it's tested properly.