Some of PyTorch's operations use nondeterministic algorithms that can produce nondeterministic results. However, some PyTorch users want reproducibility, so we've provided `torch.set_deterministic()`. This guide will show how to add support for this setting in any PyTorch operation that is nondeterministic by default.

Note: You can read about more of the user-facing reproducibility APIs here.

Basics

If `torch.set_deterministic(True)` is called, it sets a global flag that is accessible from the C++ `at` namespace. Any PyTorch operation that is nondeterministic by default should use one of the two following options if it is called while this flag is turned on:

Option 1: Call an alternate deterministic implementation

This is the ideal case. The operation should use `at::globalContext().deterministic()` to check if the deterministic flag is set, and then execute the appropriate implementation.

Example:

```
Tensor some_operator(const Tensor& t) {
    if (at::globalContext().deterministic()) {
        return some_operator_impl_deterministic(t);
    } else {
        return some_operator_impl_nondeterministic(t);
    }
}
```

This guide does not describe how to write a deterministic implementation, since nondeterminism can arise for a number of reasons.

Option 2: Throw an error

Sometimes a deterministic implementation is not possible or just not written yet. In this case, the operation can call `at::globalContext().alertNotDeterministic(<char*: name of operation>)`. This will check if the deterministic flag is turned on and then throw a RuntimeError if so. Please be sure to write a comment explaining briefly why the operation is nondeterministic.

Example:

```
Tensor some_operator(const Tensor& t) {
    // This is nondeterministic because <reason>
    at::globalContext().alertNotDeterministic("some_operator");
```

```
    ...
}
```

Unit Tests

When a deterministic implementation is added to an operator, it would be nice to be able test it to make sure it is, in fact, deterministic. But given the probabilistic nature of nondeterminism, it is not always feasible to inductively test if a given operation is deterministic. However if an operation does not have a deterministic implementation and it throws an error instead, testing the error is usually simple and recommended.

`torch.testing._internal.common_device_type.expectedAlertNondeterministic` (soon to be added in PR #41538) is a decorator used to automatically turn on the deterministic flag and check that a given function raises an error with `at::Context::alertNotDeterministic()`. Here is a basic example of a test in `test/test_torch.py` using this decorator:

```
from torch.testing._internal.common_device_type import expectedAlertNondeterministic

@expectedAlertNondeterministic('some_operator')
def test_some_operator_alert_nondeterministic(self, device):
    torch.some_operator(...)
```

`expectedAlertNondeterministic` can also be used within `torch/testing/_internal/common_methods_invo` and `torch/testing/_internal/common_nn.py`. You can look at examples introduced by PR #41538.

Currently unsupported functions

Right now, some PyTorch operators are nondeterministic and they do not raise an error. They should be fixed, and this section can be used to track them. If you know of anything that should be added to this list, please feel free to add it or let us know that we should add it.

- ☐ CUDA operations that use `gpuAtomicAdd` (PR #41538)
- ☐ CuBLAS operations if CUDA >= 10.2 and `CUBLAS_WORKSPACE_CONFIG` is not set appropriately (PR #41377)