The purpose of this document is to describe PyTorch's "code review values"; the things that we value in our codebase and would like reviewers to check for and authors to address.

About diff presentation

- **Does the diff description describe *why* the change is being made?** It might be obvious now; it won't be obvious in six months.
- **For larger diffs, does the diff description summarize what changes were made?** Ideally, every distinct change comes in a different diff, but sometimes this cannot be done conveniently. In that case, demand that the diff author list out the distinct changes made in the diff—this is as easy as just asking the diff author to read their diff and write down bullet points of changes they made.
- **Does the diff description accurately describe the contents of the diff?** The most common mistake is that the diff description says the diff does X, but inside the diff secretly also does Y. This is bad; all of the changes should be accurately reflected in the description.
- **Is the diff easy to review?** Authors of diffs have an obligation to make their diffs easily reviewable. If it is very hard to review a diff (for example, a file was moved and modified, and the diff view is not showing the incremental changes anymore), a reviewer is entitled to tell the author to reorganize the diff into smaller diffs to make it easier to review.

About engineering

- **Does the change maintain backwards compatibility of the public Python API?** BC-breaking changes to the Python API must be done with extreme care, because they break user code without any static indication that things are broken (a user only finds out when they try to actually run their code).
- **Could the change be more generally applicable elsewhere in the codebase?** Suppose a pull request fixes a bug in replication_pad1d. That bug may also exist in replication_pad2d/replication_pad3d; a reviewer should point this out and encourage the diff author to check if the problem shows up in other places.
- **Does the change use idiomatic, existing functionality in the codebase?** Newer authors may reimplement functionality that already exists in the codebase, because they don't know about the existing implementation. A reviewer can point out when this has happened and help prevent us from getting multiple copies of the same thing in the code. For example, does the author manually call cudaSetDevice, or do they use CUDADeviceGuard? Does the author call assert(), or AT_ASSERT()?
- **Is a change or comment likely to break in the future?** What you are trying to find is code that will become broken at some point in the future, because of an unrelated change elsewhere in the codebase. The

most common reason is a change relies on some unstated assumption or invariant in the code, which may be broken in the future. Be especially aware for changes which directly contradict planned future work.

- **Does it make a change for no good reason?** Every change to PyTorch should be *intentional*: there should be some underlying reason or motivation for the change. A change "just because I was randomly trying things until it worked" is not a good reason.
- **Is the feature worth it?** Every line of code we add to PyTorch means more code size and more maintenance overhead in the future. Every new feature's benefit must be weighed against its ongoing maintenance cost. Features which require complicated and delicate interactions with the rest of the system may not be worth it.
- **Is the change "fiddly?"** For example, does it introduce some new manual refcounting code? Is it introducing some new code using atomics or locks? The review bar for these types of code (among others) is higher, and for high risk code like this, it should be reviewed much more carefully, since often there is no way to efficiently test for bugs in these cases.

About documentation

- **Is there documentation?** Suppose a new feature was added in the pull request. Is there a docblock for the newly added function explaining what it does? Is it actually comprehensive documentation (similar in style and depth to our existing documentation), and not just a single sentence or two?
- **Are there comments explaining non-obvious motivation or backstory?** Did you have an epic debugging session in the PR resulting in a single line change in the diff? Comment it. Did you add a hack that is intended to be temporary and unblock some use case? Comment it. Is the reason for the existence of a new class or struct non-obvious? Comment it. In PyTorch, we have a convention that for bigger comments, they can be written out of line and titled with Note [Blah blah blah], and then referenced at appropriate locations in code using "See Note [Blah blah blah]"
- **Were relevant comments updated?** It's easy for an author to make a change, and not notice that a nearby comment also needs updating. When reviewing, it's good to also check the nearby comments are still applicable even after the changes.

About testing

- **Are there tests?** Tests should come in the same diff as the functionality that is added, as protection from the author forgetting to add them later. Tests are extra important for Python code, since otherwise there is there is no compile-time checked type system to automatically flush out what call sites are now wrong. If the diff fixes a bug, is there a test that exercises

the bug: failed before and no longer fails? If the bug is not easy to test, did the diff author describe how they tested the change by hand?

- **Are the tests comprehensive?** Suppose you added an operator that takes some optional arguments. Do your tests test each of the optional arguments, individually and all together? If an argument takes either a number or a tuple, do you test with a tuple with differing values for its elements? Do you test it on both contiguous and non-contiguous inputs? (The standard test harness checks for some of these things, but at the end of the day it's jointly the responsibility of the author and the code reviewer to make sure that the harness in conjunction with the written tests are sufficient.)
- **For hard to test changes, did the author describe how they tested their change?** Suppose a diff was submitted to improve support for some GPU on OS X situation. We don't have this configuration in our continuous integration, and thus cannot conveniently test the change. In that case, the author should give some test plan / evidence that their change in fact works.
- **If the change is a performance optimization, did the diff author include a benchmark?** A simple timeit invocation showing change in wall clock improvement is much better than nothing at all. If you are optimizing, you have an obligation to show that your optimization also showed an improvement.
- **Did the diff pass builds and lint?** In general, the diff author has the responsibility for ensuring all CI is passing before landing, but as tests can be occasionally flaky, it is helpful to lend a hand to newer authors by explicitly mentioning when CI failures look legitimate or not.

About numerical computing

- **Is the kernel bit-for-bit deterministic?** Bit-for-bit determinism is an extremely desirable property for debugging purposes. However, not all algorithms are deterministic in this way; for example, because floating-point addition is not associative, algorithms that add subresults in non-deterministic order will lead to non-deterministic results. Code review of a new kernel is the best time to identify if it is deterministic or not; it is much harder to make this determination when you are attempting to debug a nondeterminism bug in a larger program.
- **Are operations that only need to be done once done outside of a critical loop?** Dynamic dispatch, vector allocation, etc.; these are all expensive operations which should be done outside of tight loops over tensor dimensions (e.g., batch dimension). In ordinary code, this doesn't matter, but in numerical kernels, failing to move these operations outside of the loop can noticeably affect performance.
- **Does it work on data that is bigger than 4GB?** The most common reason things break in this situation is because someone used a 32-bit integer where they should have used a 64-bit integer. These are hard to

test for, since actually constructing a 4GB input is expensive and slows tests down a lot. The most common reason for breakage here is use of "int" when you should have used "int64_t".

- **Do calls to the kernel call cudaGetLastError() after the kernel launch?** This best practice helps us report asynchronous CUDA errors closer to the error site, rather than at some much later point in time.