Summary

This page describes PyTorch's Python Frontend backwards and forward compatibility policy, which is in effect starting with PyTorch 1.12. This policy lets us provide a modern user experience while minimizing disruption to PyTorch's users and developers.

Backwards compatibility (BC)

A version of PyTorch is "backwards compatible (BC)" with previous versions if it can run the same programs those older versions can run. This is an important property, because without it users would have to create new PyTorch programs with every release. In practice, PyTorch may not be completely backwards compatible with previous versions, but there should be few, clearly documented incompatibilities that are easy to address for users and developers between each version. These incompatibilities are referred to as "BC-breaking changes."

Why BC-breaking changes happen

"BC-breaking changes" stop some older PyTorch programs from running on new PyTorch versions, and this can be disruptive to users and developers. These changes are, however, necessary to give PyTorch the best user experience possible, replacing broken, slow, confusing, insufficient or otherwise outdated functionality with modern improvements. For example, PyTorch used to have a torch.fft function, but that has been replaced with the torch.fft module, which offers an easier to use interface and more functionality.

Minimizing the disruption of BC-breaking changes

PyTorch makes a best effort to minimize the disruption of BC-breaking changes to stable components (see below for a discussion of stability) by providing ample notice of the change, how to handle it, and time to adapt.

- A warning should be thrown once per process (using `TORCH_WARN_ONCE`) when the deprecated behavior is used.
  - The warning should clearly describe the deprecated behavior and why it is deprecated.
  - The warning should describe when the PyTorch release where the deprecation first takes effect.
  - If possible, the warning should provide a straightforward alternative to the deprecated behavior.
    - Note that alternative behaviors should be stable (more on stability below). And it may be wise to delay a deprecation until a stable alternative is available.
- The warning should give ample time for users of both released PyTorch version and PyTorch nightlies, so it should appear for two PyTorch releases

and 180 days of PyTorch nightlies.
- If switching to the new behavior would cause a "silent breakage" (see below for a discussion), then the old behavior should throw a runtime error for another two PyTorch releases and 180 days of PyTorch nightlies.
- The documentation should be updated with a warning that highlights the current status of the deprecation. It should include stable alternatives, if possible. The deprecation should use the `.. deprecated:: <version>` Sphinx directive. (See this issue for how this directive renders.)
- After the deprecation takes effect, the documentation should be updated with a note that describes what changes have occurred and which releases they occurred in. The change notice should use the `.. versionchanged:: <version>` Sphinx directive.

Note that it is not always possible to make BC-breaking changes this way. Sometimes, for example, changes have to be reverted because they have unintended consequences, and these reverts can cause BC-breaking changes themselves. Other times the current behavior is so unsafe it should not be used at all and there is no clear alternative to it. Still, all BC-breaking changes should do their best to comport with the above.

This policy means that PyTorch programs relying on stable components and running without deprecation warnings are likely to continue running as expected for at least two more PyTorch releases (180 days for nightly users). And when BC-breaking changes would cause silent breakages this policy provides a window of 4 PyTorch releases (360 days for nightly users). By providing ample warning time, clear alternatives, and prominent documentation we let users update their programs with a minimum of fuss and facilitate developers supporting multiple PyTorch versions. This policy will never handle every case, unfortunately, and some users may have to continue using older PyTorch versions for some programs, but it should minimize those instances.

What isn't a BC-breaking change

BC-breaking changes prevent old programs from running as expected on newer PyTorch versions. Generally, however, the following are not BC-breaking changes:

- bug fixes (where PyTorch's behavior changes to better reflect its documentation)
- small numerics changes
- numerical accuracy improvements
- changes to PyTorch's internals
- changes to module and operator signatures that don't impact users

This is especially important for developers building on PyTorch to understand. For example, adding a new keyword-only argument with a default value to an operator or module is not a BC-breaking change because PyTorch programs will continue running as before. Systems relying on PyTorch's operators and

modules should account for this possibility and not depend on the precise form of PyTorch's internals.

Prototypes, beta, and stable parts of PyTorch

Different parts of PyTorch are considered to be in a "prototype," "beta," or "stable" state. See this blog post for details about these designations. Importantly, BC-breaking changes to prototype and beta components can be made without warning or following any of the above processes. This is necessary to ensure the swift development of new features, and is a reason why users and developers may not want to rely on prototype or beta components.

Current stable behavior should, ideally, not be deprecated in favor of prototype or beta features. The deprecation should generally wait until alternatives to it are stable.

Prototype and beta components should warn users before use (using `TORCH_WARN_ONCE`) and provide a clear warning notice in their documentation that they are subject to change at any time without warning.

"Silent" Backwards Compatibility Breakages

A "silent" backwards compatibility breakage is when a PyTorch program continues running an older program but it produces semantically different results. This is a "silent" breakage because no error is thrown, but the change may still invalidate a program's output. As NumPy's Backwards Compatibility policy states, "the possibility of an incorrect result is worse than an error or even crash." Silent breakages should be avoided by introducing an error so they become "loud," like other changes.

A typical example of making a "silent" breakage "loud" is changing the default value for a keyword argument. For example, consider a function with the signature `foo(t, *, alpha=1)`. Changing the function to `foo(t, *, alpha=2)` would be a "silent" BC-breakage because programs calling `foo(a)` would continue to work, but they're now setting `alpha` differently! To make this a "loud" BC-breakage the alpha kwarg should be temporarily required, first by throwing a warning when users don't explicitly set the kwarg, then by requiring the kwarg be set (changing the signature to `foo(t, *, alpha)`, and finally by applying the new default. Requiring `alpha` causes programs with `foo(a)` to break and stipulate they want `foo(a, alpha=1)` to preserve their current behavior.

Forwards compatibility (FC)

A version of PyTorch is "forwards compatible (FC)" with future versions if programs written for those future versions run in the current version. This is practically impossible to achieve completely because PyTorch is constantly evolving and adding new features which won't work on previous versions. However, a

best effort is made for forwards compatibility, and there is a particular case we try hard to support:

- If a program runs on PyTorch with commit X, then the same program serialized using torchscript by a commit of PyTorch no more than two weeks older than X should also run on commit X.

This is a very technical case that is unlikely to come up for most users, especially users who don't use nightly builds, build from source, or use torchscript. Conceptually PyTorch tries to support a two-week torchscript FC window which allows PyTorch users leeway when updating multiple systems using PyTorch. For instance if a company has one system to train PyTorch models and another that runs them using torchscript, then this allows the the version of PyTorch running models to be up to two weeks older than the version that trains them.

Note that adding a new keyword argument with a default value to an operation should never be FC-breaking (or BC-breaking, per above).

Example Changes

Changing the behavior of a mode

Let's consider a hypothetical function `foo(..., *, mode)` that accepts a variety of modes controlling its behavior. (This function might be like `torch.div()`, which accepts a `rounding_mode` kwarg.) Now let's say that we want to change the behavior of one of those modes, but we are concerned that the change will affect backwards compatibility. For ease of exposition, we'll refer to this hypothetical mode as Mode A. Here's how that change could be made:

- create a new mode for the desired behavior of Mode A, we'll call it Mode B
- create a new mode with the same behavior as Mode A, we'll call it Mode C, if keeping a mode with Mode A's behavior is desired
- throw a warning when Mode A is used, telling users that it will be removed and they should use Mode C (if available) to preserve their behavior or Mode B if they would prefer the new behavior
- update `foo`'s documentation to indicate that Mode A is deprecated, when it was deprecated, and what the alternatives to it are
- wait for the warning to be thrown for 2 releases (180 days on nightlies)
- remove Mode A
- update `foo`'s documentation to indicate that Mode A was removed, when it was removed, and what the alternatives to it are
- wait 2 releases (180 days on nightlies)
- restore Mode A with the functionality of Mode B
- update `foo`'s documentation to indicate that Mode A was restored and when it was restored

Once Mode A is restored Mode B can be deprecated.

Making an arg keyword-only

Let's consider a hypothetical function `foo(..., ARG, *, ...)` with a positional argument ARG. This function might have been originally developed back when PyTorch supported Python 2 and keyword-only arguments didn't exist, but now PyTorch only supports later Python 3 versions and many arguments are more readable if specified as keywords. (Function calls like `foo(t, 1, 3, True)` are very tricky to read compared to `foo(t, dim=1, alpha=3, keepdim=True)`.) Here's how an argument can be made keyword-only:

- throw a warning when ARG is specified positionally
- update `foo`'s documentation to show ARG as a keyword-only argument
- wait for the warning to be thrown for 2 releases (180 days on nightlies)
- remove support for specifying ARG positionally

Legacy Torchscript Backwards Compatibility

The document above specifies PyTorch's Frontend Backward and Forward Compatibility Policy. Community members should use it as a guide to understand PyTorch's stability, and developers should use it as a guide to minimize the disruption of BC-breaking changes. An important implementation detail for developers, however, is that in addition to the above policy there is a legacy torchscript requirement, described in this section. In the future we expect this requirement to go away (that's why it's legacy).

History

PyTorch didn't always communicate a clear policy backwards or forwards compatibility policy to its community. As such, the community expected that their programs would continue working as they always had, and based on this belief some community members serialized torchscript programs once and lost the ability to update them. We encourage all community members to "phase out" these fossilized programs and be ready to update programs that have been throwing warnings for two releases. (In the future PyTorch will likely help identify these programs with new tools, like an improved messaging system). However, because this policy is new we want to respect our users and do our best to keep old serialized torchscript programs running even as we encourage the community to adapt this new model.

Upgraders

To keep old serialized torchscript programs working as expected we write upgraders that map historic operator calls to modern operator calls. For example, in PyTorch the `torch.div()` operator used to perform truncation division when dividing integer inputs like Python 2 and C++ do. It now always performs true division like Python 3 and NumPy do, and there's an upgrader that maps

old serialized `torch.div()` into either a true division or truncation division call depending on the datatype of its inputs.

Writing an upgrader is required for any BC-breaking change that can affect an older serialized torchscript model.