# Table of Contents

## Contributing to PyTorch

Thank you for your interest in contributing to PyTorch! Before you begin writing code, it is important that you share your intention to contribute with the team, based on the type of contribution:

1. You want to propose a new feature and implement it.
   - Post about your intended feature in an issue, and we shall discuss the design and implementation. Once we agree that the plan looks good, go ahead and implement it.
2. You want to implement a feature or bug-fix for an outstanding issue.
   - Search for your issue in the PyTorch issue list.
   - Pick an issue and comment that you'd like to work on the feature or bug-fix.
   - If you need more context on a particular issue, please ask and we shall provide.

Once you implement and test your feature or bug-fix, please submit a Pull Request to https://github.com/pytorch/pytorch.

This document covers some of the more technical aspects of contributing to PyTorch. For more non-technical guidance about how to contribute to PyTorch, see the Contributing Guide.

## Developing PyTorch

A full set of instructions on installing PyTorch from source is here: https://github.com/pytorch/pytorch#from-source

To develop PyTorch on your machine, here are some tips:

1. Uninstall all existing PyTorch installs. You may need to run `pip uninstall torch` multiple times. You'll know `torch` is fully uninstalled when you see `WARNING: Skipping torch as it is not installed`. (You should only have to `pip uninstall` a few times, but you can always `uninstall` with `timeout` or in a loop if you're feeling lazy.)

```
conda uninstall pytorch -y
yes | pip uninstall torch
```

2. Clone a copy of PyTorch from source:

```
git clone https://github.com/pytorch/pytorch
cd pytorch
```

2.1. If you already have PyTorch from source, update it:

```
git pull --rebase
git submodule sync --recursive
git submodule update --init --recursive --jobs 0
```

If you want to have no-op incremental rebuilds (which are fast), see the section below titled "Make no-op build fast."

3. Install PyTorch in `develop` mode:

The change you have to make is to replace

```
python setup.py install
```

with

```
python setup.py develop
```

This mode will symlink the Python files from the current local source tree into the Python install. Hence, if you modify a Python file, you do not need to reinstall PyTorch again and again. This is especially useful if you are only changing Python files.

For example:

- Install local PyTorch in `develop` mode
- modify your Python file `torch/__init__.py` (for example)
- test functionality

You do not need to repeatedly install after modifying Python files ( `.py` ). However, you would need to reinstall if you modify Python interface ( `.pyi` , `.pyi.in` ) or non-Python files ( `.cpp` , `.cc` , `.cu` , `.h` , ...).

In case you want to reinstall, make sure that you uninstall PyTorch first by running `pip uninstall torch` until you see `WARNING: Skipping torch as it is not installed` ; next run `python setup.py clean` . After that, you can install in `develop` mode again.

### Tips and Debugging

- A prerequisite to installing PyTorch is CMake. We recommend installing it with Homebrew with `brew install cmake` if you are developing on MacOS or Linux system.
- Our `setup.py` requires Python >= 3.7
- If a commit is simple and doesn't affect any code (keep in mind that some docstrings contain code that is used in tests), you can add `[skip ci]` (case sensitive) somewhere in your commit message to skip all build / test steps. Note that changing the pull request body or title on GitHub itself has no effect.
- If you run into errors when running `python setup.py develop` , here are some debugging steps:
    1. Run `printf '#include <stdio.h>\nint main() { printf("Hello World");}'|clang -x c -; ./a.out` to make sure your CMake works and can compile this simple Hello World program without errors.
    2. Nuke your `build` directory. The `setup.py` script compiles binaries into the `build` folder and caches many details along the way, which saves time the next time you build. If you're running into issues, you can always `rm -rf build` from the toplevel `pytorch` directory and start over.
    3. If you have made edits to the PyTorch repo, commit any change you'd like to keep and clean the repo with the following commands (note that clean *really* removes all untracked files and changes.):

```
git submodule deinit -f .
git clean -xdf
python setup.py clean
git submodule update --init --recursive --jobs 0 # very important to sync the
submodules
```

```
python setup.py develop                                  # then try running the
command again
```

4. The main step within `python setup.py develop` is running `make` from the `build` directory. If you want to experiment with some environment variables, you can pass them into the command:

```
ENV_KEY1=ENV_VAL1[, ENV_KEY2=ENV_VAL2]* python setup.py develop
```

- If you run into issue running `git submodule update --init --recursive --jobs 0`. Please try the following:
  - If you encountered error such as

    ```
    error: Submodule 'third_party/pybind11' could not be updated
    ```

    check whether your Git local or global config file contains any `submodule.*` settings. If yes, remove them and try again. (please reference this doc for more info).

  - If you encountered error such as

    ```
    fatal: unable to access 'https://github.com/pybind11/pybind11.git':
    could not load PEM client certificate ...
    ```

    this is likely that you are using HTTP proxying and the certificate expired. To check if the certificate is valid, run `git config --global --list` and search for config like `http.proxysslcert=<cert_file>`. Then check certificate valid date by running

    ```
     openssl x509 -noout -in <cert_file> -dates
    ```

  - If you encountered error that some third_party modules are not checkout correctly, such as

    ```
    Could not find .../pytorch/third_party/pybind11/CMakeLists.txt
    ```

    remove any `submodule.*` settings in your local git config ( `.git/config` of your pytorch repo) and try again.

- If you're a Windows contributor, please check out Best Practices.

## Nightly Checkout & Pull

The `tools/nightly.py` script is provided to ease pure Python development of PyTorch. This uses `conda` and `git` to check out the nightly development version of PyTorch and installs pre-built binaries into the current repository. This is like a development or editable install, but without needing the ability to compile any C++ code.

You can use this script to check out a new nightly branch with the following:

```
./tools/nightly.py checkout -b my-nightly-branch
conda activate pytorch-deps
```

Or if you would like to re-use an existing conda environment, you can pass in the regular environment parameters (`--name` or `--prefix`):

```
./tools/nightly.py checkout -b my-nightly-branch -n my-env
conda activate my-env
```

You can also use this tool to pull the nightly commits into the current branch:

```
./tools/nightly.py pull -n my-env
conda activate my-env
```

Pulling will reinstall the PyTorch dependencies as well as the nightly binaries into the repo directory.

## Codebase structure

- c10 - Core library files that work everywhere, both server and mobile. We are slowly moving pieces from ATen/core here. This library is intended only to contain essential functionality, and appropriate to use in settings where binary size matters. (But you'll have a lot of missing functionality if you try to use it directly.)
- aten - C++ tensor library for PyTorch (no autograd support)
  - src - README
    - ATen
      - core - Core functionality of ATen. This is migrating to top-level c10 folder.
      - native - Modern implementations of operators. If you want to write a new operator, here is where it should go. Most CPU operators go in the top level directory, except for operators which need to be compiled specially; see cpu below.
        - cpu - Not actually CPU implementations of operators, but specifically implementations which are compiled with processor-specific instructions, like AVX. See the README for more details.
        - cuda - CUDA implementations of operators.
        - sparse - CPU and CUDA implementations of COO sparse tensor operations
        - mkl mkldnn miopen cudnn
          - implementations of operators which simply bind to some backend library.
        - quantized - Quantized tensor (i.e. QTensor) operation implementations. README contains details including how to implement native quantized operations.

- torch - The actual PyTorch library. Everything that is not in csrc is a Python module, following the PyTorch Python frontend module structure.
  - csrc - C++ files composing the PyTorch library. Files in this directory tree are a mix of Python binding code, and C++ heavy lifting. Consult `setup.py` for the canonical list of Python binding files; conventionally, they are often prefixed with `python_`. README
    - jit - Compiler and frontend for TorchScript JIT frontend. README
    - autograd - Implementation of reverse-mode automatic differentiation. README
    - api - The PyTorch C++ frontend.
    - distributed - Distributed training support for PyTorch.

- tools - Code generation scripts for the PyTorch library. See README of this directory for more details.
- test - Python unit tests for PyTorch Python frontend.
  - test_torch.py - Basic tests for PyTorch functionality.
  - test_autograd.py - Tests for non-NN automatic differentiation support.
  - test_nn.py - Tests for NN operators and their automatic differentiation.
  - test_jit.py - Tests for the JIT compiler and TorchScript.
  - ...
  - cpp - C++ unit tests for PyTorch C++ frontend.
    - api - README
    - jit - README
    - tensorexpr - README
  - expect - Automatically generated "expect" files which are used to compare against expected output.
  - onnx - Tests for ONNX export functionality, using both PyTorch and Caffe2.
- caffe2 - The Caffe2 library.
  - core - Core files of Caffe2, e.g., tensor, workspace, blobs, etc.
  - operators - Operators of Caffe2.
  - python - Python bindings to Caffe2.
  - ...
- .circleci - CircleCI configuration management. README

## Unit testing

### Python Unit Testing

All PyTorch test suites are located in the `test` folder and start with `test_`. Run the entire test suite with

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

or run individual test suites using the command `python test/FILENAME.py`, where `FILENAME` represents the file containing the test suite you wish to run.

For example, to run all the TorchScript JIT tests (located at `test/test_jit.py`), you would run:

```
python test/test_jit.py
```

You can narrow down what you're testing even further by specifying the name of an individual test with `TESTCLASSNAME.TESTNAME`. Here, `TESTNAME` is the name of the test you want to run, and `TESTCLASSNAME` is the name of the class in which it is defined.

Going off the above example, let's say you want to run `test_Sequential`, which is defined as part of the `TestJit` class in `test/test_jit.py`. Your command would be:

```
python test/test_jit.py TestJit.test_Sequential
```

The `expecttest` and `hypothesis` libraries must be installed to run the tests. `mypy` is an optional dependency, and `pytest` may help run tests more selectively. All these packages can be installed with `conda` or `pip`.

### Better local unit tests with `pytest`

We don't officially support `pytest`, but it works well with our `unittest` tests and offers a number of useful features for local developing. Install it via `pip install pytest`.

If you want to just run tests that contain a specific substring, you can use the `-k` flag:

```
pytest test/test_nn.py -k Loss -v
```

The above is an example of testing a change to all Loss functions: this command runs tests such as `TestNN.test_BCELoss` and `TestNN.test_MSELoss` and can be useful to save keystrokes.

### Local linting

You can run the same linting steps that are used in CI locally via `make`:

```
# Lint all files
make lint -j 6  # run lint (using 6 parallel jobs)

# Lint only the files you have changed
make quicklint -j 6
```

These jobs may require extra dependencies that aren't dependencies of PyTorch itself, so you can install them via this command, which you should only have to run once:

```
make setup_lint
```

To run a specific linting step, use one of these targets or see the `Makefile` for a complete list of options.

```
# Check for tabs, trailing newlines, etc.
make quick_checks

make flake8

make mypy

make cmakelint

make clang-tidy
```

To run a lint only on changes, add the `CHANGED_ONLY` option:

```
make <name of lint> CHANGED_ONLY=--changed-only
```

### Running `mypy`

`mypy` is an optional static type checker for Python. We have multiple `mypy` configs for the PyTorch codebase, so you can run them all using this command:

```
make mypy
```

See Guide for adding type annotations to PyTorch for more information on how to set up `mypy` and tackle type annotation tasks.

### C++ Unit Testing

PyTorch offers a series of tests located in the `test/cpp` folder. These tests are written in C++ and use the Google Test testing framework. After compiling PyTorch from source, the test runner binaries will be written to the `build/bin` folder. The command to run one of these tests is `./build/bin/FILENAME --gtest_filter=TESTSUITE.TESTNAME`, where `TESTNAME` is the name of the test you'd like to run and `TESTSUITE` is the suite that test is defined in.

For example, if you wanted to run the test `MayContainAlias`, which is part of the test suite `ContainerAliasingTest` in the file `test/cpp/jit/test_alias_analysis.cpp`, the command would be:

```
./build/bin/test_jit --gtest_filter=ContainerAliasingTest.MayContainAlias
```

### Run Specific CI Jobs

You can generate a commit that limits the CI to only run a specific job by using `tools/testing/explicit_ci_jobs.py` like so:

```
# --job: specify one or more times to filter to a specific job + its dependencies
# --filter-gha: specify github actions workflows to keep
# --make-commit: commit CI changes to git with a message explaining the change
python tools/testing/explicit_ci_jobs.py --job
binary_linux_manywheel_3_6m_cpu_devtoolset7_nightly_test --filter-gha
'*generated*gcc5.4*' --make-commit

# Make your changes

ghstack submit
```

**NB**: It is not recommended to use this workflow unless you are also using `ghstack`. It creates a large commit that is of very low signal to reviewers.

## Writing documentation

So you want to write some documentation and don't know where to start? PyTorch has two main types of documentation:

- user-facing documentation. These are the docs that you see over at our docs website.
- developer facing documentation. Developer facing documentation is spread around our READMEs in our codebase and in the PyTorch Developer Wiki. If you're interested in adding new developer docs, please read this page on the wiki on our best practices for where to put it.

The rest of this section is about user-facing documentation.

PyTorch uses Google style for formatting docstrings. Length of line inside docstrings block must be limited to 80 characters to fit into Jupyter documentation popups.

### Building documentation

To build the documentation:

1. Build and install PyTorch
2. Install the prerequisites

```
cd docs
pip install -r requirements.txt
# `katex` must also be available in your PATH.
# You can either install katex globally if you have properly configured npm:
# npm install -g katex
# Or if you prefer an uncontaminated global executable environment or do not want to
go through the node configuration:
# npm install katex && export PATH="$PATH:$(pwd)/node_modules/.bin"
```

> *Note that if you are a Facebook employee using a devserver, yarn may be more convenient to install katex:*

```
yarn global add katex
```

3. Generate the documentation HTML files. The generated files will be in `docs/build/html`.

```
make html
```

#### Tips

The `.rst` source files live in docs/source. Some of the `.rst` files pull in docstrings from PyTorch Python code (for example, via the `autofunction` or `autoclass` directives). To vastly shorten doc build times, it is helpful to remove the files you are not working on, only keeping the base `index.rst` file and the files you are editing. The Sphinx build will produce missing file warnings but will still complete. For example, to work on `jit.rst`:

```
cd docs/source
find . -type f | grep rst | grep -v index | grep -v jit | xargs rm

# Make your changes, build the docs, etc.

# Don't commit the deletions!
git add index.rst jit.rst
...
```

#### Building C++ Documentation

For C++ documentation (https://pytorch.org/cppdocs), we use Doxygen and then convert it to Sphinx via Breathe and Exhale. Check the Doxygen reference for more information on the documentation syntax.

We run Doxygen in CI (Travis) to verify that you do not use invalid Doxygen commands. To run this check locally, run `./check-doxygen.sh` from inside `docs/cpp/source`.

To build the documentation, follow the same steps as above, but run them from `docs/cpp` instead of `docs`.

### Previewing changes locally

To view HTML files locally, you can open the files in your web browser. For example, navigate to `file:///your_pytorch_folder/docs/build/html/index.html` in a web browser.

If you are developing on a remote machine, you can set up an SSH tunnel so that you can access the HTTP server on the remote machine from your local machine. To map remote port 8000 to local port 8000, use either of the following commands.

```
# For SSH
ssh my_machine -L 8000:my_machine:8000

# For Eternal Terminal
et my_machine -t="8000:8000"
```

Then navigate to `localhost:8000` in your web browser.

**Tip:** You can start a lightweight HTTP server on the remote machine with:

```
python -m http.server 8000 <path_to_html_output>
```

Alternatively, you can run `rsync` on your local machine to copy the files from your remote machine:

```
mkdir -p build cpp/build
rsync -az me@my_machine:/path/to/pytorch/docs/build/html build
rsync -az me@my_machine:/path/to/pytorch/docs/cpp/build/html cpp/build
```

### Previewing documentation on PRs

PyTorch will host documentation previews at `https://docs-preview.pytorch.org/<pr number>/` once the `pytorch_python_doc_build` GitHub Actions job has completed on your PR. You can visit that page directly or find its link in the automated Dr. CI comment on your PR.

### Adding documentation tests

It is easy for code snippets in docstrings and `.rst` files to get out of date. The docs build includes the Sphinx Doctest Extension, which can run code in documentation as a unit test. To use the extension, use the `.. testcode::` directive in your `.rst` and docstrings.

To manually run these tests, follow steps 1 and 2 above, then run:

```
cd docs
make doctest
```

## Profiling with `py-spy`

Evaluating the performance impact of code changes in PyTorch can be complicated, particularly if code changes happen in compiled code. One simple way to profile both Python and C++ code in PyTorch is to use `py-spy`, a sampling profiler for Python that has the ability to profile native code and Python code in the same session.

`py-spy` can be installed via `pip`:

```
pip install py-spy
```

To use `py-spy`, first write a Python test script that exercises the functionality you would like to profile. For example, this script profiles `torch.add`:

```
import torch

t1 = torch.tensor([[1, 1], [1, 1.]])
t2 = torch.tensor([[0, 0], [0, 0.]])

for _ in range(1000000):
    torch.add(t1, t2)
```

Since the `torch.add` operation happens in microseconds, we repeat it a large number of times to get good statistics. The most straightforward way to use `py-spy` with such a script is to generate a flame graph:

```
py-spy record -o profile.svg --native -- python test_tensor_tensor_add.py
```

This will output a file named `profile.svg` containing a flame graph you can view in a web browser or SVG viewer. Individual stack frame entries in the graph can be selected interactively with your mouse to zoom in on a particular part of the program execution timeline. The `--native` command-line option tells `py-spy` to record stack frame entries for PyTorch C++ code. To get line numbers for C++ code it may be necessary to compile PyTorch in debug mode by prepending your `setup.py develop` call to compile PyTorch with `DEBUG=1`. Depending on your operating system it may also be necessary to run `py-spy` with root privileges.

`py-spy` can also work in an `htop`-like "live profiling" mode and can be tweaked to adjust the stack sampling rate, see the `py-spy` readme for more details.

## Managing multiple build trees

One downside to using `python setup.py develop` is that your development version of PyTorch will be installed globally on your account (e.g., if you run `import torch` anywhere else, the development version will be used.

If you want to manage multiple builds of PyTorch, you can make use of conda environments to maintain separate Python package environments, each of which can be tied to a specific build of PyTorch. To set one up:

```
conda create -n pytorch-myfeature
source activate pytorch-myfeature
# if you run python now, torch will NOT be installed
python setup.py develop
```

## C++ development tips

If you are working on the C++ code, there are a few important things that you will want to keep in mind:

1. How to rebuild only the code you are working on.
2. How to make rebuilds in the absence of changes go faster.

### Build only what you need

`python setup.py build` will build everything by default, but sometimes you are only interested in a specific component.

- Working on a test binary? Run `(cd build && ninja bin/test_binary_name)` to rebuild only that test binary (without rerunning cmake). (Replace `ninja` with `make` if you don't have ninja installed).
- Don't need Caffe2? Pass `BUILD_CAFFE2=0` to disable Caffe2 build.

On the initial build, you can also speed things up with the environment variables `DEBUG`, `USE_DISTRIBUTED`, `USE_MKLDNN`, `USE_CUDA`, `BUILD_TEST`, `USE_FBGEMM`, `USE_NNPACK` and `USE_QNNPACK`.

- `DEBUG=1` will enable debug builds (-g -O0)
- `REL_WITH_DEB_INFO=1` will enable debug symbols with optimizations (-g -O3)
- `USE_DISTRIBUTED=0` will disable distributed (c10d, gloo, mpi, etc.) build.
- `USE_MKLDNN=0` will disable using MKL-DNN.
- `USE_CUDA=0` will disable compiling CUDA (in case you are developing on something not CUDA related), to save compile time.
- `BUILD_TEST=0` will disable building C++ test binaries.
- `USE_FBGEMM=0` will disable using FBGEMM (quantized 8-bit server operators).
- `USE_NNPACK=0` will disable compiling with NNPACK.
- `USE_QNNPACK=0` will disable QNNPACK build (quantized 8-bit operators).
- `USE_XNNPACK=0` will disable compiling with XNNPACK.

For example:

```
DEBUG=1 USE_DISTRIBUTED=0 USE_MKLDNN=0 USE_CUDA=0 BUILD_TEST=0 USE_FBGEMM=0
USE_NNPACK=0 USE_QNNPACK=0 USE_XNNPACK=0 python setup.py develop
```

For subsequent builds (i.e., when `build/CMakeCache.txt` exists), the build options passed for the first time will persist; please run `ccmake build/`, run `cmake-gui build/`, or directly edit `build/CMakeCache.txt` to adapt build options.

### Code completion and IDE support

When using `python setup.py develop`, PyTorch will generate a `compile_commands.json` file that can be used by many editors to provide command completion and error highlighting for PyTorch's C++ code. You need to `pip install ninja` to generate accurate information for the code in `torch/csrc`. More information at:

- https://sarcasm.github.io/notes/dev/compilation-database.html

### Make no-op build fast

#### Use Ninja

By default, cmake will use its Makefile generator to generate your build system. You can get faster builds if you install the ninja build system with `pip install ninja`. If PyTorch was already built, you will need to run `python setup.py clean` once after installing ninja for builds to succeed.

#### Use CCache

Even when dependencies are tracked with file modification, there are many situations where files get rebuilt when a previous compilation was exactly the same. Using ccache in a situation like this is a real time-saver.

Before building pytorch, install ccache from your package manager of choice:

```
conda install ccache -f conda-forge
sudo apt install ccache
sudo yum install ccache
brew install ccache
```

You may also find the default cache size in ccache is too small to be useful. The cache sizes can be increased from the command line:

```
# config: cache dir is ~/.ccache, conf file ~/.ccache/ccache.conf
# max size of cache
ccache -M 25Gi  # -M 0 for unlimited
# unlimited number of files
ccache -F 0
```

To check this is working, do two clean builds of pytorch in a row. The second build should be substantially and noticeably faster than the first build. If this doesn't seem to be the case, check the `CMAKE_<LANG>_COMPILER_LAUNCHER` rules in `build/CMakeCache.txt`, where `<LANG>` is `C`, `CXX` and `CUDA`. Each of these 3 variables should contain ccache, e.g.

```
//CXX compiler launcher
CMAKE_CXX_COMPILER_LAUNCHER:STRING=/usr/bin/ccache
```

If not, you can define these variables on the command line before invoking `setup.py`.

```
export CMAKE_C_COMPILER_LAUNCHER=ccache
export CMAKE_CXX_COMPILER_LAUNCHER=ccache
export CMAKE_CUDA_COMPILER_LAUNCHER=ccache
python setup.py develop
```

#### Use a faster linker

If you are editing a single file and rebuilding in a tight loop, the time spent linking will dominate. The system linker available in most Linux distributions (GNU `ld`) is quite slow. Use a faster linker, like lld.

People on Mac, follow this guide instead.

The easiest way to use `lld` this is download the latest LLVM binaries and run:

```
ln -s /path/to/downloaded/ld.lld /usr/local/bin/ld
```

#### Use pre-compiled headers

Sometimes there's no way of getting around rebuilding lots of files, for example editing `native_functions.yaml` usually means 1000+ files being rebuilt. If you're using CMake newer than 3.16, you can enable pre-compiled headers by setting `USE_PRECOMPILED_HEADERS=1` either on first setup, or in the `CMakeCache.txt` file.

```
USE_PRECOMPILED_HEADERS=1 python setup.py develop
```

This adds a build step where the compiler takes `<ATen/ATen.h>` and essentially dumps it's internal AST to a file so the compiler can avoid repeating itself for every `.cpp` file.

One caveat is that when enabled, this header gets included in every file by default. Which may change what code is legal, for example:

- internal functions can never alias existing names in `<ATen/ATen.h>`
- names in `<ATen/ATen.h>` will work even if you don't explicitly include it.

#### Workaround for header dependency bug in nvcc

If re-building without modifying any files results in several CUDA files being re-compiled, you may be running into an `nvcc` bug where header dependencies are not converted to absolute paths before reporting it to the build system. This makes `ninja` think one of the header files has been deleted, so it runs the build again.

A compiler-wrapper to fix this is provided in `tools/nvcc_fix_deps.py`. You can use this as a compiler launcher, similar to `ccache`

```
export CMAKE_CUDA_COMPILER_LAUNCHER="python;`pwd`/tools/nvcc_fix_deps.py;ccache"
python setup.py develop
```

### C++ frontend development tips

We have very extensive tests in the test/cpp/api folder. The tests are a great way to see how certain components are intended to be used. When compiling PyTorch from source, the test runner binary will be written to `build/bin/test_api`. The tests use the GoogleTest framework, which you can read up about to learn how to configure the test runner. When submitting a new feature, we care very much that you write appropriate tests. Please follow the lead of the other tests to see how to write a new test case.

### GDB integration

If you are debugging pytorch inside GDB, you might be interested in pytorch-gdb. This script introduces some pytorch-specific commands which you can use from the GDB prompt. In particular, `torch-tensor-repr` prints a human-readable repr of an at::Tensor object. Example of usage:

```
$ gdb python
GNU gdb (GDB) 9.2
[...]
(gdb) # insert a breakpoint when we call .neg()
(gdb) break at::Tensor::neg
Function "at::Tensor::neg" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (at::Tensor::neg) pending.
```

```
(gdb) run
[...]
>>> import torch
>>> t = torch.tensor([1, 2, 3, 4], dtype=torch.float64)
>>> t
tensor([1., 2., 3., 4.], dtype=torch.float64)
>>> t.neg()

Thread 1 "python" hit Breakpoint 1, at::Tensor::neg (this=0x7ffb118a9c88) at
aten/src/ATen/core/TensorBody.h:3295
3295    inline at::Tensor Tensor::neg() const {
(gdb) # the default repr of 'this' is not very useful
(gdb) p this
$1 = (const at::Tensor * const) 0x7ffb118a9c88
(gdb) p *this
$2 = {impl_ = {target_ = 0x55629b5cd330}}
(gdb) torch-tensor-repr *this
Python-level repr of *this:
tensor([1., 2., 3., 4.], dtype=torch.float64)
```

GDB tries to automatically load `pytorch-gdb` thanks to the .gdbinit at the root of the pytorch repo. However, auto-loadings is disabled by default, because of security reasons:

```
$ gdb
warning: File "/path/to/pytorch/.gdbinit" auto-loading has been declined by your
`auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /path/to/pytorch/.gdbinit
line to your configuration file "/home/YOUR-USERNAME/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/YOUR-USERNAME/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
(gdb)
```

As gdb itself suggests, the best way to enable auto-loading of `pytorch-gdb` is to add the following line to your `~/.gdbinit` (i.e., the `.gdbinit` file which is in your home directory, **not** `/path/to/pytorch/.gdbinit`):

```
add-auto-load-safe-path /path/to/pytorch/.gdbinit
```

### C++ stacktraces

Set `TORCH_SHOW_CPP_STACKTRACES=1` to get the C++ stacktrace when an error occurs in Python.

## CUDA development tips

If you are working on the CUDA code, here are some useful CUDA debugging tips:

1. `CUDA_DEVICE_DEBUG=1` will enable CUDA device function debug symbols ( `-g -G` ). This will be particularly helpful in debugging device code. However, it will slow down the build process for about 50% (compared to only `DEBUG=1` ), so use wisely.
2. `cuda-gdb` and `cuda-memcheck` are your best CUDA debugging friends. Unlike `gdb` , `cuda-gdb` can display actual values in a CUDA tensor (rather than all zeros).
3. CUDA supports a lot of C++11/14 features such as, `std::numeric_limits` , `std::nextafter` , `std::tuple` etc. in device code. Many of such features are possible because of the [--expt-relaxed-constexpr](--expt-relaxed-constexpr) nvcc flag. There is a known [issue](issue) that ROCm errors out on device code, which uses such stl functions.
4. A good performance metric for a CUDA kernel is the [Effective Memory Bandwidth](Effective Memory Bandwidth). It is useful for you to measure this metric whenever you are writing/optimizing a CUDA kernel. Following script shows how we can measure the effective bandwidth of CUDA `uniform_` kernel.

```
import torch
from torch.utils.benchmark import Timer
size = 128*512
nrep = 100
nbytes_read_write = 4 # this is number of bytes read + written by a kernel.
Change this to fit your kernel.

for i in range(10):
    a=torch.empty(size).cuda().uniform_()
    torch.cuda.synchronize()
    out = a.uniform_()
    torch.cuda.synchronize()
    t = Timer(stmt="a.uniform_()", globals=globals())
    res = t.blocked_autorange()
    timec = res.median
    print("uniform, size, elements", size, "forward", timec, "bandwidth
(GB/s)", size*(nbytes_read_write)*1e-9/timec)
    size *=2
```

See more cuda development tips [here](here)

## Windows development tips

For building from source on Windows, consult [our documentation](our documentation) on it.

Occasionally, you will write a patch which works on Linux, but fails CI on Windows. There are a few aspects in which MSVC (the Windows compiler toolchain we use) is stricter than Linux, which are worth keeping in mind when fixing these problems.

1. Symbols are NOT exported by default on Windows; instead, you have to explicitly mark a symbol as exported/imported in a header file with `__declspec(dllexport)` / `__declspec(dllimport)` . We have codified this pattern into a set of macros which follow the convention `*_API` , e.g., `TORCH_API` inside Caffe2, Aten and Torch. (Every separate shared library needs a unique macro name, because symbol visibility is on a per shared library basis. See c10/macros/Macros.h for more details.)

   The upshot is if you see an "unresolved external" error in your Windows build, this is probably because you forgot to mark a function with `*_API` . However, there is one important counterexample to this principle:

if you want a *templated* function to be instantiated at the call site, do NOT mark it with `*_API` (if you do mark it, you'll have to explicitly instantiate all of the specializations used by the call sites.)

2. If you link against a library, this does not make its dependencies transitively visible. You must explicitly specify a link dependency against every library whose symbols you use. (This is different from Linux where in most environments, transitive dependencies can be used to fulfill unresolved symbols.)

3. If you have a Windows box (we have a few on EC2 which you can request access to) and you want to run the build, the easiest way is to just run `.jenkins/pytorch/win-build.sh`. If you need to rebuild, run `REBUILD=1 .jenkins/pytorch/win-build.sh` (this will avoid blowing away your Conda environment.)

Even if you don't know anything about MSVC, you can use cmake to build simple programs on Windows; this can be helpful if you want to learn more about some peculiar linking behavior by reproducing it on a small example. Here's a simple example cmake file that defines two dynamic libraries, one linking with the other:

```
project(myproject CXX)
set(CMAKE_CXX_STANDARD 14)
add_library(foo SHARED foo.cpp)
add_library(bar SHARED bar.cpp)
# NB: don't forget to __declspec(dllexport) at least one symbol from foo,
# otherwise foo.lib will not be created.
target_link_libraries(bar PUBLIC foo)
```

You can build it with:

```
mkdir build
cd build
cmake ..
cmake --build .
```

### Known MSVC (and MSVC with NVCC) bugs

The PyTorch codebase sometimes likes to use exciting C++ features, and these exciting features lead to exciting bugs in Windows compilers. To add insult to injury, the error messages will often not tell you which line of code actually induced the erroring template instantiation.

We've found the most effective way to debug these problems is to carefully read over diffs, keeping in mind known bugs in MSVC/NVCC. Here are a few well known pitfalls and workarounds:

- This is not actually a bug per se, but in general, code generated by MSVC is more sensitive to memory errors; you may have written some code that does a use-after-free or stack overflows; on Linux the code might work, but on Windows your program will crash. ASAN may not catch all of these problems: stay vigilant to the possibility that your crash is due to a real memory problem.

- (NVCC) `c10::optional` does not work when used from device code. Don't use it from kernels. Upstream issue: https://github.com/akrzemi1/Optional/issues/58 and our local issue #10329.

- `constexpr` generally works less well on MSVC.

    - The idiom `static_assert(f() == f())` to test if `f` is constexpr does not work; you'll get "error C2131: expression did not evaluate to a constant". Don't use these asserts on Windows.

(Example: `c10/util/intrusive_ptr.h`)

- (NVCC) Code you access inside a `static_assert` will eagerly be evaluated as if it were device code, and so you might get an error that the code is "not accessible".

```
class A {
  static A singleton_;
  static constexpr inline A* singleton() {
    return &singleton_;
  }
};
static_assert(std::is_same(A*, decltype(A::singleton()))::value, "hmm");
```

- The compiler will run out of heap space if you attempt to compile files that are too large. Splitting such files into separate files helps. (Example: `THTensorMath`, `THTensorMoreMath`, `THTensorEvenMoreMath`.)
- MSVC's preprocessor (but not the standard compiler) has a bug where it incorrectly tokenizes raw string literals, ending when it sees a `"`. This causes preprocessor tokens inside the literal like an `#endif` to be incorrectly treated as preprocessor directives. See https://godbolt.org/z/eVTIJq as an example.
- Either MSVC or the Windows headers have a PURE macro defined and will replace any occurrences of the PURE token in code with an empty string. This is why we have AliasAnalysisKind::PURE_FUNCTION and not AliasAnalysisKind::PURE. The same is likely true for other identifiers that we just didn't try to use yet.

### Building on legacy code and CUDA

CUDA, MSVC, and PyTorch versions are interdependent; please install matching versions from this table:

| CUDA version | Newest supported VS version | PyTorch version |
|---|---|---|
| 10.1 | Visual Studio 2019 (16.X) (`_MSC_VER` < 1930) | 1.3.0 ~ 1.7.0 |
| 10.2 | Visual Studio 2019 (16.X) (`_MSC_VER` < 1930) | 1.5.0 ~ 1.7.0 |
| 11.0 | Visual Studio 2019 (16.X) (`_MSC_VER` < 1930) | 1.7.0 |

Note: There's a compilation issue in several Visual Studio 2019 versions since 16.7.1, so please make sure your Visual Studio 2019 version is not in 16.7.1 ~ 16.7.5

## Running clang-tidy

Clang-Tidy is a C++ linter and static analysis tool based on the clang compiler. We run clang-tidy in our CI to make sure that new C++ code is safe, sane and efficient. See the `clang-tidy` job in our GitHub Workflow's lint.yml file for the simple commands we use for this.

To run clang-tidy locally, follow these steps:

1. Install clang-tidy. We provide custom built binaries which have additional checks enabled. You can install it by running:

```
python3 -m tools.linter.install.clang_tidy
```

We currently only support Linux and MacOS (x86).

2. Install clang-tidy driver script dependencies

```
pip3 install -r tools/linter/clang_tidy/requirements.txt
```

3. Run clang-tidy

```
# Run clang-tidy on the entire codebase
make clang-tidy
# Run clang-tidy only on your changes
make clang-tidy CHANGED_ONLY=--changed-only
```

This internally invokes our driver script and closely mimics how clang-tidy is run on CI.

## Pre-commit tidy/linting hook

We use clang-tidy to perform additional formatting and semantic checking of code. We provide a pre-commit git hook for performing these checks, before a commit is created:

```
ln -s ../../tools/git-pre-commit .git/hooks/pre-commit
```

If you have already committed files and CI reports `flake8` errors, you can run the check locally in your PR branch with:

```
flake8 $(git diff --name-only $(git merge-base --fork-point master))
```

You'll need to install an appropriately configured flake8; see Lint as you type for documentation on how to do this.

Fix the code so that no errors are reported when you re-run the above check again, and then commit the fix.

## Building PyTorch with ASAN

ASAN is very useful for debugging memory errors in C++. We run it in CI, but here's how to get the same thing to run on your local machine.

First, install LLVM 8. The easiest way is to get prebuilt binaries and extract them to folder (later called `$LLVM_ROOT`).

Then set up the appropriate scripts. You can put this in your `.bashrc`:

```
LLVM_ROOT=<wherever your llvm install is>
PYTORCH_ROOT=<wherever your pytorch checkout is>

LIBASAN_RT="$LLVM_ROOT/lib/clang/8.0.0/lib/linux/libclang_rt.asan-x86_64.so"
build_with_asan()
{
  LD_PRELOAD=${LIBASAN_RT} \
  CC="$LLVM_ROOT/bin/clang" \
```

```
  CXX="$LLVM_ROOT/bin/clang++" \
  LDSHARED="clang --shared" \
  LDFLAGS="-stdlib=libstdc++" \
  CFLAGS="-fsanitize=address -fno-sanitize-recover=all -shared-libasan -pthread" \
  CXX_FLAGS="-pthread" \
  USE_CUDA=0 USE_OPENMP=0 BUILD_CAFFE2_OPS=0 USE_DISTRIBUTED=0 DEBUG=1 \
  python setup.py develop
}

run_with_asan()
{
  LD_PRELOAD=${LIBASAN_RT} $@
}

# you can look at build-asan.sh to find the latest options the CI uses
export ASAN_OPTIONS=detect_leaks=0:symbolize=1:strict_init_order=true
export UBSAN_OPTIONS=print_stacktrace=1:suppressions=$PYTORCH_ROOT/ubsan.supp
export ASAN_SYMBOLIZER_PATH=$LLVM_ROOT/bin/llvm-symbolizer
```

Then you can use the scripts like:

```
suo-devfair ~/pytorch ❯ build_with_asan
suo-devfair ~/pytorch ❯ run_with_asan python test/test_jit.py
```

### Getting `ccache` to work

The scripts above specify the `clang` and `clang++` binaries directly, which bypasses `ccache`. Here's how to get `ccache` to work:

1. Make sure the ccache symlinks for `clang` and `clang++` are set up (see CONTRIBUTING.md)
2. Make sure `$LLVM_ROOT/bin` is available on your `$PATH`.
3. Change the `CC` and `CXX` variables in `build_with_asan()` to point directly to `clang` and `clang++`.

### Why this stuff with `LD_PRELOAD` and `LIBASAN_RT`?

The "standard" workflow for ASAN assumes you have a standalone binary:

1. Recompile your binary with `-fsanitize=address`.
2. Run the binary, and ASAN will report whatever errors it find.

Unfortunately, PyTorch is a distributed as a shared library that is loaded by a third-party executable (Python). It's too much of a hassle to recompile all of Python every time we want to use ASAN. Luckily, the ASAN folks have a workaround for cases like this:

1. Recompile your library with `-fsanitize=address -shared-libasan`. The extra `-shared-libasan` tells the compiler to ask for the shared ASAN runtime library.
2. Use `LD_PRELOAD` to tell the dynamic linker to load the ASAN runtime library before anything else.

More information can be found here.

### Why LD_PRELOAD in the build function?

We need `LD_PRELOAD` because there is a cmake check that ensures that a simple program builds and runs. If we are building with ASAN as a shared library, we need to `LD_PRELOAD` the runtime library, otherwise there will dynamic linker errors and the check will fail.

We don't actually need either of these if we fix the cmake checks.

### Why no leak detection?

Python leaks a lot of memory. Possibly we could configure a suppression file, but we haven't gotten around to it.

## Caffe2 notes

In 2018, we merged Caffe2 into the PyTorch source repository. While the steady state aspiration is that Caffe2 and PyTorch share code freely, in the meantime there will be some separation.

If you submit a PR to only PyTorch or only Caffe2 code, CI will only run for the project you edited. The logic for this is implemented in `.jenkins/pytorch/dirty.sh` and `.jenkins/caffe2/dirty.sh`; you can look at this to see what path prefixes constitute changes. This also means if you ADD a new top-level path, or you start sharing code between projects, you need to modify these files.

There are a few "unusual" directories which, for historical reasons, are Caffe2/PyTorch specific. Here they are:

- `CMakeLists.txt`, `Makefile`, `binaries`, `cmake`, `conda`, `modules`, `scripts` are Caffe2-specific. Don't put PyTorch code in them without extra coordination.
- `mypy*`, `requirements.txt`, `setup.py`, `test`, `tools` are PyTorch-specific. Don't put Caffe2 code in them without extra coordination.

## CI failure tips

Once you submit a PR or push a new commit to a branch that is in an active PR, CI jobs will be run automatically. Some of these may fail and you will need to find out why, by looking at the logs.

Fairly often, a CI failure might be unrelated to your changes. You can confirm by going to our HUD and seeing if the CI job is failing upstream already. In this case, you can usually ignore the failure. See the following subsection for more details.

Some failures might be related to specific hardware or environment configurations. In this case, if you're a Meta employee, you can ssh into the job's session to perform manual debugging following the instructions in our CI wiki.

### Which commit is used in CI?

For CI run on `master`, this repository is checked out for a given `master` commit, and CI is run on that commit (there isn't really any other choice). For PRs, however, it's a bit more complicated. Consider this commit graph, where `master` is at commit `A`, and the branch for PR #42 (just a placeholder) is at commit `B`:

```
      o---o---B  (refs/pull/42/head)
     /         \
    /           C  (refs/pull/42/merge)
   /           /
---o---o---o---A  (refs/heads/master)
```

There are two possible choices for which commit to use:

1. Checkout commit `B`, the head of the PR (manually committed by the PR author).
2. Checkout commit `C`, the hypothetical result of what would happen if the PR were merged into `master` (automatically generated by GitHub).

This choice depends on several factors; here is the decision tree as of 2021-03-30:

- For CI jobs on CircleCI:
  - If the name of the job (or one of its ancestors in the workflow DAG) contains "xla" or "gcc5", choice **2** is used. This includes the following jobs:
    - pytorch_linux_xenial_py3_6_gcc5_4_build
      - pytorch_cpp_doc_build
      - pytorch_doc_test
      - pytorch_linux_forward_backward_compatibility_check_test
      - pytorch_linux_xenial_py3_6_gcc5_4_jit_legacy_test
      - pytorch_linux_xenial_py3_6_gcc5_4_test
      - pytorch_python_doc_build
    - pytorch_xla_linux_bionic_py3_6_clang9_build
      - pytorch_xla_linux_bionic_py3_6_clang9_test
  - Otherwise, choice **1** is used.
- For CI jobs on GitHub Actions:
  - If the PR was created using `ghstack`, choice **1** is used.
  - Otherwise, choice **2** is used.

This is important to be aware of, because if you see a CI failure on your PR and choice **2** is being used for that CI job, it is possible that the failure is nondeterministically caused by a commit that does not exist in the ancestry of your PR branch. If you happen to have write access to this repo, you can choose to use `ghstack` to eliminate this nondeterminism for GitHub Actions jobs on your PRs, but it will still be present for the select CircleCI jobs listed above.