Contributing guidelines

Pull Request Checklist

Before sending your pull requests, make sure you do the following:

- Read the contributing guidelines.
- Read the Code of Conduct.
- Ensure you have signed the Contributor License Agreement (CLA).
- Check if your changes are consistent with the guidelines.
- Changes are consistent with the Coding Style.
- Run the unit tests.

How to become a contributor and submit your own code

Contributor License Agreements

We'd love to accept your patches! Before we can take them, we have to jump a couple of legal hurdles.

Please fill out either the individual or corporate Contributor License Agreement (CLA).

- If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an individual CLA.
- If you work for a company that wants to allow you to contribute your work, then you'll need to sign a corporate CLA.

Follow either of the two links above to access the appropriate CLA and instructions for how to sign and return it. Once we receive it, we'll be able to accept your pull requests.

NOTE: Only original source code from you and other people that have signed the CLA can be accepted into the main repository.

Contributing code

If you have improvements to TensorFlow, send us your pull requests! For those just getting started, Github has a how to.

TensorFlow team members will be assigned to review your pull requests. Once the pull requests are approved and pass continuous integration checks, a TensorFlow team member will apply `ready to pull` label to your change. This means we are working on getting your pull request submitted to our internal repository. After the change has been submitted internally, your pull request will be merged automatically on GitHub.

If you want to contribute, start working through the TensorFlow codebase, navigate to the Github "issues" tab and start looking through interesting issues. If you are not sure of where to start, then start by trying one of the smaller/easier issues here i.e. issues with the "good first issue" label and then take a look at the issues with the "contributions welcome" label. These are issues that we believe are particularly well suited for outside contributions, often because we probably won't get to them right now. If you decide to start on an issue, leave a comment so that other people know that you're working on it. If you want to help out, but not alone, use the issue comment thread to coordinate.

Contribution guidelines and standards

Before sending your pull request for review, make sure your changes are consistent with the guidelines and follow the TensorFlow coding style.

General guidelines and philosophy for contribution

- Include unit tests when you contribute new features, as they help to a) prove that your code works correctly, and b) guard against future breaking changes to lower the maintenance cost.
- Bug fixes also generally require unit tests, because the presence of bugs usually indicates insufficient test coverage.
- Keep API compatibility in mind when you change code in core TensorFlow, e.g., code in tensorflow/core and tensorflow/python. TensorFlow has passed version 1.0 and hence cannot make non-backward-compatible API changes without a major release. Reviewers of your pull request will comment on any API compatibility issues following API review practices.
- When you contribute a new feature to TensorFlow, the maintenance burden is (by default) transferred to the TensorFlow team. This means that the benefit of the contribution must be compared against the cost of maintaining the feature.
- Full new features (e.g., a new op implementing a cutting-edge algorithm) typically will live in tensorflow/addons to get some airtime before a decision is made regarding whether they are to be migrated to the core.
- As every PR requires several CPU/GPU hours of CI testing, we discourage submitting PRs to fix one typo, one warning,etc. We recommend fixing the same issue at the file level at least (e.g.: fix all typos in a file, fix all compiler warning in a file, etc.)
- Tests should follow the testing best practices guide.

License

Include a license at the top of new files.

- C/C++ license example
- Python license example
- Java license example
- Go license example
- Bash license example
- HTML license example
- JavaScript/TypeScript license example

Bazel BUILD files also need to include a license section, e.g., BUILD example.

C++ coding style

Changes to TensorFlow C++ code should conform to Google C++ Style Guide.

Use `clang-tidy` to check your C/C++ changes. To install `clang-tidy` on ubuntu:16.04, do:

```
apt-get install -y clang-tidy
```

You can check a C/C++ file by doing:

```
clang-format <my_cc_file> --style=google > /tmp/my_cc_file.cc
diff <my_cc_file> /tmp/my_cc_file.cc
```

Python coding style

Changes to TensorFlow Python code should conform to Google Python Style Guide

Use `pylint` to check your Python changes. To install `pylint` and check a file with `pylint` against TensorFlow's custom style definition:

```
pip install pylint
pylint --rcfile=tensorflow/tools/ci_build/pylintrc myfile.py
```

Note `pylint --rcfile=tensorflow/tools/ci_build/pylintrc` should run from the top level tensorflow directory.

Coding style for other languages

- Google Java Style Guide
- Google JavaScript Style Guide
- Google Shell Style Guide
- Google Objective-C Style Guide

Running sanity check

If you have Docker installed on your system, you can perform a sanity check on your changes by running the command:

```
tensorflow/tools/ci_build/ci_build.sh CPU tensorflow/tools/ci_build/ci_sanity.sh
```

This will catch most license, Python coding style and BUILD file issues that may exist in your changes.

Running unit tests

There are two ways to run TensorFlow unit tests.

1. Using tools and libraries installed directly on your system.

   Refer to the CPU-only developer Dockerfile and GPU developer Dockerfile for the required packages. Alternatively, use the said Docker images, e.g., `tensorflow/tensorflow:devel` and `tensorflow/tensorflow:devel-gpu` for development to avoid installing the packages directly on your system (in which case remember to change directory from `/root` to `/tensorflow` once you get into the running container so `bazel` can find the `tensorflow` workspace).

   Once you have the packages installed, you can run a specific unit test in bazel by doing as follows:

   ```
   export flags="--config=opt -k"
   ```

   If the tests are to be run on GPU, add CUDA paths to LD_LIBRARY_PATH and add the `cuda` option flag

   ```
   export
   LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:/usr/local/cuda/lib64:/usr/local/cuda/extras


   export flags="--config=opt --config=cuda -k"
   ```

   For example, to run all tests under tensorflow/python, do:

   ```
   bazel test ${flags} //tensorflow/python/...
   ```

   For a single component e.g. softmax op:

```
bazel test ${flags} tensorflow/python/kernel_tests:softmax_op_test
```

For a single/parameterized test e.g. `test_capture_variables` in `tensorflow/python/saved_model/load_test.py`:

(Requires `python>=3.7`)

```
bazel test ${flags} //tensorflow/python/saved_model:load_test --
test_filter=*LoadTest.test_capture_variables*
```

Note: You can add `--test_sharding_strategy=disabled` to the `flags` to disable the sharding so that all the test outputs are in one file. However, it may slow down the tests for not running in parallel and may cause the test to timeout but it could be useful when you need to execute a single test or more in general your filtered/selected tests have a very low execution time and the sharding could create an overhead on the test execution.

2. Using Docker and TensorFlow's CI scripts.

```
# Install Docker first, then this will build and run cpu tests
tensorflow/tools/ci_build/ci_build.sh CPU bazel test //tensorflow/...
```

See TensorFlow Builds for details.

Running doctest for testable docstring

There are two ways to test the code in the docstring locally:

1. If you are only changing the docstring of a class/function/method, then you can test it by passing that file's path to tf_doctest.py. For example:

```
python tf_doctest.py --file=<file_path>
```

This will run it using your installed version of TensorFlow. To be sure you're running the same code that you're testing:

- Use an up to date tf-nightly `pip install -U tf-nightly`
- Rebase your pull request onto a recent pull from TensorFlow's master branch.

2. If you are changing the code and the docstring of a class/function/method, then you will need to build TensorFlow from source. Once you are setup to build from source, you can run the tests:

```
bazel run //tensorflow/tools/docs:tf_doctest
```

or

```
bazel run //tensorflow/tools/docs:tf_doctest -- --module=ops.array_ops
```

The `--module` is relative to `tensorflow.python`.

Debug builds

When building Tensorflow, passing `--config=dbg` to Bazel will build with debugging information and without optimizations, allowing you to use GDB or other debuggers to debug C++ code. For example, you can build the pip package with debugging information by running:

```
bazel build --config=dbg //tensorflow/tools/pip_package:build_pip_package
```

TensorFlow kernels and TensorFlow's dependencies are still not built with debugging information with `--config=dbg`, as issues occur on Linux if there is too much debug info (see this GitHub issue for context). If you want to debug a kernel, you can compile specific files with `-g` using the `--per_file_copt` bazel option. For example, if you want to debug the Identity op, which are in files starting with `identity_op`, you can run

```
bazel build --config=dbg --per_file_copt=+tensorflow/core/kernels/identity_op.*@-g
//tensorflow/tools/pip_package:build_pip_package
```

Note that the `--config=dbg` option is not officially supported.