Multiprocessing best practices

:mod:`torch.multiprocessing` is a drop in replacement for Python's :mod:`python:multiprocessing` module. It supports the exact same operations, but extends it, so that all tensors sent through a :class:`python:multiprocessing.Queue`, will have their data moved into shared memory and will only send a handle to another process.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 6); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 6); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 6); *backlink*

Unknown interpreted text role "class".

Note

When a :class:`~torch.Tensor` is sent to another process, the :class:`~torch.Tensor` data is shared. If :attr:`torch.Tensor.grad` is not `None`, it is also shared. After a :class:`~torch.Tensor` without a :attr:`torch.Tensor.grad` field is sent to the other process, it creates a standard process-specific `.grad` :class:`~torch.Tensor` that is not automatically shared across all processes, unlike how the :class:`~torch.Tensor`'s data has been shared.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 14); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 14); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 14); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 14); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 14); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 14); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 14); *backlink*

Unknown interpreted text role "class".

This allows to implement various training methods, like Hogwild, A3C, or any others that require asynchronous operation.

CUDA in multiprocessing

The CUDA runtime does not support the `fork` start method; either the `spawn` or `forkserver` start method are required to use CUDA in subprocesses.

Note

The start method can be set via either creating a context with `multiprocessing.get_context(...)` or directly using `multiprocessing.set_start_method(...)`.

Unlike CPU tensors, the sending process is required to keep the original tensor as long as the receiving process retains a copy of the tensor. It is implemented under the hood but requires users to follow the best practices for the program to run correctly. For example, the sending process must stay alive as long as the consumer process has references to the tensor, and the refcounting can not save you if the consumer process exits abnormally via a fatal signal. See :ref:`this section <multiprocessing-cuda-sharing-details>`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 38); *backlink*

Unknown interpreted text role "ref".

See also: :ref:`cuda-nn-ddp-instead`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 46); *backlink*

Unknown interpreted text role "ref".

Best practices and tips

Avoiding and fighting deadlocks

There are a lot of things that can go wrong when a new process is spawned, with the most common cause of deadlocks being background threads. If there's any thread that holds a lock or imports a module, and `fork` is called, it's very likely that the subprocess will be in a corrupted state and will deadlock or fail in a different way. Note that even if you don't, Python built in libraries do - no need to look further than :mod:`python:multiprocessing`. :class:`python:multiprocessing.Queue` is actually a very complex class, that spawns multiple threads used to serialize, send and receive objects, and they can cause aforementioned problems too. If you find yourself in such situation try using a :class:`~python:multiprocessing.queues.SimpleQueue`, that doesn't use any additional threads.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 55); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line

55); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 55); *backlink*

Unknown interpreted text role "class".

We're trying our best to make it easy for you and ensure these deadlocks don't happen but some things are out of our control. If you have any issues you can't cope with for a while, try reaching out on forums, and we'll see if it's an issue we can fix.

Reuse buffers passed through a Queue

Remember that each time you put a :class:`~torch.Tensor` into a :class:`python:multiprocessing.Queue`, it has to be moved into shared memory. If it's already shared, it is a no-op, otherwise it will incur an additional memory copy that can slow down the whole process. Even if you have a pool of processes sending data to a single one, make it send the buffers back - this is nearly free and will let you avoid a copy when sending next batch.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 75); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 75); *backlink*

Unknown interpreted text role "class".

Asynchronous multiprocess training (e.g. Hogwild)

Using :mod:`torch.multiprocessing`, it is possible to train a model asynchronously, with parameters either shared all the time, or being periodically synchronized. In the first case, we recommend sending over the whole model object, while in the latter, we advise to only send the :meth:`~torch.nn.Module.state_dict`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 85); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 85); *backlink*

Unknown interpreted text role "meth".

We recommend using :class:`python:multiprocessing.Queue` for passing all kinds of PyTorch objects between processes. It is possible to e.g. inherit the tensors and storages already in shared memory, when using the `fork` start method, however it is very bug prone and should be used with care, and only by advanced users. Queues, even though they're sometimes a less elegant solution, will work properly in all cases.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]multiprocessing.rst`, line 91); *backlink*

Unknown interpreted text role "class".

Warning

You should be careful about having global statements, that are not guarded with an `if __name__ == '__main__'`. If a different start method than `fork` is used, they will be executed in all subprocesses.

Hogwild

A concrete Hogwild implementation can be found in the examples repository, but to showcase the overall structure of the code, there's also a minimal example below as well:

```
import torch.multiprocessing as mp
from model import MyModel

def train(model):
    # Construct data_loader, optimizer, etc.
    for data, labels in data_loader:
        optimizer.zero_grad()
        loss_fn(model(data), labels).backward()
        optimizer.step()  # This will update the shared parameters

if __name__ == '__main__':
    num_processes = 4
    model = MyModel()
    # NOTE: this is required for the ``fork`` method to work
    model.share_memory()
    processes = []
    for rank in range(num_processes):
        p = mp.Process(target=train, args=(model,))
        p.start()
        processes.append(p)
    for p in processes:
        p.join()
```