Reproducibility

Completely reproducible results are not guaranteed across PyTorch releases, individual commits, or different platforms. Furthermore, results may not be reproducible between CPU and GPU executions, even when using identical seeds.

However, there are some steps you can take to limit the number of sources of nondeterministic behavior for a specific platform, device, and PyTorch release. First, you can control sources of randomness that can cause multiple executions of your application to behave differently. Second, you can configure PyTorch to avoid using nondeterministic algorithms for some operations, so that multiple calls to those operations, given the same inputs, will produce the same result.

> **Warning**
>
> Deterministic operations are often slower than nondeterministic operations, so single-run performance may decrease for your model. However, determinism may save time in development by facilitating experimentation, debugging, and regression testing.

Controlling sources of randomness

PyTorch random number generator

You can use :meth:`torch.manual_seed()` to seed the RNG for all devices (both CPU and CUDA):

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]randomness.rst`, line 29); *backlink***
>
> Unknown interpreted text role "meth".

```
import torch
torch.manual_seed(0)
```

Python

For custom operators, you might need to set python seed as well:

```
import random
random.seed(0)
```

Random number generators in other libraries

If you or any of the libraries you are using rely on NumPy, you can seed the global NumPy RNG with:

```
import numpy as np
np.random.seed(0)
```

However, some applications and libraries may use NumPy Random Generator objects, not the global RNG (https://numpy.org/doc/stable/reference/random/generator.html), and those will need to be seeded consistently as well.

If you are using any other libraries that use random number generators, refer to the documentation for those libraries to see how to set consistent seeds for them.

CUDA convolution benchmarking

The cuDNN library, used by CUDA convolution operations, can be a source of nondeterminism across multiple executions of an application. When a cuDNN convolution is called with a new set of size parameters, an optional feature can run multiple convolution algorithms, benchmarking them to find the fastest one. Then, the fastest algorithm will be used consistently during the rest of the process for the corresponding set of size parameters. Due to benchmarking noise and different hardware, the benchmark may select different algorithms on subsequent runs, even on the same machine.

Disabling the benchmarking feature with `torch.backends.cudnn.benchmark = False` causes cuDNN to deterministically select an algorithm, possibly at the cost of reduced performance.

However, if you do not need reproducibility across multiple executions of your application, then performance might improve if the benchmarking feature is enabled with `torch.backends.cudnn.benchmark = True`.

Note that this setting is different from the `torch.backends.cudnn.deterministic` setting discussed below.

Avoiding nondeterministic algorithms

:meth:`torch.use_deterministic_algorithms` lets you configure PyTorch to use deterministic algorithms instead of nondeterministic ones where available, and to throw an error if an operation is known to be nondeterministic (and without a deterministic alternative).

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]randomness.rst`, line 82); ***backlink***
>
> Unknown interpreted text role "meth".

Please check the documentation for :meth:`torch.use_deterministic_algorithms()` for a full list of affected operations. If an operation does not act correctly according to the documentation, or if you need a deterministic implementation of an operation that does not have one, please submit an issue: https://github.com/pytorch/pytorch/issues?q=label:%22topic:%20determinism%22

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]randomness.rst`, line 87); ***backlink***
>
> Unknown interpreted text role "meth".

For example, running the nondeterministic CUDA implementation of :meth:`torch.Tensor.index_add_` will throw an error:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]randomness.rst`, line 93); ***backlink***
>
> Unknown interpreted text role "meth".

```
>>> import torch
>>> torch.use_deterministic_algorithms(True)
>>> torch.randn(2, 2).cuda().index_add_(0, torch.tensor([0, 1]), torch.randn(2, 2))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
RuntimeError: index_add_cuda_ does not have a deterministic implementation, but you set
'torch.use_deterministic_algorithms(True)'. ...
```

When :meth:`torch.bmm` is called with sparse-dense CUDA tensors it typically uses a nondeterministic algorithm, but when the deterministic flag is turned on, its alternate deterministic implementation will be used:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]randomness.rst`, line 104); ***backlink***
>
> Unknown interpreted text role "meth".

```
>>> import torch
>>> torch.use_deterministic_algorithms(True)
>>> torch.bmm(torch.randn(2, 2, 2).to_sparse().cuda(), torch.randn(2, 2, 2).cuda())
tensor([[[ 1.1900, -2.3409],
         [ 0.4796,  0.8003]],
        [[ 0.1509,  1.8027],
         [ 0.0333, -1.1444]]], device='cuda:0')
```

Furthermore, if you are using CUDA tensors, and your CUDA version is 10.2 or greater, you should set the environment variable *CUBLAS_WORKSPACE_CONFIG* according to CUDA documentation: https://docs.nvidia.com/cuda/cublas/index.html#cublasApi_reproducibility

CUDA convolution determinism

While disabling CUDA convolution benchmarking (discussed above) ensures that CUDA selects the same algorithm each time an application is run, that algorithm itself may be nondeterministic, unless either `torch.use_deterministic_algorithms(True)` or `torch.backends.cudnn.deterministic = True` is set. The latter setting controls only this behavior, unlike :meth:`torch.use_deterministic_algorithms` which will make other PyTorch operations behave deterministically, too.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]randomness.rst`, line 122); ***backlink***
>
> Unknown interpreted text role "meth".

CUDA RNN and LSTM

In some versions of CUDA, RNNs and LSTM networks may have non-deterministic behavior. See :meth:`torch.nn.RNN` and :meth:`torch.nn.LSTM` for details and workarounds.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]randomness.rst`, line 132); ***backlink***

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]randomness.rst`, line 132); ***backlink***

Unknown interpreted text role "meth".

DataLoader

DataLoader will reseed workers following :ref:`data-loading-randomness` algorithm. Use :meth:`worker_init_fn` and *generator* to preserve reproducibility:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]randomness.rst`, line 138); ***backlink***

Unknown interpreted text role "ref".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]randomness.rst`, line 138); ***backlink***

Unknown interpreted text role "meth".

```
def seed_worker(worker_id):
    worker_seed = torch.initial_seed() % 2**32
    numpy.random.seed(worker_seed)
    random.seed(worker_seed)

g = torch.Generator()
g.manual_seed(0)

DataLoader(
    train_dataset,
    batch_size=batch_size,
    num_workers=num_workers,
    worker_init_fn=seed_worker,
    generator=g,
)
```