torch.utils.data — PyTorch 2.4 documentation

pytorch.org/docs/stable/data.html

torch.utils.data

At the heart of PyTorch data loading utility is the `torch.utils.data.DataLoader` class. It represents a Python iterable over a dataset, with support for

- map-style and iterable-style datasets,
- customizing data loading order,
- automatic batching,
- single- and multi-process data loading,
- automatic memory pinning.

These options are configured by the constructor arguments of a `DataLoader`, which has signature:

```
DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
           batch_sampler=None, num_workers=0, collate_fn=None,
           pin_memory=False, drop_last=False, timeout=0,
           worker_init_fn=None, *, prefetch_factor=2,
           persistent_workers=False)
```

The sections below describe in details the effects and usages of these options.

Dataset Types

The most important argument of `DataLoader` constructor is `dataset`, which indicates a dataset object to load data from. PyTorch supports two different types of datasets:

- map-style datasets,
- iterable-style datasets.

Map-style datasets

A map-style dataset is one that implements the `__getitem__()` and `__len__()` protocols, and represents a map from (possibly non-integral) indices/keys to data samples.

For example, such a dataset, when accessed with `dataset[idx]`, could read the `idx`-th image and its corresponding label from a folder on the disk.

See `Dataset` for more details.

Iterable-style datasets

An iterable-style dataset is an instance of a subclass of `IterableDataset` that implements the `__iter__()` protocol, and represents an iterable over data samples. This type of datasets is particularly suitable for cases where random reads are expensive or even improbable, and where the batch size depends on the fetched data.

For example, such a dataset, when called `iter(dataset)`, could return a stream of data reading from a database, a remote server, or even logs generated in real time.

See `IterableDataset` for more details.

Note

When using a `IterableDataset` with multi-process data loading. The same dataset object is replicated on each worker process, and thus the replicas must be configured differently to avoid duplicated data. See `IterableDataset` documentations for how to achieve this.

Data Loading Order and `Sampler`

For iterable-style datasets, data loading order is entirely controlled by the user-defined iterable. This allows easier implementations of chunk-reading and dynamic batch size (e.g., by yielding a batched sample at each time).

The rest of this section concerns the case with map-style datasets. `torch.utils.data.Sampler` classes are used to specify the sequence of indices/keys used in data loading. They represent iterable objects over the indices to datasets. E.g., in the common case with stochastic gradient decent (SGD), a `Sampler` could randomly permute a list of indices and yield each one at a time, or yield a small number of them for mini-batch SGD.

A sequential or shuffled sampler will be automatically constructed based on the `shuffle` argument to a `DataLoader`. Alternatively, users may use the `sampler` argument to specify a custom `Sampler` object that at each time yields the next index/key to fetch.

A custom `Sampler` that yields a list of batch indices at a time can be passed as the `batch_sampler` argument. Automatic batching can also be enabled via `batch_size` and `drop_last` arguments. See the next section for more details on this.

Note

Neither `sampler` nor `batch_sampler` is compatible with iterable-style datasets, since such datasets have no notion of a key or an index.

Loading Batched and Non-Batched Data

`DataLoader` supports automatically collating individual fetched data samples into batches via arguments `batch_size`, `drop_last`, `batch_sampler`, and `collate_fn` (which has a default function).

Automatic batching (default)

This is the most common case, and corresponds to fetching a minibatch of data and collating them into batched samples, i.e., containing Tensors with one dimension being the batch dimension (usually the first).

When `batch_size` (default `1`) is not `None`, the data loader yields batched samples instead of individual samples. `batch_size` and `drop_last` arguments are used to specify how the data loader obtains batches of dataset keys. For map-style datasets, users can alternatively specify `batch_sampler`, which yields a list of keys at a time.

Note

The `batch_size` and `drop_last` arguments essentially are used to construct a `batch_sampler` from `sampler`. For map-style datasets, the `sampler` is either provided by user or constructed based on the `shuffle` argument. For iterable-style datasets, the `sampler` is a dummy infinite one. See this section on more details on samplers.

Note

When fetching from iterable-style datasets with multi-processing, the `drop_last` argument drops the last non-full batch of each worker's dataset replica.

After fetching a list of samples using the indices from sampler, the function passed as the `collate_fn` argument is used to collate lists of samples into batches.

In this case, loading from a map-style dataset is roughly equivalent with:

```
for indices in batch_sampler:
    yield collate_fn([dataset[i] for i in indices])
```

and loading from an iterable-style dataset is roughly equivalent with:

```
dataset_iter = iter(dataset)
for indices in batch_sampler:
    yield collate_fn([next(dataset_iter) for _ in indices])
```

A custom `collate_fn` can be used to customize collation, e.g., padding sequential data to max length of a batch. See this section on more about `collate_fn`.

Disable automatic batching

In certain cases, users may want to handle batching manually in dataset code, or simply load individual samples. For example, it could be cheaper to directly load batched data (e.g., bulk reads from a database or reading continuous chunks of memory), or the batch size is data dependent, or the program is designed to work on individual samples. Under

these scenarios, it's likely better to not use automatic batching (where `collate_fn` is used to collate the samples), but let the data loader directly return each member of the `dataset` object.

When both `batch_size` and `batch_sampler` are `None` (default value for `batch_sampler` is already `None`), automatic batching is disabled. Each sample obtained from the `dataset` is processed with the function passed as the `collate_fn` argument.

When automatic batching is disabled, the default `collate_fn` simply converts NumPy arrays into PyTorch Tensors, and keeps everything else untouched.

In this case, loading from a map-style dataset is roughly equivalent with:

```
for index in sampler:
    yield collate_fn(dataset[index])
```

and loading from an iterable-style dataset is roughly equivalent with:

```
for data in iter(dataset):
    yield collate_fn(data)
```

See this section on more about `collate_fn`.

Working with `collate_fn`

The use of `collate_fn` is slightly different when automatic batching is enabled or disabled.

When automatic batching is disabled, `collate_fn` is called with each individual data sample, and the output is yielded from the data loader iterator. In this case, the default `collate_fn` simply converts NumPy arrays in PyTorch tensors.

When automatic batching is enabled, `collate_fn` is called with a list of data samples at each time. It is expected to collate the input samples into a batch for yielding from the data loader iterator. The rest of this section describes the behavior of the default `collate_fn` (`default_collate()`).

For instance, if each data sample consists of a 3-channel image and an integral class label, i.e., each element of the dataset returns a tuple `(image, class_index)`, the default `collate_fn` collates a list of such tuples into a single tuple of a batched image tensor and a batched class label Tensor. In particular, the default `collate_fn` has the following properties:

- It always prepends a new dimension as the batch dimension.
- It automatically converts NumPy arrays and Python numerical values into PyTorch Tensors.

- It preserves the data structure, e.g., if each sample is a dictionary, it outputs a dictionary with the same set of keys but batched Tensors as values (or lists if the values can not be converted into Tensors). Same for `list` s, `tuple` s, `namedtuple` s, etc.

Users may use customized `collate_fn` to achieve custom batching, e.g., collating along a dimension other than the first, padding sequences of various lengths, or adding support for custom data types.

If you run into a situation where the outputs of `DataLoader` have dimensions or type that is different from your expectation, you may want to check your `collate_fn`.

Single- and Multi-process Data Loading

A `DataLoader` uses single-process data loading by default.

Within a Python process, the Global Interpreter Lock (GIL) prevents true fully parallelizing Python code across threads. To avoid blocking computation code with data loading, PyTorch provides an easy switch to perform multi-process data loading by simply setting the argument `num_workers` to a positive integer.

Single-process data loading (default)

In this mode, data fetching is done in the same process a `DataLoader` is initialized. Therefore, data loading may block computing. However, this mode may be preferred when resource(s) used for sharing data among processes (e.g., shared memory, file descriptors) is limited, or when the entire dataset is small and can be loaded entirely in memory. Additionally, single-process loading often shows more readable error traces and thus is useful for debugging.

Multi-process data loading

Setting the argument `num_workers` as a positive integer will turn on multi-process data loading with the specified number of loader worker processes.

Warning

After several iterations, the loader worker processes will consume the same amount of CPU memory as the parent process for all Python objects in the parent process which are accessed from the worker processes. This can be problematic if the Dataset contains a lot of data (e.g., you are loading a very large list of filenames at Dataset construction time) and/or you are using a lot of workers (overall memory usage is `number of workers * size of parent process`). The simplest workaround is to replace Python objects with non-refcounted representations such as Pandas, Numpy or PyArrow objects. Check out issue #13246 for more details on why this occurs and example code for how to workaround these problems.

In this mode, each time an iterator of a `DataLoader` is created (e.g., when you call `enumerate(dataloader)`), `num_workers` worker processes are created. At this point, the `dataset`, `collate_fn`, and `worker_init_fn` are passed to each worker, where they are used to initialize, and fetch data. This means that dataset access together with its internal IO, transforms (including `collate_fn`) runs in the worker process.

`torch.utils.data.get_worker_info()` returns various useful information in a worker process (including the worker id, dataset replica, initial seed, etc.), and returns `None` in main process. Users may use this function in dataset code and/or `worker_init_fn` to individually configure each dataset replica, and to determine whether the code is running in a worker process. For example, this can be particularly helpful in sharding the dataset.

For map-style datasets, the main process generates the indices using `sampler` and sends them to the workers. So any shuffle randomization is done in the main process which guides loading by assigning indices to load.

For iterable-style datasets, since each worker process gets a replica of the `dataset` object, naive multi-process loading will often result in duplicated data. Using `torch.utils.data.get_worker_info()` and/or `worker_init_fn`, users may configure each replica independently. (See `IterableDataset` documentations for how to achieve this.) For similar reasons, in multi-process loading, the `drop_last` argument drops the last non-full batch of each worker's iterable-style dataset replica.

Workers are shut down once the end of the iteration is reached, or when the iterator becomes garbage collected.

Warning

It is generally not recommended to return CUDA tensors in multi-process loading because of many subtleties in using CUDA and sharing CUDA tensors in multiprocessing (see CUDA in multiprocessing). Instead, we recommend using automatic memory pinning (i.e., setting `pin_memory=True`), which enables fast data transfer to CUDA-enabled GPUs.

Platform-specific behaviors

Since workers rely on Python `multiprocessing`, worker launch behavior is different on Windows compared to Unix.

- On Unix, `fork()` is the default `multiprocessing` start method. Using `fork()`, child workers typically can access the `dataset` and Python argument functions directly through the cloned address space.
- On Windows or MacOS, `spawn()` is the default `multiprocessing` start method. Using `spawn()`, another interpreter is launched which runs your main script, followed by the internal worker function that receives the `dataset`, `collate_fn` and other arguments through `pickle` serialization.

This separate serialization means that you should take two steps to ensure you are compatible with Windows while using multi-process data loading:

- Wrap most of you main script's code within `if __name__ == '__main__':` block, to make sure it doesn't run again (most likely generating error) when each worker process is launched. You can place your dataset and `DataLoader` instance creation logic here, as it doesn't need to be re-executed in workers.
- Make sure that any custom `collate_fn`, `worker_init_fn` or `dataset` code is declared as top level definitions, outside of the `__main__` check. This ensures that they are available in worker processes. (this is needed since functions are pickled as references only, not `bytecode`.)

Randomness in multi-process data loading

By default, each worker will have its PyTorch seed set to `base_seed + worker_id`, where `base_seed` is a long generated by main process using its RNG (thereby, consuming a RNG state mandatorily) or a specified `generator`. However, seeds for other libraries may be duplicated upon initializing workers, causing each worker to return identical random numbers. (See this section in FAQ.).

In `worker_init_fn`, you may access the PyTorch seed set for each worker with either `torch.utils.data.get_worker_info().seed` or `torch.initial_seed()`, and use it to seed other libraries before data loading.

Memory Pinning

Host to GPU copies are much faster when they originate from pinned (page-locked) memory. See Use pinned memory buffers for more details on when and how to use pinned memory generally.

For data loading, passing `pin_memory=True` to a `DataLoader` will automatically put the fetched data Tensors in pinned memory, and thus enables faster data transfer to CUDA-enabled GPUs.

The default memory pinning logic only recognizes Tensors and maps and iterables containing Tensors. By default, if the pinning logic sees a batch that is a custom type (which will occur if you have a `collate_fn` that returns a custom batch type), or if each element of your batch is a custom type, the pinning logic will not recognize them, and it will return that batch (or those elements) without pinning the memory. To enable memory pinning for custom batch or data type(s), define a `pin_memory()` method on your custom type(s).

See the example below.

Example:

```
class SimpleCustomBatch:
    def __init__(self, data):
        transposed_data = list(zip(*data))
        self.inp = torch.stack(transposed_data[0], 0)
        self.tgt = torch.stack(transposed_data[1], 0)

    # custom memory pinning method on custom type
    def pin_memory(self):
        self.inp = self.inp.pin_memory()
        self.tgt = self.tgt.pin_memory()
        return self

def collate_wrapper(batch):
    return SimpleCustomBatch(batch)

inps = torch.arange(10 * 5, dtype=torch.float32).view(10, 5)
tgts = torch.arange(10 * 5, dtype=torch.float32).view(10, 5)
dataset = TensorDataset(inps, tgts)

loader = DataLoader(dataset, batch_size=2, collate_fn=collate_wrapper,
                    pin_memory=True)

for batch_ndx, sample in enumerate(loader):
    print(sample.inp.is_pinned())
    print(sample.tgt.is_pinned())
```

Data loader combines a dataset and a sampler, and provides an iterable over the given dataset.

The `DataLoader` supports both map-style and iterable-style datasets with single- or multi-process loading, customizing loading order and optional automatic batching (collation) and memory pinning.

See `torch.utils.data` documentation page for more details.

Parameters

- **dataset** (*Dataset*) – dataset from which to load the data.
- **batch_size** (*int, optional*) – how many samples per batch to load (default: `1`).
- **shuffle** (*bool, optional*) – set to `True` to have the data reshuffled at every epoch (default: `False`).
- **sampler** (*Sampler or Iterable, optional*) – defines the strategy to draw samples from the dataset. Can be any `Iterable` with `__len__` implemented. If specified, `shuffle` must not be specified.
- **batch_sampler** (*Sampler or Iterable, optional*) – like `sampler`, but returns a batch of indices at a time. Mutually exclusive with `batch_size`, `shuffle`, `sampler`, and `drop_last`.

- **num_workers** (*int, optional*) – how many subprocesses to use for data loading. `0` means that the data will be loaded in the main process. (default: `0`)
- **collate_fn** (*Callable, optional*) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.
- **pin_memory** (*bool, optional*) – If `True`, the data loader will copy Tensors into device/CUDA pinned memory before returning them. If your data elements are a custom type, or your `collate_fn` returns a batch that is a custom type, see the example below.
- **drop_last** (*bool, optional*) – set to `True` to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If `False` and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: `False`)
- **timeout** (*numeric, optional*) – if positive, the timeout value for collecting a batch from workers. Should always be non-negative. (default: `0`)
- **worker_init_fn** (*Callable, optional*) – If not `None`, this will be called on each worker subprocess with the worker id (an int in `[0, num_workers - 1]`) as input, after seeding and before data loading. (default: `None`)
- **multiprocessing_context** (*str or multiprocessing.context.BaseContext, optional*) – If `None`, the default multiprocessing context of your operating system will be used. (default: `None`)
- **generator** (*torch.Generator, optional*) – If not `None`, this RNG will be used by RandomSampler to generate random indexes and multiprocessing to generate `base_seed` for workers. (default: `None`)
- **prefetch_factor** (*int, optional, keyword-only arg*) – Number of batches loaded in advance by each worker. `2` means there will be a total of 2 * num_workers batches prefetched across all workers. (default value depends on the set value for num_workers. If value of num_workers=0 default is `None`. Otherwise, if value of `num_workers > 0` default is `2`).
- **persistent_workers** (*bool, optional*) – If `True`, the data loader will not shut down the worker processes after a dataset has been consumed once. This allows to maintain the workers *Dataset* instances alive. (default: `False`)
- **pin_memory_device** (*str, optional*) – the device to `pin_memory` to if `pin_memory` is `True`.

Warning

If the `spawn` start method is used, `worker_init_fn` cannot be an unpicklable object, e.g., a lambda function. See Multiprocessing best practices on more details related to multiprocessing in PyTorch.

Warning

`len(dataloader)` heuristic is based on the length of the sampler used. When `dataset` is an `IterableDataset`, it instead returns an estimate based on `len(dataset) / batch_size`, with proper rounding depending on `drop_last`, regardless of multi-process loading configurations. This represents the best guess PyTorch can make because PyTorch trusts user `dataset` code in correctly handling multi-process loading to avoid duplicate data.

However, if sharding results in multiple workers having incomplete last batches, this estimate can still be inaccurate, because (1) an otherwise complete batch can be broken into multiple ones and (2) more than one batch worth of samples can be dropped when `drop_last` is set. Unfortunately, PyTorch can not detect such cases in general.

See Dataset Types for more details on these two types of datasets and how `IterableDataset` interacts with Multi-process data loading.

Warning

See Reproducibility, and My data loader workers return identical random numbers, and Randomness in multi-process data loading notes for random seed related questions.