torch :

Type: module

Docstring:

The torch package contains data structures for multi-dimensional
tensors and defines mathematical operations over these tensors.
Additionally, it provides many utilities for efficient serializing of
Tensors and arbitrary types, and other useful utilities.

It has a CUDA counterpart, that enables you to run your tensor computations
on an NVIDIA GPU with compute capability >= 3.0.

DataLoader:

```
DataLoader(dataset: torch.utils.data.dataset.Dataset[+T_co],
   batch_size: Optional[int] = 1,
   shuffle: bool = False,
   sampler: Optional[torch.utils.data.sampler.Sampler] = None,
   batch_sampler: Optional[torch.utils.data.sampler.Sampler[Sequence]] = None,
   num_workers: int = 0,
   collate_fn: Optional[Callable[[List[~T]], Any]] = None,
   pin_memory: bool = False,
   drop_last: bool = False,
   timeout: float = 0,
   worker_init_fn: Optional[Callable[[int], NoneType]] = None,
   multiprocessing_context=None,
   generator=None,    *,
   prefetch_factor: int = 2,
   persistent_workers: bool = False,)
```

Docstring:

```
Data loader. Combines a dataset and a sampler, and provides an iterable over the given dataset.
The :class:`~torch.utils.data.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 :py:mod:`torch.utils.data` documentation page for more details.
Args:
    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, :attr:`shuffle` must not be specified.
    batch_sampler (Sampler or Iterable, optional): like :attr:`sampler`, but
        returns a batch of indices at a time. Mutually exclusive with
        :attr:`batch_size`, :attr:`shuffle`, :attr:`sampler`,
        and :attr:`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 CUDA pinned memory before returning them.  If your data elements
        are a custom type, or your :attr:`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``)
    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 samples loaded
        in advance by each worker. ``2`` means there will be a total of
        2 * num_workers samples prefetched across all workers. (default: ``2``)
    persistent_workers (bool, optional): If ``True``, the data loader will not shutdown
        the worker processes after a dataset has been consumed once. This allows to
        maintain the workers `Dataset` instances alive. (default: ``False``)


.. warning:: If the ``spawn`` start method is used, :attr:`worker_init_fn`
             cannot be an unpicklable object, e.g., a lambda function. See
             :ref:`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 :attr:`dataset` is an :class:`~torch.utils.data.IterableDataset`,
             it instead returns an estimate based on ``len(dataset) / batch_size``, with proper
             rounding depending on :attr:`drop_last`, regardless of multi-process loading
             configurations. This represents the best guess PyTorch can make because PyTorch
             trusts user :attr:`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 :attr:`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 :class:`~torch.utils.data.IterableDataset` interacts with `Multi-process data loading`_.

.. warning:: See :ref:`reproducibility`, and :ref:`dataloader-workers-random-seed`, and :ref:`data-loading-randomness` notes for random

----

Subset:

```
(dataset: torch.utils.data.dataset.Dataset[+T_co],
   indices: Sequence[int])
```

Docstring:

Subset of a dataset at specified indices.

```
Args:
   dataset (Dataset): The whole Dataset
   indices (sequence): Indices in the whole set selected for subset
```

---

torchvision.transforms.ToTensor()

Docstring:

Convert a ``PIL Image`` or ``numpy.ndarray`` to tensor. This transform does not support torchscript.

Converts a PIL Image or numpy.ndarray (H x W x C) in the range
[0, 255] to a torch.FloatTensor of shape (C x H x W) in the range [0.0, 1.0]
if the PIL Image belongs to one of the modes (L, LA, P, I, F, RGB, YCbCr, RGBA, CMYK, 1)
or if the numpy.ndarray has dtype = np.uint8

In the other cases, tensors are returned without scaling.

note:
    Because the input image is scaled to [0.0, 1.0], this transformation should not be used when
    transforming target image masks. See the `references`_ for implementing the transforms for
image masks.

---

torchvision.transforms.Normalize:

torchvision.transforms.Normalize(mean, std, inplace=False)
Docstring:
Normalize a tensor image with mean and standard deviation.
This transform does not support PIL Image.
Given mean: ``(mean[1],...,mean[n])`` and std: ``(std[1],..,std[n])`` for ``n``
channels, this transform will normalize each channel of the input
``torch.*Tensor`` i.e.,
``output[channel] = (input[channel] - mean[channel]) / std[channel]``

note:
    This transform acts out of place, i.e., it does not mutate the input tensor.

Args:
    mean (sequence): Sequence of means for each channel.
    std (sequence): Sequence of standard deviations for each channel.
    inplace(bool,optional): Bool to make this operation in-place.

---

torchvision.transforms.Compose(transforms)

Docstring:

Composes several transforms together. This transform does not support torchscript.
Please, see the note below.

Args:
    transforms (list of ``Transform`` objects): list of transforms to compose.

Example:
```
    >>> transforms.Compose([
    >>>     transforms.CenterCrop(10),
    >>>     transforms.PILToTensor(),
    >>>     transforms.ConvertImageDtype(torch.float),
    >>> ])
```

note:
In order to script the transformations, please use ``torch.nn.Sequential`` as below.

```
>>> transforms = torch.nn.Sequential(
>>>     transforms.CenterCrop(10),
>>>     transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)),
>>> )
>>> scripted_transforms = torch.jit.script(transforms)
```

Make sure to use only scriptable transformations, i.e. that work with ``torch.Tensor``, does not require
`lambda` functions or ``PIL.Image``.
