Table of Contents

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``.

---

nn.Conv2d

```
(
    in_channels: int,
    out_channels: int,
    kernel_size: Union[int, Tuple[int, int]],
    stride: Union[int, Tuple[int, int]] = 1,
    padding: Union[str, int, Tuple[int, int]] = 0,
    dilation: Union[int, Tuple[int, int]] = 1,
    groups: int = 1,
    bias: bool = True,
    padding_mode: str = 'zeros',
    device=None,
    dtype=None,
) -> None
```

Docstring:

Applies a 2D convolution over an input signal composed of several input
planes.

In the simplest case, the output value of the layer with input size
:math:`(N, C_{\text{in}}, H, W)` and output :math:`(N, C_{\text{out}}, H_{\text{out}},
W_{\text{out}})`

can be precisely described as:

.. math::
    \text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) +
    \sum_{k = 0}^{C_{\text{in}} - 1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)

where :math:`\star` is the valid 2D `cross-correlation`_ operator,
:math:`N` is a batch size, :math:`C` denotes a number of channels,
:math:`H` is a height of input planes in pixels, and :math:`W` is
width in pixels.

This module supports :ref:`TensorFloat32<tf32_on_ampere>`.

* :attr:`stride` controls the stride for the cross-correlation, a single
  number or a tuple.

* :attr:`padding` controls the amount of padding applied to the input. It
  can be either a string {'valid', 'same'} or a tuple of ints giving the
  amount of implicit padding applied on both sides.

* :attr:`dilation` controls the spacing between the kernel points; also
  known as the à trous algorithm. It is harder to describe, but this `link`_
  has a nice visualization of what :attr:`dilation` does.

* :attr:`groups` controls the connections between inputs and outputs.
  :attr:`in_channels` and :attr:`out_channels` must both be divisible by
  :attr:`groups`. For example,

    * At groups=1, all inputs are convolved to all outputs.
    * At groups=2, the operation becomes equivalent to having two conv
      layers side by side, each seeing half the input channels
      and producing half the output channels, and both subsequently
      concatenated.
    * At groups= :attr:`in_channels`, each input channel is convolved with
      its own set of filters (of size
      :math:`\frac{\text{out\_channels}}{\text{in\_channels}}`).

The parameters :attr:`kernel_size`, :attr:`stride`, :attr:`padding`, :attr:`dilation` can either be:

    - a single ``int`` -- in which case the same value is used for the height and width dimension
    - a ``tuple`` of two ints -- in which case, the first `int` is used for the height dimension,
      and the second `int` for the width dimension

Note:
When `groups == in_channels` and `out_channels == K * in_channels`,
where `K` is a positive integer, this operation is also known as a "depthwise convolution".

In other words, for an input of size :math:`(N, C_{in}, L_{in})`,
a depthwise convolution with a depthwise multiplier `K` can be performed with the arguments
:math:`(C_\text{in}=C_\text{in}, C_\text{out}=C_\text{in} \times \text{K}, ..., \text{groups}=C_\text{in})`.

Note:
In some circumstances when given tensors on a CUDA device and using CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting ``torch.backends.cudnn.deterministic = True``. See :doc:`/notes/randomness` for more information.

Note:
``padding='valid'`` is the same as no padding. ``padding='same'`` pads
the input so the output has the shape as the input. However, this mode
doesn't support any stride values other than 1.

Args:
in_channels (int): Number of channels in the input image
out_channels (int): Number of channels produced by the convolution
kernel_size (int or tuple): Size of the convolving kernel
stride (int or tuple, optional): Stride of the convolution. Default: 1
padding (int, tuple or str, optional): Padding added to all four sides of
the input. Default: 0
padding_mode (string, optional): ``'zeros'``, ``'reflect'``,
``'replicate'`` or ``'circular'``. Default: ``'zeros'``
dilation (int or tuple, optional): Spacing between kernel elements. Default: 1
groups (int, optional): Number of blocked connections from input
channels to output channels. Default: 1
bias (bool, optional): If ``True``, adds a learnable bias to the
output. Default: ``True``

Shape:
- Input: :math:`(N, C_{in}, H_{in}, W_{in})`
- Output: :math:`(N, C_{out}, H_{out}, W_{out})` where

.. math::
H_{out} = \left\lfloor\frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0]
\times (\text{kernel_size}[0] - 1) - 1}{\text{stride}[0]} + 1\right\rfloor

.. math::

W_{out} = \left\lfloor\frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1]
\times (\text{kernel_size}[1] - 1) - 1}{\text{stride}[1]} + 1\right\rfloor

Attributes:
weight (Tensor): the learnable weights of the module of shape
:math:`(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}},`
:math:`\text{kernel\_size[0]}, \text{kernel\_size[1]})`.
The values of these weights are sampled from
:math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where

:math:`k = \frac{groups}{C_\text{in} * \prod_{i=0}^{1}\text{kernel\_size}[i]}`
bias (Tensor):   the learnable bias of the module of shape
(out_channels). If :attr:`bias` is ``True``,
then the values of these weights are
sampled from :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
:math:`k = \frac{groups}{C_\text{in} * \prod_{i=0}^{1}\text{kernel\_size}[i]}`

Examples:

```
>>> # With square kernels and equal stride
>>> m = nn.Conv2d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
>>> # non-square kernels and unequal stride and with padding and dilation
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2), dilation=(3, 1))
>>> input = torch.randn(20, 16, 50, 100)
>>> output = m(input)
```

.. _cross-correlation:
https://en.wikipedia.org/wiki/Cross-correlation

.. _link:
https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md

---

BatchNorm2d

nn.BatchNorm2d(
num_features,
eps=1e-05,
momentum=0.1,
affine=True,
track_running_stats=True,
device=None,

```
    dtype=None,
)
Docstring:
Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs
with additional channel dimension) as described in the paper
`Batch Normalization: Accelerating Deep Network Training by Reducing
Internal Covariate Shift <https://arxiv.org/abs/1502.03167>`__ .

.. math::

    y = \frac{x - \mathrm{E}[x]}{ \sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta

The mean and standard-deviation are calculated per-dimension over
the mini-batches and :math:`\gamma` and :math:`\beta` are learnable parameter vectors
of size `C` (where `C` is the input size). By default, the elements of :math:`\gamma` are set
to 1 and the elements of :math:`\beta` are set to 0. The standard-deviation is calculated
via the biased estimator, equivalent to `torch.var(input, unbiased=False)`.

Also by default, during training this layer keeps running estimates of its
computed mean and variance, which are then used for normalization during
evaluation. The running estimates are kept with a default :attr:`momentum`
of 0.1.

If :attr:`track_running_stats` is set to ``False``, this layer then does not
keep running estimates, and batch statistics are instead used during
evaluation time as well.

.. note::
    This :attr:`momentum` argument is different from one used in optimizer
    classes and the conventional notion of momentum. Mathematically, the
    update rule for running statistics here is
    :math:`\hat{x}_\text{new} = (1 - \text{momentum}) \times \hat{x} + \text{momentum}
\times x_t`,
    where :math:`\hat{x}` is the estimated statistic and :math:`x_t` is the
    new observed value.

Because the Batch Normalization is done over the `C` dimension, computing statistics
on `(N, H, W)` slices, it's common terminology to call this Spatial Batch Normalization.

Args:
    num_features: :math:`C` from an expected input of size
        :math:`(N, C, H, W)`
    eps: a value added to the denominator for numerical stability.
        Default: 1e-5
    momentum: the value used for the running_mean and running_var
        computation. Can be set to ``None`` for cumulative moving average
```

(i.e. simple average). Default: 0.1
affine: a boolean value that when set to ``True``, this module has
learnable affine parameters. Default: ``True``
track_running_stats: a boolean value that when set to ``True``, this
module tracks the running mean and variance, and when set to ``False``,
this module does not track such statistics, and initializes statistics
buffers :attr:`running_mean` and :attr:`running_var` as ``None``.
When these buffers are ``None``, this module always uses batch statistics.
in both training and eval modes. Default: ``True``

Shape:
- Input: :math:`(N, C, H, W)`
- Output: :math:`(N, C, H, W)` (same shape as input)

Examples::

    >>> # With Learnable Parameters
    >>> m = nn.BatchNorm2d(100)
    >>> # Without Learnable Parameters
    >>> m = nn.BatchNorm2d(100, affine=False)
    >>> input = torch.randn(20, 100, 35, 45)
    >>> output = m(input)

Init docstring: Initializes internal Module state, shared by both nn.Module and ScriptModule.
File: c:\users\mzand\appdata\roaming\python\python39\site-packages\torch\nn\modules\batchnorm.py
Type: type
Subclasses: BatchNorm2d, BatchNorm2d

---

nn.Linear

nn.Linear(
in_features: int,
out_features: int,
bias: bool = True,
device=None,
dtype=None)

Docstring:

Applies a linear transformation to the incoming data: :math:`y = xA^T + b`

This module supports :ref:`TensorFloat32<tf32_on_ampere>`.

Args:
in_features: size of each input sample

```
    out_features: size of each output sample
    bias: If set to ``False``, the layer will not learn an additive bias.
        Default: ``True``

Shape:
    - Input: :math:`(*, H_{in})` where :math:`*` means any number of
      dimensions including none and :math:`H_{in} = \text{in\_features}`.
    - Output: :math:`(*, H_{out})` where all but the last dimension
      are the same shape as the input and :math:`H_{out} = \text{out\_features}`.

Attributes:
    weight: the learnable weights of the module of shape
        :math:`(\text{out\_features}, \text{in\_features})`. The values are
        initialized from :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})`, where
        :math:`k = \frac{1}{\text{in\_features}}`
    bias:   the learnable bias of the module of shape :math:`(\text{out\_features})`.
            If :attr:`bias` is ``True``, the values are initialized from
            :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
            :math:`k = \frac{1}{\text{in\_features}}`

Examples::

    >>> m = nn.Linear(20, 30)
    >>> input = torch.randn(128, 20)
    >>> output = m(input)
    >>> print(output.size())
    torch.Size([128, 30])
Init docstring: Initializes internal Module state, shared by both nn.Module and ScriptModule.
File:           c:\users\mzand\appdata\roaming\python\python39\site-packages\torch\nn\modules\linear.py
Type:           type
Subclasses:     NonDynamicallyQuantizableLinear, LazyLinear, Linear, Linear
```
