Contents

Signature:torch.nn.functional.max_pool2d()

torch.nn.functional.max_pool2d(input, kernel_size, stride=None, padding=0, dilation=1, ceil_mode=False, return_indices=False)

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

Note:The order of ceil_mode and return_indices is different from what seen in maxpool2d , and will change in a future release.

Parameters:

> **input** : input tensor (\text{minibatch} , \text{in_channels} , iH , iW)(minibatch,in_channels,iH,iW), minibatch dim optional.
>
> **kernel_size** – size of the pooling region. Can be a single number or a tuple (kH, kW)
>
> **stride** : stride of the pooling operation. Can be a single number or a tuple (sH, sW). Default: kernel_size
>
> **padding** : Implicit negative infinity padding to be added on both sides, must be >= 0 and <= kernel_size / 2.
>
> **dilation** : The stride between elements within a sliding window, must be > 0.
>
> **ceil_mode** : If True, will use ceil instead of floor to compute the output shape. This ensures that every element in the input tensor is covered by a sliding window.
>
> **return_indices** : If True, will return the argmax along with the max values. Useful for torch.nn.functional.max_unpool2d later

[*Refrence: https://pytorch.org/docs/stable/generated/torch.nn.functional.max_pool2d.html*]

Signature: torch.nn.functional.leaky_relu()

leaky_relu() : Leaky ReLU function is **an improved version of the ReLU activation function**. As for the ReLU activation function, the gradient is 0 for all the values of inputs that are less than zero, which would deactivate the neurons in that region and may cause dying ReLU problem.

F.leaky_relu(input: torch.Tensor, negative_slope: float = 0.01, inplace: bool = False)

What is the difference between ReLU and leaky ReLU?

Parametric ReLU has the same advantage with the only difference that the slope of the output for negative inputs is a learnable parameter while in the Leaky ReLU it's a hyperparameter**.**

[*Refrence : https://ai.stackexchange.com/questions/7274/what-are-the-advantages-of-relu-vs-leaky-relu-and-parametric-relu-if-any*]

Signature:torch.nn.functional.dropout()

torch.nn.functional.dropout(input: torch.Tensor, p: float = 0.5, training: bool = True,
    inplace: bool = False)
-> torch.Tensor

Docstring:

During training, randomly zeroes some of the elements of the input
tensor with probability :attr:`p` using samples from a Bernoulli
distribution.

See :class:`~torch.nn.Dropout` for details.

Args:
    p: probability of an element to be zeroed. Default: 0.5
    training: apply dropout if is ``True``. Default: ``True``
    inplace: If set to ``True``, will do this operation in-place. Default: ``False``

Signature: torch.nn.Module.train(self: ~T, mode: bool = True) -> ~T

Docstring:

Sets the module in training mode.

This has any effect only on certain modules. See documentations of
particular modules for details of their behaviors in training/evaluation
mode, if they are affected, e.g. :class:`Dropout`, :class:`BatchNorm`,
etc.

Args:
    mode (bool): whether to set training mode (``True``) or evaluation
                mode (``False``). Default: ``True``.

Returns:
    Module: self
Type: function

More explanation:

model.train() tells your model that you are training the model. So effectively layers like dropout, batchnorm etc. which behave different on the train and test procedures know what is going on and hence can behave accordingly.

More details: It sets the mode to train (see source code). You can call either model.eval() or model.train(mode=False) to tell that you are testing. It is somewhat intuitive to expect train function to train model but it does not do that. It just sets the mode.

[*Refrence: https://stackoverflow.com/questions/51433378/what-does-model-train-do-in-pytorch*]

Signature: torch.nn.Module.eval()

torch.nn.Module.eval(self: ~T) -> ~T

Docstring:

Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of
particular modules for details of their behaviors in training/evaluation
mode, if they are affected, e.g. :class:`Dropout`, :class:`BatchNorm`,
etc.

This is equivalent with :meth:`self.train(False) <torch.nn.Module.train>`.

See :ref:`locally-disable-grad-doc` for a comparison between
`.eval()` and several similar mechanisms that may be confused with it.
Returns:
Module: self
Type: function

---

signature: torch.optim.Adam()

torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)

Docstring:

Implements Adam algorithm.

Args:

params (iterable): iterable of parameters to optimize or dicts defining
parameter groups

lr (float, optional): learning rate (default: 1e-3)

betas (Tuple[float, float], optional): coefficients used for computing
running averages of gradient and its square (default: (0.9, 0.999))

eps (float, optional): term added to the denominator to improve
numerical stability (default: 1e-8)

weight_decay (float, optional): weight decay (L2 penalty) (default: 0)

amsgrad (boolean, optional): whether to use the AMSGrad variant of this
algorithm from the paper `On the Convergence of Adam and Beyond`_
(default: False)

.. _Adam\: A Method for Stochastic Optimization:
https://arxiv.org/abs/1412.6980
.. _On the Convergence of Adam and Beyond:
https://openreview.net/forum?id=ryQu7f-RZ
packages\torch\optim\adam.py

---

signature： nn.CrossEntropyLoss()

nn.CrossEntropyLoss(weight: Optional[torch.Tensor] = None, size_average=None, ignore_index: int = -100, reduce=None, reduction: str = 'mean', label_smoothing: float = 0.0)

Docstring：
This criterion computes the cross entropy loss between input and target.

It is useful when training a classification problem with `C` classes.
If provided, the optional argument :attr:`weight` should be a 1D `Tensor`
assigning weight to each of the classes.
This is particularly useful when you have an unbalanced training set.

The `input` is expected to contain raw, unnormalized scores for each class.
`input` has to be a Tensor of size either :math:`(minibatch, C)` or
:math:`(minibatch, C, d_1, d_2, ..., d_K)` with :math:`K \geq 1` for the
`K`-dimensional case. The latter is useful for higher dimension inputs, such
as computing cross entropy loss per-pixel for 2D images.

The `target` that this criterion expects should contain either:

- Class indices in the range :math:`[0, C-1]` where :math:`C` is the number of classes; if`ignore_index` is specified, this loss also accepts this class index (this index may not necessarily be in the class range). The unreduced (i.e. with :attr:`reduction` set to ``'none'``) loss for this case can be described as:

Note that this case is equivalent to the combination of class:`~torch.nn.LogSoftmax` and class:`~torch.nn.NLLLoss`.

- Probabilities for each class; useful when labels beyond a single class per minibatch item are required, such as for blended labels, label smoothing, etc. The unreduced (i.e. with :attr:`reduction` set to ``'none'``) loss for this case can be described as:

note:
The performance of this criterion is generally better when `target` contains class indices, as this allows for optimized computation. Consider providing `target` as class probabilities only when a single class label per minibatch item is too restrictive.

```
Args:
    weight (Tensor, optional): a manual rescaling weight given to each class.
        If given, has to be a Tensor of size `C`
    size_average (bool, optional): Deprecated (see :attr:`reduction`). By default,
        the losses are averaged over each loss element in the batch. Note that for
        some losses, there are multiple elements per sample. If the field :attr:`size_average`
        is set to ``False``, the losses are instead summed for each minibatch. Ignored
        when :attr:`reduce` is ``False``. Default: ``True``
    ignore_index (int, optional): Specifies a target value that is ignored
        and does not contribute to the input gradient. When :attr:`size_average` is
        ``True``, the loss is averaged over non-ignored targets. Note that
        :attr:`ignore_index` is only applicable when the target contains class indices.
```

```
reduce (bool, optional): Deprecated (see :attr:`reduction`). By default, the
    losses are averaged or summed over observations for each minibatch depending
    on :attr:`size_average`. When :attr:`reduce` is ``False``, returns a loss per
    batch element instead and ignores :attr:`size_average`. Default: ``True``
reduction (string, optional): Specifies the reduction to apply to the output:
    ``'none'`` | ``'mean'`` | ``'sum'``. ``'none'``: no reduction will
    be applied, ``'mean'``: the weighted mean of the output is taken,
    ``'sum'``: the output will be summed. Note: :attr:`size_average`
    and :attr:`reduce` are in the process of being deprecated, and in
    the meantime, specifying either of those two args will override
    :attr:`reduction`. Default: ``'mean'``
label_smoothing (float, optional): A float in [0.0, 1.0]. Specifies the amount
    of smoothing when computing the loss, where 0.0 means no smoothing. The targets
    become a mixture of the original ground truth and a uniform distribution as described in
    `Rethinking the Inception Architecture for Computer Vision <https://arxiv.org/abs/1512.00567>`__. Default: :math:`0.0`.
```