import warnings

from module import Module

import functional as F

import \_reduction as \_Reduction

class \_Loss(Module):

def \_\_init\_\_(self, size\_average=None, reduce=None, reduction='mean'):

super(\_Loss, self).\_\_init\_\_()

if size\_average is not None or reduce is not None:

self.reduction = \_Reduction.legacy\_get\_string(size\_average, reduce)

else:

self.reduction = reduction

class \_WeightedLoss(\_Loss):

def \_\_init\_\_(self, weight=None, size\_average=None, reduce=None, reduction='mean'):

super(\_WeightedLoss, self).\_\_init\_\_(size\_average, reduce, reduction)

self.register\_buffer('weight', weight)

class L1Loss(\_Loss):

r"""Creates a criterion that measures the mean absolute error (MAE) between each element in

the input :math:`x` and target :math:`y`.

The unreduced (i.e. with :attr:`reduction` set to ``'none'``) loss can be described as:

.. math::

\ell(x, y) = L = \{l\_1,\dots,l\_N\}^\top, \quad

l\_n = \left| x\_n - y\_n \right|,

where :math:`N` is the batch size. If :attr:`reduction` is not ``'none'``

(default ``'mean'``), then:

.. math::

\ell(x, y) =

\begin{cases}

\operatorname{mean}(L), & \text{if reduction} = \text{'mean';}\\

\operatorname{sum}(L), & \text{if reduction} = \text{'sum'.}

\end{cases}

:math:`x` and :math:`y` are tensors of arbitrary shapes with a total

of :math:`n` elements each.

The sum operation still operates over all the elements, and divides by :math:`n`.

The division by :math:`n` can be avoided if one sets ``reduction = 'sum'``.

Args:

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 reduce is ``False``. Default: ``True``

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

Shape:

- Input: :math:`(N, \*)` where :math:`\*` means, any number of additional

dimensions

- Target: :math:`(N, \*)`, same shape as the input

- Output: scalar. If :attr:`reduction` is ``'none'``, then

:math:`(N, \*)`, same shape as the input

Examples::

>>> loss = nn.L1Loss()

>>> input = torch.randn(3, 5, requires\_grad=True)

>>> target = torch.randn(3, 5)

>>> output = loss(input, target)

>>> output.backward()

"""

\_\_constants\_\_ = ['reduction']

def \_\_init\_\_(self, size\_average=None, reduce=None, reduction='mean'):

super(L1Loss, self).\_\_init\_\_(size\_average, reduce, reduction)

def forward(self, input, target):

return F.l1\_loss(input, target, reduction=self.reduction)

class NLLLoss(\_WeightedLoss):

r"""The negative log likelihood loss. It is useful to train 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` given through a forward call is expected to contain

log-probabilities of 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 (described later).

Obtaining log-probabilities in a neural network is easily achieved by

adding a `LogSoftmax` layer in the last layer of your network.

You may use `CrossEntropyLoss` instead, if you prefer not to add an extra

layer.

The `target` that this loss expects should be a class index in the range :math:`[0, C-1]`

where `C = 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 can be described as:

.. math::

\ell(x, y) = L = \{l\_1,\dots,l\_N\}^\top, \quad

l\_n = - w\_{y\_n} x\_{n,y\_n}, \quad

w\_{c} = \text{weight}[c] \cdot \mathbb{1}\{c \not= \text{ignore\\_index}\},

where :math:`x` is the input, :math:`y` is the target, :math:`w` is the weight, and

:math:`N` is the batch size. If :attr:`reduction` is not ``'none'``

(default ``'mean'``), then

.. math::

\ell(x, y) = \begin{cases}

\sum\_{n=1}^N \frac{1}{\sum\_{n=1}^N w\_{y\_n}} l\_n, &

\text{if reduction} = \text{'mean';}\\

\sum\_{n=1}^N l\_n, &

\text{if reduction} = \text{'sum'.}

\end{cases}

Can also be used for higher dimension inputs, such as 2D images, by providing

an input of size :math:`(minibatch, C, d\_1, d\_2, ..., d\_K)` with :math:`K \geq 1`,

where :math:`K` is the number of dimensions, and a target of appropriate shape

(see below). In the case of images, it computes NLL loss per-pixel.

Args:

weight (Tensor, optional): a manual rescaling weight given to each

class. If given, it has to be a Tensor of size `C`. Otherwise, it is

treated as if having all ones.

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 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.

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

Shape:

- Input: :math:`(N, C)` where `C = number of classes`, or

:math:`(N, C, d\_1, d\_2, ..., d\_K)` with :math:`K \geq 1`

in the case of `K`-dimensional loss.

- Target: :math:`(N)` where each value is :math:`0 \leq \text{targets}[i] \leq C-1`, or

:math:`(N, d\_1, d\_2, ..., d\_K)` with :math:`K \geq 1` in the case of

K-dimensional loss.

- Output: scalar.

If :attr:`reduction` is ``'none'``, then the same size as the target: :math:`(N)`, or

:math:`(N, d\_1, d\_2, ..., d\_K)` with :math:`K \geq 1` in the case

of K-dimensional loss.

Examples::

>>> m = nn.LogSoftmax(dim=1)

>>> loss = nn.NLLLoss()

>>> # input is of size N x C = 3 x 5

>>> input = torch.randn(3, 5, requires\_grad=True)

>>> # each element in target has to have 0 <= value < C

>>> target = torch.tensor([1, 0, 4])

>>> output = loss(m(input), target)

>>> output.backward()

>>>

>>>

>>> # 2D loss example (used, for example, with image inputs)

>>> N, C = 5, 4

>>> loss = nn.NLLLoss()

>>> # input is of size N x C x height x width

>>> data = torch.randn(N, 16, 10, 10)

>>> conv = nn.Conv2d(16, C, (3, 3))

>>> m = nn.LogSoftmax(dim=1)

>>> # each element in target has to have 0 <= value < C

>>> target = torch.empty(N, 8, 8, dtype=torch.long).random\_(0, C)

>>> output = loss(m(conv(data)), target)

>>> output.backward()

"""

\_\_constants\_\_ = ['ignore\_index', 'reduction']

def \_\_init\_\_(self, weight=None, size\_average=None, ignore\_index=-100,

reduce=None, reduction='mean'):

super(NLLLoss, self).\_\_init\_\_(weight, size\_average, reduce, reduction)

self.ignore\_index = ignore\_index

def forward(self, input, target):

return F.nll\_loss(input, target, weight=self.weight, ignore\_index=self.ignore\_index, reduction=self.reduction)

class NLLLoss2d(NLLLoss):

def \_\_init\_\_(self, weight=None, size\_average=None, ignore\_index=-100,

reduce=None, reduction='mean'):

warnings.warn("NLLLoss2d has been deprecated. "

"Please use NLLLoss instead as a drop-in replacement and see "

"<https://pytorch.org/docs/master/nn.html#torch.nn.NLLLoss for more details.>")

super(NLLLoss2d, self).\_\_init\_\_(weight, size\_average, ignore\_index, reduce, reduction)

class PoissonNLLLoss(\_Loss):

r"""Negative log likelihood loss with Poisson distribution of target.

The loss can be described as:

.. math::

\text{target} \sim \mathrm{Poisson}(\text{input})

\text{loss}(\text{input}, \text{target}) = \text{input} - \text{target} \* \log(\text{input})

+ \log(\text{target!})

The last term can be omitted or approximated with Stirling formula. The

approximation is used for target values more than 1. For targets less or

equal to 1 zeros are added to the loss.

Args:

log\_input (bool, optional): if ``True`` the loss is computed as

:math:`\exp(\text{input}) - \text{target}\*\text{input}`, if ``False`` the loss is

:math:`\text{input} - \text{target}\*\log(\text{input}+\text{eps})`.

full (bool, optional): whether to compute full loss, i. e. to add the

Stirling approximation term

.. math::

\text{target}\*\log(\text{target}) - \text{target} + 0.5 \* \log(2\pi\text{target}).

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 reduce is ``False``. Default: ``True``

eps (float, optional): Small value to avoid evaluation of :math:`\log(0)` when

:attr:`log\_input = False`. Default: 1e-8

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

Examples::

>>> loss = nn.PoissonNLLLoss()

>>> log\_input = torch.randn(5, 2, requires\_grad=True)

>>> target = torch.randn(5, 2)

>>> output = loss(log\_input, target)

>>> output.backward()

Shape:

- Input: :math:`(N, \*)` where :math:`\*` means, any number of additional

dimensions

- Target: :math:`(N, \*)`, same shape as the input

- Output: scalar by default. If :attr:`reduction` is ``'none'``, then :math:`(N, \*)`,

the same shape as the input

"""

\_\_constants\_\_ = ['log\_input', 'full', 'eps', 'reduction']

def \_\_init\_\_(self, log\_input=True, full=False, size\_average=None,

eps=1e-8, reduce=None, reduction='mean'):

super(PoissonNLLLoss, self).\_\_init\_\_(size\_average, reduce, reduction)

self.log\_input = log\_input

self.full = full

self.eps = eps

def forward(self, log\_input, target):

return F.poisson\_nll\_loss(log\_input, target, log\_input=self.log\_input, full=self.full,

eps=self.eps, reduction=self.reduction)

class KLDivLoss(\_Loss):

r"""The `Kullback-Leibler divergence`\_ Loss

KL divergence is a useful distance measure for continuous distributions

and is often useful when performing direct regression over the space of

(discretely sampled) continuous output distributions.

As with :class:`~torch.nn.NLLLoss`, the `input` given is expected to contain

\*log-probabilities\* and is not restricted to a 2D Tensor.

The targets are interpreted as \*probabilities\* by default, but could be considered

as \*log-probabilities\* with :attr:`log\_target` set to ``True``.

This criterion expects a `target` `Tensor` of the same size as the

`input` `Tensor`.

The unreduced (i.e. with :attr:`reduction` set to ``'none'``) loss can be described as:

.. math::

l(x,y) = L = \{ l\_1,\dots,l\_N \}, \quad

l\_n = y\_n \cdot \left( \log y\_n - x\_n \right)

where the index :math:`N` spans all dimensions of ``input`` and :math:`L` has the same

shape as ``input``. If :attr:`reduction` is not ``'none'`` (default ``'mean'``), then:

.. math::

\ell(x, y) = \begin{cases}

\operatorname{mean}(L), & \text{if reduction} = \text{'mean';} \\

\operatorname{sum}(L), & \text{if reduction} = \text{'sum'.}

\end{cases}

In default :attr:`reduction` mode ``'mean'``, the losses are averaged for each minibatch over observations

\*\*as well as\*\* over dimensions. ``'batchmean'`` mode gives the correct KL divergence where losses

are averaged over batch dimension only. ``'mean'`` mode's behavior will be changed to the same as

``'batchmean'`` in the next major release.

.. \_Kullback-Leibler divergence:

<https://en.wikipedia.org/wiki/Kullback-Leibler_divergence>

Args:

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 reduce is ``False``. Default: ``True``

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'`` | ``'batchmean'`` | ``'sum'`` | ``'mean'``.

``'none'``: no reduction will be applied.

``'batchmean'``: the sum of the output will be divided by batchsize.

``'sum'``: the output will be summed.

``'mean'``: the output will be divided by the number of elements in the output.

Default: ``'mean'``

log\_target (bool, optional): Specifies whether `target` is passed in the log space.

Default: ``False``

.. 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`.

.. note::

:attr:`reduction` = ``'mean'`` doesn't return the true kl divergence value, please use

:attr:`reduction` = ``'batchmean'`` which aligns with KL math definition.

In the next major release, ``'mean'`` will be changed to be the same as ``'batchmean'``.

Shape:

- Input: :math:`(N, \*)` where :math:`\*` means, any number of additional

dimensions

- Target: :math:`(N, \*)`, same shape as the input

- Output: scalar by default. If :attr:``reduction`` is ``'none'``, then :math:`(N, \*)`,

the same shape as the input

"""

\_\_constants\_\_ = ['reduction']

def \_\_init\_\_(self, size\_average=None, reduce=None, reduction='mean', log\_target=False):

super(KLDivLoss, self).\_\_init\_\_(size\_average, reduce, reduction)

self.log\_target = log\_target

def forward(self, input, target):

return F.kl\_div(input, target, reduction=self.reduction, log\_target=self.log\_target)

class MSELoss(\_Loss):

r"""Creates a criterion that measures the mean squared error (squared L2 norm) between

each element in the input :math:`x` and target :math:`y`.

The unreduced (i.e. with :attr:`reduction` set to ``'none'``) loss can be described as:

.. math::

\ell(x, y) = L = \{l\_1,\dots,l\_N\}^\top, \quad

l\_n = \left( x\_n - y\_n \right)^2,

where :math:`N` is the batch size. If :attr:`reduction` is not ``'none'``

(default ``'mean'``), then:

.. math::

\ell(x, y) =

\begin{cases}

\operatorname{mean}(L), & \text{if reduction} = \text{'mean';}\\

\operatorname{sum}(L), & \text{if reduction} = \text{'sum'.}

\end{cases}

:math:`x` and :math:`y` are tensors of arbitrary shapes with a total

of :math:`n` elements each.

The mean operation still operates over all the elements, and divides by :math:`n`.

The division by :math:`n` can be avoided if one sets ``reduction = 'sum'``.

Args:

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 reduce is ``False``. Default: ``True``

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

Shape:

- Input: :math:`(N, \*)` where :math:`\*` means, any number of additional

dimensions

- Target: :math:`(N, \*)`, same shape as the input

Examples::

>>> loss = nn.MSELoss()

>>> input = torch.randn(3, 5, requires\_grad=True)

>>> target = torch.randn(3, 5)

>>> output = loss(input, target)

>>> output.backward()

"""

\_\_constants\_\_ = ['reduction']

def \_\_init\_\_(self, size\_average=None, reduce=None, reduction='mean'):

super(MSELoss, self).\_\_init\_\_(size\_average, reduce, reduction)

def forward(self, input, target):

return F.mse\_loss(input, target, reduction=self.reduction)

class BCELoss(\_WeightedLoss):

r"""Creates a criterion that measures the Binary Cross Entropy

between the target and the output:

The unreduced (i.e. with :attr:`reduction` set to ``'none'``) loss can be described as:

.. math::

\ell(x, y) = L = \{l\_1,\dots,l\_N\}^\top, \quad

l\_n = - w\_n \left[ y\_n \cdot \log x\_n + (1 - y\_n) \cdot \log (1 - x\_n) \right],

where :math:`N` is the batch size. If :attr:`reduction` is not ``'none'``

(default ``'mean'``), then

.. math::

\ell(x, y) = \begin{cases}

\operatorname{mean}(L), & \text{if reduction} = \text{'mean';}\\

\operatorname{sum}(L), & \text{if reduction} = \text{'sum'.}

\end{cases}

This is used for measuring the error of a reconstruction in for example

an auto-encoder. Note that the targets :math:`y` should be numbers

between 0 and 1.

Notice that if :math:`x\_n` is either 0 or 1, one of the log terms would be

mathematically undefined in the above loss equation. PyTorch chooses to set

:math:`\log (0) = -\infty`, since :math:`\lim\_{x\to 0} \log (x) = -\infty`.

However, an infinite term in the loss equation is not desirable for several reasons.

For one, if either :math:`y\_n = 0` or :math:`(1 - y\_n) = 0`, then we would be

multipying 0 with infinity. Secondly, if we have an infinite loss value, then

we would also have an infinite term in our gradient, since

:math:`\lim\_{x\to 0} \frac{d}{dx} \log (x) = \infty`.

This would make BCELoss's backward method nonlinear with respect to :math:`x\_n`,

and using it for things like linear regression would not be straight-forward.

Our solution is that BCELoss clamps its log function outputs to be greater than

or equal to -100. This way, we can always have a finite loss value and a linear

backward method.

Args:

weight (Tensor, optional): a manual rescaling weight given to the loss

of each batch element. If given, has to be a Tensor of size `nbatch`.

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 reduce is ``False``. Default: ``True``

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

Shape:

- Input: :math:`(N, \*)` where :math:`\*` means, any number of additional

dimensions

- Target: :math:`(N, \*)`, same shape as the input

- Output: scalar. If :attr:`reduction` is ``'none'``, then :math:`(N, \*)`, same

shape as input.

Examples::

>>> m = nn.Sigmoid()

>>> loss = nn.BCELoss()

>>> input = torch.randn(3, requires\_grad=True)

>>> target = torch.empty(3).random\_(2)

>>> output = loss(m(input), target)

>>> output.backward()

"""

\_\_constants\_\_ = ['reduction']

def \_\_init\_\_(self, weight=None, size\_average=None, reduce=None, reduction='mean'):

super(BCELoss, self).\_\_init\_\_(weight, size\_average, reduce, reduction)

def forward(self, input, target):

return F.binary\_cross\_entropy(input, target, weight=self.weight, reduction=self.reduction)

class BCEWithLogitsLoss(\_Loss):

r"""This loss combines a `Sigmoid` layer and the `BCELoss` in one single

class. This version is more numerically stable than using a plain `Sigmoid`

followed by a `BCELoss` as, by combining the operations into one layer,

we take advantage of the log-sum-exp trick for numerical stability.

The unreduced (i.e. with :attr:`reduction` set to ``'none'``) loss can be described as:

.. math::

\ell(x, y) = L = \{l\_1,\dots,l\_N\}^\top, \quad

l\_n = - w\_n \left[ y\_n \cdot \log \sigma(x\_n)

+ (1 - y\_n) \cdot \log (1 - \sigma(x\_n)) \right],

where :math:`N` is the batch size. If :attr:`reduction` is not ``'none'``

(default ``'mean'``), then

.. math::

\ell(x, y) = \begin{cases}

\operatorname{mean}(L), & \text{if reduction} = \text{'mean';}\\

\operatorname{sum}(L), & \text{if reduction} = \text{'sum'.}

\end{cases}

This is used for measuring the error of a reconstruction in for example

an auto-encoder. Note that the targets `t[i]` should be numbers

between 0 and 1.

It's possible to trade off recall and precision by adding weights to positive examples.

In the case of multi-label classification the loss can be described as:

.. math::

\ell\_c(x, y) = L\_c = \{l\_{1,c},\dots,l\_{N,c}\}^\top, \quad

l\_{n,c} = - w\_{n,c} \left[ p\_c y\_{n,c} \cdot \log \sigma(x\_{n,c})

+ (1 - y\_{n,c}) \cdot \log (1 - \sigma(x\_{n,c})) \right],

where :math:`c` is the class number (:math:`c > 1` for multi-label binary classification,

:math:`c = 1` for single-label binary classification),

:math:`n` is the number of the sample in the batch and

:math:`p\_c` is the weight of the positive answer for the class :math:`c`.

:math:`p\_c > 1` increases the recall, :math:`p\_c < 1` increases the precision.

For example, if a dataset contains 100 positive and 300 negative examples of a single class,

then `pos\_weight` for the class should be equal to :math:`\frac{300}{100}=3`.

The loss would act as if the dataset contains :math:`3\times 100=300` positive examples.

Examples::

>>> target = torch.ones([10, 64], dtype=torch.float32) # 64 classes, batch size = 10

>>> output = torch.full([10, 64], 1.5) # A prediction (logit)

>>> pos\_weight = torch.ones([64]) # All weights are equal to 1

>>> criterion = torch.nn.BCEWithLogitsLoss(pos\_weight=pos\_weight)

>>> criterion(output, target) # -log(sigmoid(1.5))

tensor(0.2014)

Args:

weight (Tensor, optional): a manual rescaling weight given to the loss

of each batch element. If given, has to be a Tensor of size `nbatch`.

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 reduce is ``False``. Default: ``True``

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

pos\_weight (Tensor, optional): a weight of positive examples.

Must be a vector with length equal to the number of classes.

Shape:

- Input: :math:`(N, \*)` where :math:`\*` means, any number of additional dimensions

- Target: :math:`(N, \*)`, same shape as the input

- Output: scalar. If :attr:`reduction` is ``'none'``, then :math:`(N, \*)`, same

shape as input.

Examples::

>>> loss = nn.BCEWithLogitsLoss()

>>> input = torch.randn(3, requires\_grad=True)

>>> target = torch.empty(3).random\_(2)

>>> output = loss(input, target)

>>> output.backward()

"""

def \_\_init\_\_(self, weight=None, size\_average=None, reduce=None, reduction='mean', pos\_weight=None):

super(BCEWithLogitsLoss, self).\_\_init\_\_(size\_average, reduce, reduction)

self.register\_buffer('weight', weight)

self.register\_buffer('pos\_weight', pos\_weight)

def forward(self, input, target):

return F.binary\_cross\_entropy\_with\_logits(input, target,

self.weight,

pos\_weight=self.pos\_weight,

reduction=self.reduction)

class HingeEmbeddingLoss(\_Loss):

r"""Measures the loss given an input tensor :math:`x` and a labels tensor :math:`y`

(containing 1 or -1).

This is usually used for measuring whether two inputs are similar or

dissimilar, e.g. using the L1 pairwise distance as :math:`x`, and is typically

used for learning nonlinear embeddings or semi-supervised learning.

The loss function for :math:`n`-th sample in the mini-batch is

.. math::

l\_n = \begin{cases}

x\_n, & \text{if}\; y\_n = 1,\\

\max \{0, \Delta - x\_n\}, & \text{if}\; y\_n = -1,

\end{cases}

and the total loss functions is

.. math::

\ell(x, y) = \begin{cases}

\operatorname{mean}(L), & \text{if reduction} = \text{'mean';}\\

\operatorname{sum}(L), & \text{if reduction} = \text{'sum'.}

\end{cases}

where :math:`L = \{l\_1,\dots,l\_N\}^\top`.

Args:

margin (float, optional): Has a default value of `1`.

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 reduce is ``False``. Default: ``True``

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

Shape:

- Input: :math:`(\*)` where :math:`\*` means, any number of dimensions. The sum operation

operates over all the elements.

- Target: :math:`(\*)`, same shape as the input

- Output: scalar. If :attr:`reduction` is ``'none'``, then same shape as the input

"""

\_\_constants\_\_ = ['margin', 'reduction']

def \_\_init\_\_(self, margin=1.0, size\_average=None, reduce=None, reduction='mean'):

super(HingeEmbeddingLoss, self).\_\_init\_\_(size\_average, reduce, reduction)

self.margin = margin

def forward(self, input, target):

return F.hinge\_embedding\_loss(input, target, margin=self.margin, reduction=self.reduction)

class MultiLabelMarginLoss(\_Loss):

r"""Creates a criterion that optimizes a multi-class multi-classification

hinge loss (margin-based loss) between input :math:`x` (a 2D mini-batch `Tensor`)

and output :math:`y` (which is a 2D `Tensor` of target class indices).

For each sample in the mini-batch:

.. math::

\text{loss}(x, y) = \sum\_{ij}\frac{\max(0, 1 - (x[y[j]] - x[i]))}{\text{x.size}(0)}

where :math:`x \in \left\{0, \; \cdots , \; \text{x.size}(0) - 1\right\}`, \

:math:`y \in \left\{0, \; \cdots , \; \text{y.size}(0) - 1\right\}`, \

:math:`0 \leq y[j] \leq \text{x.size}(0)-1`, \

and :math:`i \neq y[j]` for all :math:`i` and :math:`j`.

:math:`y` and :math:`x` must have the same size.

The criterion only considers a contiguous block of non-negative targets that

starts at the front.

This allows for different samples to have variable amounts of target classes.

Args:

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 reduce is ``False``. Default: ``True``

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

Shape:

- Input: :math:`(C)` or :math:`(N, C)` where `N` is the batch size and `C`

is the number of classes.

- Target: :math:`(C)` or :math:`(N, C)`, label targets padded by -1 ensuring same shape as the input.

- Output: scalar. If :attr:`reduction` is ``'none'``, then :math:`(N)`.

Examples::

>>> loss = nn.MultiLabelMarginLoss()

>>> x = torch.FloatTensor([[0.1, 0.2, 0.4, 0.8]])

>>> # for target y, only consider labels 3 and 0, not after label -1

>>> y = torch.LongTensor([[3, 0, -1, 1]])

>>> loss(x, y)

>>> # 0.25 \* ((1-(0.1-0.2)) + (1-(0.1-0.4)) + (1-(0.8-0.2)) + (1-(0.8-0.4)))

tensor(0.8500)

"""

\_\_constants\_\_ = ['reduction']

def \_\_init\_\_(self, size\_average=None, reduce=None, reduction='mean'):

super(MultiLabelMarginLoss, self).\_\_init\_\_(size\_average, reduce, reduction)

def forward(self, input, target):

return F.multilabel\_margin\_loss(input, target, reduction=self.reduction)

class SmoothL1Loss(\_Loss):

r"""Creates a criterion that uses a squared term if the absolute

element-wise error falls below 1 and an L1 term otherwise.

It is less sensitive to outliers than the `MSELoss` and in some cases

prevents exploding gradients (e.g. see `Fast R-CNN` paper by Ross Girshick).

Also known as the Huber loss:

.. math::

\text{loss}(x, y) = \frac{1}{n} \sum\_{i} z\_{i}

where :math:`z\_{i}` is given by:

.. math::

z\_{i} =

\begin{cases}

0.5 (x\_i - y\_i)^2, & \text{if } |x\_i - y\_i| < 1 \\

|x\_i - y\_i| - 0.5, & \text{otherwise }

\end{cases}

:math:`x` and :math:`y` arbitrary shapes with a total of :math:`n` elements each

the sum operation still operates over all the elements, and divides by :math:`n`.

The division by :math:`n` can be avoided if sets ``reduction = 'sum'``.

Args:

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 reduce is ``False``. Default: ``True``

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

Shape:

- Input: :math:`(N, \*)` where :math:`\*` means, any number of additional

dimensions

- Target: :math:`(N, \*)`, same shape as the input

- Output: scalar. If :attr:`reduction` is ``'none'``, then

:math:`(N, \*)`, same shape as the input

"""

\_\_constants\_\_ = ['reduction']

def \_\_init\_\_(self, size\_average=None, reduce=None, reduction='mean'):

super(SmoothL1Loss, self).\_\_init\_\_(size\_average, reduce, reduction)

def forward(self, input, target):

return F.smooth\_l1\_loss(input, target, reduction=self.reduction)

class SoftMarginLoss(\_Loss):

r"""Creates a criterion that optimizes a two-class classification

logistic loss between input tensor :math:`x` and target tensor :math:`y`

(containing 1 or -1).

.. math::

\text{loss}(x, y) = \sum\_i \frac{\log(1 + \exp(-y[i]\*x[i]))}{\text{x.nelement}()}

Args:

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 reduce is ``False``. Default: ``True``

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

Shape:

- Input: :math:`(\*)` where :math:`\*` means, any number of additional

dimensions

- Target: :math:`(\*)`, same shape as the input

- Output: scalar. If :attr:`reduction` is ``'none'``, then same shape as the input

"""

\_\_constants\_\_ = ['reduction']

def \_\_init\_\_(self, size\_average=None, reduce=None, reduction='mean'):

super(SoftMarginLoss, self).\_\_init\_\_(size\_average, reduce, reduction)

def forward(self, input, target):

return F.soft\_margin\_loss(input, target, reduction=self.reduction)

class CrossEntropyLoss(\_WeightedLoss):

r"""This criterion combines :func:`nn.LogSoftmax` and :func:`nn.NLLLoss` in one single class.

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 (described later).

This criterion expects a class index in the range :math:`[0, C-1]` as the

`target` for each value of a 1D tensor of size `minibatch`; if `ignore\_index`

is specified, this criterion also accepts this class index (this index may not

necessarily be in the class range).

The loss can be described as:

.. math::

\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum\_j \exp(x[j])}\right)

= -x[class] + \log\left(\sum\_j \exp(x[j])\right)

or in the case of the :attr:`weight` argument being specified:

.. math::

\text{loss}(x, class) = weight[class] \left(-x[class] + \log\left(\sum\_j \exp(x[j])\right)\right)

The losses are averaged across observations for each minibatch.

Can also be used for higher dimension inputs, such as 2D images, by providing

an input of size :math:`(minibatch, C, d\_1, d\_2, ..., d\_K)` with :math:`K \geq 1`,

where :math:`K` is the number of dimensions, and a target of appropriate shape

(see below).

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 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.

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

Shape:

- Input: :math:`(N, C)` where `C = number of classes`, or

:math:`(N, C, d\_1, d\_2, ..., d\_K)` with :math:`K \geq 1`

in the case of `K`-dimensional loss.

- Target: :math:`(N)` where each value is :math:`0 \leq \text{targets}[i] \leq C-1`, or

:math:`(N, d\_1, d\_2, ..., d\_K)` with :math:`K \geq 1` in the case of

K-dimensional loss.

- Output: scalar.

If :attr:`reduction` is ``'none'``, then the same size as the target:

:math:`(N)`, or

:math:`(N, d\_1, d\_2, ..., d\_K)` with :math:`K \geq 1` in the case

of K-dimensional loss.

Examples::

>>> loss = nn.CrossEntropyLoss()

>>> input = torch.randn(3, 5, requires\_grad=True)

>>> target = torch.empty(3, dtype=torch.long).random\_(5)

>>> output = loss(input, target)

>>> output.backward()

"""

\_\_constants\_\_ = ['ignore\_index', 'reduction']

def \_\_init\_\_(self, weight=None, size\_average=None, ignore\_index=-100,

reduce=None, reduction='mean'):

super(CrossEntropyLoss, self).\_\_init\_\_(weight, size\_average, reduce, reduction)

self.ignore\_index = ignore\_index

def forward(self, input, target):

return F.cross\_entropy(input, target, weight=self.weight,

ignore\_index=self.ignore\_index, reduction=self.reduction)

class MultiLabelSoftMarginLoss(\_WeightedLoss):

r"""Creates a criterion that optimizes a multi-label one-versus-all

loss based on max-entropy, between input :math:`x` and target :math:`y` of size

:math:`(N, C)`.

For each sample in the minibatch:

.. math::

loss(x, y) = - \frac{1}{C} \* \sum\_i y[i] \* \log((1 + \exp(-x[i]))^{-1})

+ (1-y[i]) \* \log\left(\frac{\exp(-x[i])}{(1 + \exp(-x[i]))}\right)

where :math:`i \in \left\{0, \; \cdots , \; \text{x.nElement}() - 1\right\}`,

:math:`y[i] \in \left\{0, \; 1\right\}`.

Args:

weight (Tensor, optional): a manual rescaling weight given to each

class. If given, it has to be a Tensor of size `C`. Otherwise, it is

treated as if having all ones.

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 reduce is ``False``. Default: ``True``

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

Shape:

- Input: :math:`(N, C)` where `N` is the batch size and `C` is the number of classes.

- Target: :math:`(N, C)`, label targets padded by -1 ensuring same shape as the input.

- Output: scalar. If :attr:`reduction` is ``'none'``, then :math:`(N)`.

"""

\_\_constants\_\_ = ['reduction']

def \_\_init\_\_(self, weight=None, size\_average=None, reduce=None, reduction='mean'):

super(MultiLabelSoftMarginLoss, self).\_\_init\_\_(weight, size\_average, reduce, reduction)

def forward(self, input, target):

return F.multilabel\_soft\_margin\_loss(input, target, weight=self.weight, reduction=self.reduction)

class CosineEmbeddingLoss(\_Loss):

r"""Creates a criterion that measures the loss given input tensors

:math:`x\_1`, :math:`x\_2` and a `Tensor` label :math:`y` with values 1 or -1.

This is used for measuring whether two inputs are similar or dissimilar,

using the cosine distance, and is typically used for learning nonlinear

embeddings or semi-supervised learning.

The loss function for each sample is:

.. math::

\text{loss}(x, y) =

\begin{cases}

1 - \cos(x\_1, x\_2), & \text{if } y = 1 \\

\max(0, \cos(x\_1, x\_2) - \text{margin}), & \text{if } y = -1

\end{cases}

Args:

margin (float, optional): Should be a number from :math:`-1` to :math:`1`,

:math:`0` to :math:`0.5` is suggested. If :attr:`margin` is missing, the

default value is :math:`0`.

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 reduce is ``False``. Default: ``True``

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

"""

\_\_constants\_\_ = ['margin', 'reduction']

def \_\_init\_\_(self, margin=0., size\_average=None, reduce=None, reduction='mean'):

super(CosineEmbeddingLoss, self).\_\_init\_\_(size\_average, reduce, reduction)

self.margin = margin

def forward(self, input1, input2, target):

return F.cosine\_embedding\_loss(input1, input2, target, margin=self.margin, reduction=self.reduction)

class MarginRankingLoss(\_Loss):

r"""Creates a criterion that measures the loss given

inputs :math:`x1`, :math:`x2`, two 1D mini-batch `Tensors`,

and a label 1D mini-batch tensor :math:`y` (containing 1 or -1).

If :math:`y = 1` then it assumed the first input should be ranked higher

(have a larger value) than the second input, and vice-versa for :math:`y = -1`.

The loss function for each sample in the mini-batch is:

.. math::

\text{loss}(x, y) = \max(0, -y \* (x1 - x2) + \text{margin})

Args:

margin (float, optional): Has a default value of :math:`0`.

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 reduce is ``False``. Default: ``True``

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

Shape:

- Input: :math:`(N, D)` where `N` is the batch size and `D` is the size of a sample.

- Target: :math:`(N)`

- Output: scalar. If :attr:`reduction` is ``'none'``, then :math:`(N)`.

"""

\_\_constants\_\_ = ['margin', 'reduction']

def \_\_init\_\_(self, margin=0., size\_average=None, reduce=None, reduction='mean'):

super(MarginRankingLoss, self).\_\_init\_\_(size\_average, reduce, reduction)

self.margin = margin

def forward(self, input1, input2, target):

return F.margin\_ranking\_loss(input1, input2, target, margin=self.margin, reduction=self.reduction)

class MultiMarginLoss(\_WeightedLoss):

r"""Creates a criterion that optimizes a multi-class classification hinge

loss (margin-based loss) between input :math:`x` (a 2D mini-batch `Tensor`) and

output :math:`y` (which is a 1D tensor of target class indices,

:math:`0 \leq y \leq \text{x.size}(1)-1`):

For each mini-batch sample, the loss in terms of the 1D input :math:`x` and scalar

output :math:`y` is:

.. math::

\text{loss}(x, y) = \frac{\sum\_i \max(0, \text{margin} - x[y] + x[i]))^p}{\text{x.size}(0)}

where :math:`x \in \left\{0, \; \cdots , \; \text{x.size}(0) - 1\right\}`

and :math:`i \neq y`.

Optionally, you can give non-equal weighting on the classes by passing

a 1D :attr:`weight` tensor into the constructor.

The loss function then becomes:

.. math::

\text{loss}(x, y) = \frac{\sum\_i \max(0, w[y] \* (\text{margin} - x[y] + x[i]))^p)}{\text{x.size}(0)}

Args:

p (int, optional): Has a default value of :math:`1`. :math:`1` and :math:`2`

are the only supported values.

margin (float, optional): Has a default value of :math:`1`.

weight (Tensor, optional): a manual rescaling weight given to each

class. If given, it has to be a Tensor of size `C`. Otherwise, it is

treated as if having all ones.

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 reduce is ``False``. Default: ``True``

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

"""

\_\_constants\_\_ = ['p', 'margin', 'reduction']

def \_\_init\_\_(self, p=1, margin=1., weight=None, size\_average=None,

reduce=None, reduction='mean'):

super(MultiMarginLoss, self).\_\_init\_\_(weight, size\_average, reduce, reduction)

if p != 1 and p != 2:

raise ValueError("only p == 1 and p == 2 supported")

assert weight is None or weight.dim() == 1

self.p = p

self.margin = margin

def forward(self, input, target):

return F.multi\_margin\_loss(input, target, p=self.p, margin=self.margin,

weight=self.weight, reduction=self.reduction)

class TripletMarginLoss(\_Loss):

r"""Creates a criterion that measures the triplet loss given an input

tensors :math:`x1`, :math:`x2`, :math:`x3` and a margin with a value greater than :math:`0`.

This is used for measuring a relative similarity between samples. A triplet

is composed by `a`, `p` and `n` (i.e., `anchor`, `positive examples` and `negative

examples` respectively). The shapes of all input tensors should be

:math:`(N, D)`.

The distance swap is described in detail in the paper `Learning shallow

convolutional feature descriptors with triplet losses`\_ by

V. Balntas, E. Riba et al.

The loss function for each sample in the mini-batch is:

.. math::

L(a, p, n) = \max \{d(a\_i, p\_i) - d(a\_i, n\_i) + {\rm margin}, 0\}

where

.. math::

d(x\_i, y\_i) = \left\lVert {\bf x}\_i - {\bf y}\_i \right\rVert\_p

Args:

margin (float, optional): Default: :math:`1`.

p (int, optional): The norm degree for pairwise distance. Default: :math:`2`.

swap (bool, optional): The distance swap is described in detail in the paper

`Learning shallow convolutional feature descriptors with triplet losses` by

V. Balntas, E. Riba et al. Default: ``False``.

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 reduce is ``False``. Default: ``True``

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 sum of the output will be divided by the number of

elements in the output, ``'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'``

Shape:

- Input: :math:`(N, D)` where :math:`D` is the vector dimension.

- Output: scalar. If :attr:`reduction` is ``'none'``, then :math:`(N)`.

>>> triplet\_loss = nn.TripletMarginLoss(margin=1.0, p=2)

>>> anchor = torch.randn(100, 128, requires\_grad=True)

>>> positive = torch.randn(100, 128, requires\_grad=True)

>>> negative = torch.randn(100, 128, requires\_grad=True)

>>> output = triplet\_loss(anchor, positive, negative)

>>> output.backward()

.. \_Learning shallow convolutional feature descriptors with triplet losses:

<http://www.bmva.org/bmvc/2016/papers/paper119/index.html>

"""

\_\_constants\_\_ = ['margin', 'p', 'eps', 'swap', 'reduction']

def \_\_init\_\_(self, margin=1.0, p=2., eps=1e-6, swap=False, size\_average=None,

reduce=None, reduction='mean'):

super(TripletMarginLoss, self).\_\_init\_\_(size\_average, reduce, reduction)

self.margin = margin

self.p = p

self.eps = eps

self.swap = swap

def forward(self, anchor, positive, negative):

return F.triplet\_margin\_loss(anchor, positive, negative, margin=self.margin, p=self.p,

eps=self.eps, swap=self.swap, reduction=self.reduction)

class CTCLoss(\_Loss):

r"""The Connectionist Temporal Classification loss.

Calculates loss between a continuous (unsegmented) time series and a target sequence. CTCLoss sums over the

probability of possible alignments of input to target, producing a loss value which is differentiable

with respect to each input node. The alignment of input to target is assumed to be "many-to-one", which

limits the length of the target sequence such that it must be :math:`\leq` the input length.

Args:

blank (int, optional): blank label. Default :math:`0`.

reduction (string, optional): Specifies the reduction to apply to the output:

``'none'`` | ``'mean'`` | ``'sum'``. ``'none'``: no reduction will be applied,

``'mean'``: the output losses will be divided by the target lengths and

then the mean over the batch is taken. Default: ``'mean'``

zero\_infinity (bool, optional):

Whether to zero infinite losses and the associated gradients.

Default: ``False``

Infinite losses mainly occur when the inputs are too short

to be aligned to the targets.

Shape:

- Log\_probs: Tensor of size :math:`(T, N, C)`,

where :math:`T = \text{input length}`,

:math:`N = \text{batch size}`, and

:math:`C = \text{number of classes (including blank)}`.

The logarithmized probabilities of the outputs (e.g. obtained with

:func:`torch.nn.functional.log\_softmax`).

- Targets: Tensor of size :math:`(N, S)` or

:math:`(\operatorname{sum}(\text{target\\_lengths}))`,

where :math:`N = \text{batch size}` and

:math:`S = \text{max target length, if shape is } (N, S)`.

It represent the target sequences. Each element in the target

sequence is a class index. And the target index cannot be blank (default=0).

In the :math:`(N, S)` form, targets are padded to the

length of the longest sequence, and stacked.

In the :math:`(\operatorname{sum}(\text{target\\_lengths}))` form,

the targets are assumed to be un-padded and

concatenated within 1 dimension.

- Input\_lengths: Tuple or tensor of size :math:`(N)`,

where :math:`N = \text{batch size}`. It represent the lengths of the

inputs (must each be :math:`\leq T`). And the lengths are specified

for each sequence to achieve masking under the assumption that sequences

are padded to equal lengths.

- Target\_lengths: Tuple or tensor of size :math:`(N)`,

where :math:`N = \text{batch size}`. It represent lengths of the targets.

Lengths are specified for each sequence to achieve masking under the

assumption that sequences are padded to equal lengths. If target shape is

:math:`(N,S)`, target\_lengths are effectively the stop index

:math:`s\_n` for each target sequence, such that ``target\_n = targets[n,0:s\_n]`` for

each target in a batch. Lengths must each be :math:`\leq S`

If the targets are given as a 1d tensor that is the concatenation of individual

targets, the target\_lengths must add up to the total length of the tensor.

- Output: scalar. If :attr:`reduction` is ``'none'``, then

:math:`(N)`, where :math:`N = \text{batch size}`.

Examples::

>>> # Target are to be padded

>>> T = 50 # Input sequence length

>>> C = 20 # Number of classes (including blank)

>>> N = 16 # Batch size

>>> S = 30 # Target sequence length of longest target in batch (padding length)

>>> S\_min = 10 # Minimum target length, for demonstration purposes

>>>

>>> # Initialize random batch of input vectors, for \*size = (T,N,C)

>>> input = torch.randn(T, N, C).log\_softmax(2).detach().requires\_grad\_()

>>>

>>> # Initialize random batch of targets (0 = blank, 1:C = classes)

>>> target = torch.randint(low=1, high=C, size=(N, S), dtype=torch.long)

>>>

>>> input\_lengths = torch.full(size=(N,), fill\_value=T, dtype=torch.long)

>>> target\_lengths = torch.randint(low=S\_min, high=S, size=(N,), dtype=torch.long)

>>> ctc\_loss = nn.CTCLoss()

>>> loss = ctc\_loss(input, target, input\_lengths, target\_lengths)

>>> loss.backward()

>>>

>>>

>>> # Target are to be un-padded

>>> T = 50 # Input sequence length

>>> C = 20 # Number of classes (including blank)

>>> N = 16 # Batch size

>>>

>>> # Initialize random batch of input vectors, for \*size = (T,N,C)

>>> input = torch.randn(T, N, C).log\_softmax(2).detach().requires\_grad\_()

>>> input\_lengths = torch.full(size=(N,), fill\_value=T, dtype=torch.long)

>>>

>>> # Initialize random batch of targets (0 = blank, 1:C = classes)

>>> target\_lengths = torch.randint(low=1, high=T, size=(N,), dtype=torch.long)

>>> target = torch.randint(low=1, high=C, size=(sum(target\_lengths),), dtype=torch.long)

>>> ctc\_loss = nn.CTCLoss()

>>> loss = ctc\_loss(input, target, input\_lengths, target\_lengths)

>>> loss.backward()

Reference:

A. Graves et al.: Connectionist Temporal Classification:

Labelling Unsegmented Sequence Data with Recurrent Neural Networks:

<https://www.cs.toronto.edu/~graves/icml_2006.pdf>

Note:

In order to use CuDNN, the following must be satisfied: :attr:`targets` must be

in concatenated format, all :attr:`input\_lengths` must be `T`. :math:`blank=0`,

:attr:`target\_lengths` :math:`\leq 256`, the integer arguments must be of

dtype :attr:`torch.int32`.

The regular implementation uses the (more common in PyTorch) `torch.long` dtype.

Note:

In some circumstances when using the CUDA backend with 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``.

Please see the notes on :doc:`/notes/randomness` for background.

"""

\_\_constants\_\_ = ['blank', 'reduction']

def \_\_init\_\_(self, blank=0, reduction='mean', zero\_infinity=False):

super(CTCLoss, self).\_\_init\_\_(reduction=reduction)

self.blank = blank

self.zero\_infinity = zero\_infinity

def forward(self, log\_probs, targets, input\_lengths, target\_lengths):

return F.ctc\_loss(log\_probs, targets, input\_lengths, target\_lengths, self.blank, self.reduction,

self.zero\_infinity)

# TODO: L1HingeEmbeddingCriterion

# TODO: MSECriterion weight

# TODO: ClassSimplexCriterion