Criterions

`Criterions` are helpful to train a neural network. Given an input and a target, they compute a gradient according to a given loss function.

- Classification criterions:
  - `BCECriterion` : binary cross-entropy for `Sigmoid` (two-class version of `ClassNLLCriterion`);
  - `ClassNLLCriterion` : negative log-likelihood for `LogSoftMax` (multi-class);
  - `CrossEntropyCriterion` : combines `LogSoftMax` and `ClassNLLCriterion`;
  - `ClassSimplexCriterion` : A simplex embedding criterion for classification.
  - `MarginCriterion` : two class margin-based loss;
  - `SoftMarginCriterion` : two class softmargin-based loss;
  - `MultiMarginCriterion` : multi-class margin-based loss;
  - `MultiLabelMarginCriterion` : multi-class multi-classification margin-based loss;
  - `MultiLabelSoftMarginCriterion` : multi-class multi-classification loss based on binary cross-entropy;

- Regression criterions:
  - `AbsCriterion` : measures the mean absolute value of the element-wise difference between input;
  - `SmoothL1Criterion` : a smooth version of the AbsCriterion;
  - `MSECriterion` : mean square error (a classic);
  - `SpatialAutoCropMSECriterion` : Spatial mean square error when the input is spatially smaller than the target, by only comparing their spatial overlap;
  - `DistKLDivCriterion` : Kullback–Leibler divergence (for fitting continuous probability distributions);

- Embedding criterions (measuring whether two inputs are similar or dissimilar):
  - `HingeEmbeddingCriterion` : takes a distance as input;
  - `L1HingeEmbeddingCriterion` : L1 distance between two inputs;
  - `CosineEmbeddingCriterion` : cosine distance between two inputs;
  - `DistanceRatioCriterion` : Probabilistic criterion for training siamese model with triplets.

- Miscelaneus criterions:
  - `MultiCriterion` : a weighted sum of other criterions each applied to the same input and target;
  - `ParallelCriterion` : a weighted sum of other criterions each applied to a

different input and target;
- `MarginRankingCriterion` : ranks two inputs;

Criterion

This is an abstract class which declares methods defined in all criterions.
This class is serializable.

[output] forward(input, target)

Given an `input` and a `target`, compute the loss function associated to the criterion and return the result.
In general `input` and `target` are `Tensor`s, but some specific criterions might require some other type of object.

The `output` returned should be a scalar in general.

The state variable `self.output` should be updated after a call to `forward()`.

[gradInput] backward(input, target)

Given an `input` and a `target`, compute the gradients of the loss function associated to the criterion and return the result.
In general `input`, `target` and `gradInput` are `Tensor`s, but some specific criterions might require some other type of object.

The state variable `self.gradInput` should be updated after a call to `backward()`.

State variable: output

State variable which contains the result of the last `forward(input, target)` call.

State variable: gradInput

State variable which contains the result of the last `backward(input, target)` call.

AbsCriterion

```
criterion = nn.AbsCriterion()
```

Creates a criterion that measures the mean absolute value of the element-wise difference between input `x` and target `y`:

```
loss(x, y)  = 1/n \sum |x_i - y_i|
```

If `x` and `y` are `d`-dimensional `Tensor`s with a total of `n` elements, the sum operation still operates over all the elements, and divides by `n`.

The division by `n` can be avoided if one sets the internal variable `sizeAverage` to `false`:

```
criterion = nn.AbsCriterion()
criterion.sizeAverage = false
```

ClassNLLCriterion

```
criterion = nn.ClassNLLCriterion([weights])
```

The negative log likelihood criterion. It is useful to train a classification problem with `n` classes.
If provided, the optional argument `weights` 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()` is expected to contain *log-probabilities* of each class: `input` has to be a 1D `Tensor` of size `n`.
Obtaining log-probabilities in a neural network is easily achieved by adding a `LogSoftMax` layer in the last layer of your neural network.

You may use `CrossEntropyCriterion` instead, if you prefer not to add an extra layer to your network.
This criterion expects a class index (1 to the number of class) as `target` when calling `forward(input, target)` and `backward(input, target)`.

The loss can be described as:

```
loss(x, class) = -x[class]
```

or in the case of the `weights` argument it is specified as follows:

```
loss(x, class) = -weights[class] * x[class]
```

Due to the behaviour of the backend code, it is necessary to set sizeAverage to false when calculating losses *in non-batch mode*.

The following is a code fragment showing how to make a gradient step given an input `x`, a desired output `y` (an integer `1` to `n`, in this case `n = 2` classes), a network `mlp` and a learning rate `learningRate`:

```
function gradUpdate(mlp, x, y, learningRate)
   local criterion = nn.ClassNLLCriterion()
   local pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   mlp:zeroGradParameters()
   local t = criterion:backward(pred, y)
   mlp:backward(x, t)
   mlp:updateParameters(learningRate)
end
```

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed for each minibatch.

CrossEntropyCriterion

```
criterion = nn.CrossEntropyCriterion([weights])
```

This criterion combines `LogSoftMax` and `ClassNLLCriterion` in one single class.

It is useful to train a classification problem with `n` classes.
If provided, the optional argument `weights` 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()` is expected to contain scores for each class: `input` has to be a 1D `Tensor` of size `n`.
This criterion expect a class index (1 to the number of class) as `target` when calling `forward(input, target)` and `backward(input, target)`.

The loss can be described as:

```
loss(x, class) = -log(exp(x[class]) / (\sum_j exp(x[j])))
               = -x[class] + log(\sum_j exp(x[j]))
```

or in the case of the `weights` argument being specified:

```
loss(x, class) = weights[class] * (-x[class] + log(\sum_j
exp(x[j])))
```

Due to the behaviour of the backend code, it is necessary to set sizeAverage to false when calculating losses *in non-batch mode*.

```
crit = nn.CrossEntropyCriterion(weights)
crit.nll.sizeAverage = false
```

The losses are averaged across observations for each minibatch.

ClassSimplexCriterion

```
criterion = nn.ClassSimplexCriterion(nClasses)
```

ClassSimplexCriterion implements a criterion for classification.
It learns an embedding per class, where each class' embedding is a point on an (N-1)-dimensional simplex,
where N is the number of classes.

The `input` given through a `forward()` is expected to be the output of a Normalized Linear layer with no bias:
- `input` has to be a 1D `Tensor` of size `n` for a single sample
- a 2D `Tensor` of size `batchSize x n` for a mini-batch of samples

This Criterion is best used in combination with a neural network where the last layers are:
- a weight-normalized bias-less Linear layer. Example source code
- followed by an output normalization layer (nn.Normalize).

The loss is described in detail in the paper Scale-invariant learning and convolutional networks.

The following is a code fragment showing how to make a gradient step given an input `x`, a desired output `y` (an integer `1` to `n`, in this case `n = 30` classes), a network `mlp` and a learning rate `learningRate`:

```
nInput = 10
nClasses = 30
nHidden = 100
mlp = nn.Sequential()
mlp:add(nn.Linear(nInput, nHidden)):add(nn.ReLU())
mlp:add(nn.NormalizedLinearNoBias(nHidden, nClasses))
mlp:add(nn.Normalize(2))

criterion = nn.ClassSimplexCriterion(nClasses)

function gradUpdate(mlp, x, y, learningRate)
   pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   mlp:zeroGradParameters()
   local t = criterion:backward(pred, y)
   mlp:backward(x, t)
   mlp:updateParameters(learningRate)
end
```

This criterion also provides two helper functions `getPredictions(input)` and `getTopPrediction(input)` that return the raw predictions and the top prediction index respectively, given an input sample.

DistKLDivCriterion

```
criterion = nn.DistKLDivCriterion()
```

The Kullback–Leibler divergence criterion.
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 ClassNLLCriterion, the `input` given through a `forward()` is expected to contain *log-probabilities*, however unlike ClassNLLCriterion, `input` is not restricted to a 1D or 2D vector (as the criterion is applied element-wise).

This criterion expect a `target` `Tensor` of the same size as the `input` `Tensor` when calling `forward(input, target)` and `backward(input, target)`.

The loss can be described as:

```
loss(x, target) = 1/n \sum(target_i * (log(target_i) - x_i))
```

By default, the losses are averaged for each minibatch over observations *as well as* over dimensions. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

BCECriterion

```
criterion = nn.BCECriterion([weights])
```

Creates a criterion that measures the Binary Cross Entropy between the target and the output:

```
loss(o, t) = - 1/n sum_i (t[i] * log(o[i]) + (1 - t[i]) * log(1 -
o[i]))
```

or in the case of the weights argument being specified:

```
loss(o, t) = - 1/n sum_i weights[i] * (t[i] * log(o[i]) + (1 -
t[i]) * log(1 - o[i]))
```

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the outputs `o[i]` should be numbers between 0 and 1, for instance, the output of an `nn.Sigmoid` layer and should be interpreted as the probability of predicting `t[i] = 1`. Note `t[i]` can be either 0 or 1.

By default, the losses are averaged for each minibatch over observations *as well as* over dimensions. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

MarginCriterion

```
criterion = nn.MarginCriterion([margin])
```

Creates a criterion that optimizes a two-class classification hinge loss (margin-based loss) between input `x` (a `Tensor` of dimension `1`) and output `y` (which is a tensor containing either `1` s or `-1` s).
`margin`, if unspecified, is by default `1`.

```
loss(x, y) = sum_i (max(0, margin - y[i]*x[i])) / x:nElement()
```

The normalization by the number of elements in the input can be disabled by setting `self.sizeAverage` to `false`.

Example

```
function gradUpdate(mlp, x, y, criterion, learningRate)
   local pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   local gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(learningRate)
end

mlp = nn.Sequential()
mlp:add(nn.Linear(5, 1))
```

```
x1 = torch.rand(5)
x1_target = torch.Tensor{1}
x2 = torch.rand(5)
x2_target = torch.Tensor{-1}
criterion=nn.MarginCriterion(1)

for i = 1, 1000 do
   gradUpdate(mlp, x1, x1_target, criterion, 0.01)
   gradUpdate(mlp, x2, x2_target, criterion, 0.01)
end

print(mlp:forward(x1))
print(mlp:forward(x2))

print(criterion:forward(mlp:forward(x1), x1_target))
print(criterion:forward(mlp:forward(x2), x2_target))
```

gives the output:

```
 1.0043
[torch.Tensor of dimension 1]


-1.0061
[torch.Tensor of dimension 1]

0
0
```

i.e. the mlp successfully separates the two data points such that they both have a `margin` of `1`, and hence a loss of `0`.

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

SoftMarginCriterion

```
criterion = nn.SoftMarginCriterion()
```

Creates a criterion that optimizes a two-class classification logistic loss between input `x` (a `Tensor` of dimension `1`) and output `y` (which is a tensor containing either `1`s or `-1`s).

```
loss(x, y) = sum_i (log(1 + exp(-y[i]*x[i]))) / x:nElement()
```

The normalization by the number of elements in the input can be disabled by setting `self.sizeAverage` to `false`.

Example

```
function gradUpdate(mlp, x, y, criterion, learningRate)
   local pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   local gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(learningRate)
end

mlp = nn.Sequential()
mlp:add(nn.Linear(5, 1))

x1 = torch.rand(5)
x1_target = torch.Tensor{1}
x2 = torch.rand(5)
x2_target = torch.Tensor{-1}
criterion=nn.SoftMarginCriterion(1)

for i = 1, 1000 do
   gradUpdate(mlp, x1, x1_target, criterion, 0.01)
   gradUpdate(mlp, x2, x2_target, criterion, 0.01)
end

print(mlp:forward(x1))
print(mlp:forward(x2))

print(criterion:forward(mlp:forward(x1), x1_target))
print(criterion:forward(mlp:forward(x2), x2_target))
```

gives the output:

```
 0.7471
[torch.DoubleTensor of size 1]

-0.9607
[torch.DoubleTensor of size 1]

0.38781049558836
0.32399356957564
```

i.e. the mlp successfully separates the two data points.

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

MultiMarginCriterion

```
criterion = nn.MultiMarginCriterion(p, [weights], [margin])
```

Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss) between input `x` (a `Tensor` of dimension 1) and output `y` (which is a target class index, `1` <= `y` <= `x:size(1)`):

```
loss(x, y) = sum_i(max(0, (margin - x[y] + x[i]))^p) / x:size(1)
```

where `i == 1` to `x:size(1)` and `i ~= y`.
Note that this criterion also works with 2D inputs and 1D targets.

Optionally, you can give non-equal weighting on the classes by passing a 1D `weights` tensor into the constructor.
The loss function then becomes:

```
loss(x, y) = sum_i(max(0, w[y] * (margin - x[y] - x[i]))^p) /
x:size(1)
```

This criterion is especially useful for classification when used in conjunction with a module ending in the following output layer:

```
mlp = nn.Sequential()
mlp:add(nn.Euclidean(n, m)) -- outputs a vector of distances
mlp:add(nn.MulConstant(-1)) -- distance to similarity
```

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

MultiLabelMarginCriterion

```
criterion = nn.MultiLabelMarginCriterion()
```

Creates a criterion that optimizes a multi-class multi-classification hinge loss (margin-based loss) between input `x` (a 1D `Tensor`) and output `y` (which is a 1D `Tensor` of target class indices):

```
loss(x, y) = sum_ij(max(0, 1 - (x[y[j]] - x[i]))) / x:size(1)
```

where `i == 1` to `x:size(1)`, `j == 1` to `y:size(1)`, `y[j] ~= 0`, and `i ~= y[j]` for all `i` and `j`.
Note that this criterion also works with 2D inputs and targets.

`y` and `x` must have the same size.
The criterion only considers the first non zero `y[j]` targets.
This allows for different samples to have variable amounts of target classes:

```
criterion = nn.MultiLabelMarginCriterion()
input = torch.randn(2, 4)
target = torch.Tensor{{1, 3, 0, 0}, {4, 0, 0, 0}} -- zero-values
are ignored
criterion:forward(input, target)
```

MultiLabelSoftMarginCriterion

MultiLabelSoftMarginCriterion

```
criterion = nn.MultiLabelSoftMarginCriterion()
```

Creates a criterion that optimizes a multi-label one-versus-all loss based on max-entropy, between input `x` (a 1D `Tensor`) and target `y` (a binary 1D `Tensor`):

```
loss(x, y) = - sum_i (y[i] log( exp(x[i]) / (1 + exp(x[i]))) + (1-
y[i]) log(1/(1+exp(x[i])))) / x:nElement()
```

where `i == 1` to `x:nElement()`, `y[i] in {0,1}`.
Note that this criterion also works with 2D inputs and targets.

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

MSECriterion

```
criterion = nn.MSECriterion()
```

Creates a criterion that measures the mean squared error between `n` elements in the input `x` and output `y`:

```
loss(x, y) = 1/n \sum |x_i - y_i|^2 .
```

If `x` and `y` are `d`-dimensional `Tensor`s with a total of `n` elements, the sum operation still operates over all the elements, and divides by `n`.
The two `Tensor`s must have the same number of elements (but their sizes might be different).

The division by `n` can be avoided if one sets the internal variable `sizeAverage` to `false`:

```
criterion = nn.MSECriterion()
criterion.sizeAverage = false
```

By default, the losses are averaged over observations for each minibatch. However, if the field

`sizeAverage` is set to `false`, the losses are instead summed.

SpatialAutoCropMSECriterion

```
criterion = nn.SpatialAutoCropMSECriterion()
```

Creates a criterion that measures the mean squared error between the input and target, even if the target is spatially larger than the input. It achieves this by center-cropping the target to the same spatial resolution as the input, the mean squared error is then calculated between the input and this cropped target.

If the input and cropped target tensors are `d`-dimensional `Tensor`s with a total of `n` elements, the sum operation operates over all the elements, and divides by `n`.

The division by `n` can be avoided if one sets the internal variable `sizeAverage` to `false`:

```
criterion = nn.SpatialAutoCropMSECriterion()
criterion.sizeAverage = false
```

MultiCriterion

```
criterion = nn.MultiCriterion()
```

This returns a Criterion which is a weighted sum of other Criterion.
Criterions are added using the method:

```
criterion:add(singleCriterion [, weight])
```

where `weight` is a scalar (default 1). Each criterion is applied to the same `input` and `target`.

Example :

```
input = torch.rand(2,10)
target = torch.IntTensor{1,8}
nll = nn.ClassNLLCriterion()
nll2 = nn.CrossEntropyCriterion()
mc = nn.MultiCriterion():add(nll, 0.5):add(nll2)
output = mc:forward(input, target)
```

ParallelCriterion

```
criterion = nn.ParallelCriterion([repeatTarget])
```

This returns a Criterion which is a weighted sum of other Criterion.
Criterions are added using the method:

```
criterion:add(singleCriterion [, weight])
```

where `weight` is a scalar (default 1). The criterion expects an `input` and `target` table. Each criterion is applied to the commensurate `input` and `target` element in the tables. However, if `repeatTarget=true`, the `target` is repeatedly presented to each criterion (with a different `input`).

Example :

```
input = {torch.rand(2,10), torch.randn(2,10)}
target = {torch.IntTensor{1,8}, torch.randn(2,10)}
nll = nn.ClassNLLCriterion()
mse = nn.MSECriterion()
pc = nn.ParallelCriterion():add(nll, 0.5):add(mse)
output = pc:forward(input, target)
```

SmoothL1Criterion

```
criterion = nn.SmoothL1Criterion()
```

Creates a criterion that can be thought of as a smooth version of the `AbsCriterion`. It uses a squared term if the absolute element-wise error falls below 1. It is less sensitive to outliers than the `MSECriterion` and in some cases prevents exploding gradients (e.g. see “Fast R-CNN” paper by Ross Girshick).

```
                      ⎧ 0.5 * (x_i - y_i)^2, if |x_i - y_i| < 1
loss(x, y) = 1/n \sum ⎨
                      ⎩ |x_i - y_i| - 0.5,   otherwise
```

If `x` and `y` are `d`-dimensional `Tensor`s with a total of `n` elements, the sum operation still operates over all the elements, and divides by `n`.

The division by `n` can be avoided if one sets the internal variable `sizeAverage` to `false`:

```
criterion = nn.SmoothL1Criterion()
criterion.sizeAverage = false
```

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

HingeEmbeddingCriterion

```
criterion = nn.HingeEmbeddingCriterion([margin])
```

Creates a criterion that measures the loss given an input `x` which is a 1-dimensional vector and a label `y` (`1` or `-1`).
This is usually used for measuring whether two inputs are similar or dissimilar, e.g. using the L1 pairwise distance, and is typically used for learning nonlinear embeddings or semi-supervised learning.

```
                  ⎧ x_i,                  if y_i ==  1
loss(x, y) = 1/n ⎨
                  ⎩ max(0, margin - x_i), if y_i == -1
```

If `x` and `y` are `n`-dimensional `Tensor`s, the sum operation still operates over all the elements, and divides by `n` (this can be avoided if one sets the internal variable `sizeAverage` to `false`). The `margin` has a default value of `1`, or can be set in the constructor.

Example

```
-- imagine we have one network we are interested in, it is called
"p1_mlp"
p1_mlp = nn.Sequential(); p1_mlp:add(nn.Linear(5, 2))

-- But we want to push examples towards or away from each other so
we make another copy
-- of it called p2_mlp; this *shares* the same weights via the set
command, but has its
-- own set of temporary gradient storage that's why we create it
again (so that the gradients
-- of the pair don't wipe each other)
p2_mlp = nn.Sequential(); p2_mlp:add(nn.Linear(5, 2))
p2_mlp:get(1).weight:set(p1_mlp:get(1).weight)
p2_mlp:get(1).bias:set(p1_mlp:get(1).bias)

-- we make a parallel table that takes a pair of examples as input.
-- They both go through the same (cloned) mlp
prl = nn.ParallelTable()
prl:add(p1_mlp)
prl:add(p2_mlp)

-- now we define our top level network that takes this parallel
table
-- and computes the pairwise distance betweem the pair of outputs
mlp = nn.Sequential()
mlp:add(prl)
mlp:add(nn.PairwiseDistance(1))

-- and a criterion for pushing together or pulling apart pairs
crit = nn.HingeEmbeddingCriterion(1)

-- lets make two example vectors
x = torch.rand(5)
y = torch.rand(5)
```

```
-- Use a typical generic gradient update function
function gradUpdate(mlp, x, y, criterion, learningRate)
local pred = mlp:forward(x)
local err = criterion:forward(pred, y)
local gradCriterion = criterion:backward(pred, y)
mlp:zeroGradParameters()
mlp:backward(x, gradCriterion)
mlp:updateParameters(learningRate)
end

-- push the pair x and y together, notice how then the distance
between them given
-- by print(mlp:forward({x, y})[1]) gets smaller
for i = 1, 10 do
   gradUpdate(mlp, {x, y}, 1, crit, 0.01)
   print(mlp:forward({x, y})[1])
end

-- pull apart the pair x and y, notice how then the distance
between them given
-- by print(mlp:forward({x, y})[1]) gets larger

for i = 1, 10 do
   gradUpdate(mlp, {x, y}, -1, crit, 0.01)
   print(mlp:forward({x, y})[1])
end
```

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

L1HingeEmbeddingCriterion

```
criterion = nn.L1HingeEmbeddingCriterion([margin])
```

Creates a criterion that measures the loss given an input `x` = `{x1, x2}`, a table of two `Tensor`s, and a label `y` (`1` or `-1`): this is used for measuring whether two inputs are similar or dissimilar, using the L1 distance, and is typically used for learning nonlinear embeddings or semi-supervised learning.

```
             ⎧ ||x1 - x2||_1,                  if y ==  1
loss(x, y) = ⎨
             ⎩ max(0, margin - ||x1 - x2||_1), if y == -1
```

The `margin` has a default value of `1`, or can be set in the constructor.

CosineEmbeddingCriterion

```
criterion = nn.CosineEmbeddingCriterion([margin])
```

Creates a criterion that measures the loss given an input `x` = `{x1, x2}`, a table of two `Tensor`s, and a `Tensor` label `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.

`margin` should be a number from `-1` to `1`, `0` to `0.5` is suggested.
`Forward` and `Backward` have to be used alternately. If `margin` is missing, the default value is `0`.

The loss function for each sample is:

```
             ⎧ 1 - cos(x1, x2),              if y ==  1
loss(x, y) = ⎨
             ⎩ max(0, cos(x1, x2) - margin), if y == -1
```

For batched inputs, if the internal variable `sizeAverage` is equal to `true`, the loss function averages the loss over the batch samples; if `sizeAverage` is `false`, then the loss function sums over the batch samples. By default, `sizeAverage` equals to `true`.

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

DistanceRatioCriterion

Ref A. Unsupervised Learning through Spatial Contrasting

```
criterion = nn.DistanceRatioCriterion(sizeAverage)
```

This criterion is probabilistic treatment of margin cost. The model is trained using sample triplets `{Xs, Xa, Xd}` where `Xa` is anchor sample, `Xs` is sample similar to anchor sample and `Xd` is a sample not similar to anchor sample. Let `Ds` be distance between embeddings of `{Xs, Xa}` and `Dd` be distance between embeddings of `{Xa, Xd}` then the loss is defined as follow

```
    loss = -log( exp(-Ds) / ( exp(-Ds) + exp(-Dd) ) )
```

Sample example

```
    torch.setdefaulttensortype("torch.FloatTensor")

    require 'nn'

    -- triplet : with batchSize of 32 and dimensionality 512
    sample = {torch.rand(32, 512), torch.rand(32, 512),
torch.rand(32, 512)}

    embeddingModel = nn.Sequential()
    embeddingModel:add(nn.Linear(512, 96)):add(nn.ReLU())

    tripleModel = nn.ParallelTable()
    tripleModel:add(embeddingModel)
    tripleModel:add(embeddingModel:clone('weight', 'bias',
                                         'gradWeight', 'gradBias'))
    tripleModel:add(embeddingModel:clone('weight', 'bias',
                                         'gradWeight', 'gradBias'))

    -- Similar sample distance w.r.t anchor sample
    posDistModel = nn.Sequential()
    posDistModel:add(nn.NarrowTable(1,2)):add(nn.PairwiseDistance())

    -- Different sample distance w.r.t anchor sample
    negDistModel = nn.Sequential()
    negDistModel:add(nn.NarrowTable(2,2)):add(nn.PairwiseDistance())

    distanceModel =
nn.ConcatTable():add(posDistModel):add(negDistModel)
```

```
-- Complete Model
model = nn.Sequential():add(tripleModel):add(distanceModel)

-- DistanceRatioCriterion
criterion = nn.DistanceRatioCriterion(true)

-- Forward & Backward
output = model:forward(sample)
loss   = criterion:forward(output)
dLoss  = criterion:backward(output)
model:backward(sample, dLoss)
```

MarginRankingCriterion

```
criterion = nn.MarginRankingCriterion(margin)
```

Creates a criterion that measures the loss given an input `x = {x1, x2}`, a table of two `Tensor`s of size 1 (they contain only scalars), and a label `y` (`1` or `-1`).
In batch mode, `x` is a table of two `Tensor`s of size `batchsize`, and `y` is a `Tensor` of size `batchsize` containing `1` or `-1` for each corresponding pair of elements in the input `Tensor`.

If `y == 1` then it assumed the first input should be ranked higher (have a larger value) than the second input, and vice-versa for `y == -1`.

The loss function is:

```
loss(x, y) = max(0, -y * (x[1] - x[2]) + margin)
```

For batched inputs, if the internal variable `sizeAverage` is equal to `true`, the loss function averages the loss over the batch samples; if `sizeAverage` is `false`, then the loss function sums over the batch samples. By default, `sizeAverage` equals to `true`.
By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

Example

```
p1_mlp = nn.Linear(5, 2)
p2_mlp = p1_mlp:clone('weight', 'bias')

prl = nn.ParallelTable()
prl:add(p1_mlp)
prl:add(p2_mlp)

mlp1 = nn.Sequential()
mlp1:add(prl)
mlp1:add(nn.DotProduct())

mlp2 = mlp1:clone('weight', 'bias')

mlpa = nn.Sequential()
prla = nn.ParallelTable()
prla:add(mlp1)
prla:add(mlp2)
mlpa:add(prla)

crit = nn.MarginRankingCriterion(0.1)

x=torch.randn(5)
y=torch.randn(5)
z=torch.randn(5)

-- Use a typical generic gradient update function
function gradUpdate(mlp, x, y, criterion, learningRate)
   local pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   local gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(learningRate)
end

for i = 1, 100 do
   gradUpdate(mlpa, {{x, y}, {x, z}}, 1, crit, 0.01)
   if true then
      o1 = mlp1:forward{x, y}[1]
      o2 = mlp2:forward{x, z}[1]
      o = crit:forward(mlpa:forward{{x, y}, {x, z}}, 1)
      print(o1, o2, o)
   end
end
```

```
print "--"

for i = 1, 100 do
   gradUpdate(mlpa, {{x, y}, {x, z}}, -1, crit, 0.01)
   if true then
      o1 = mlp1:forward{x, y}[1]
      o2 = mlp2:forward{x, z}[1]
      o = crit:forward(mlpa:forward{{x, y}, {x, z}}, -1)
      print(o1, o2, o)
   end
end
```