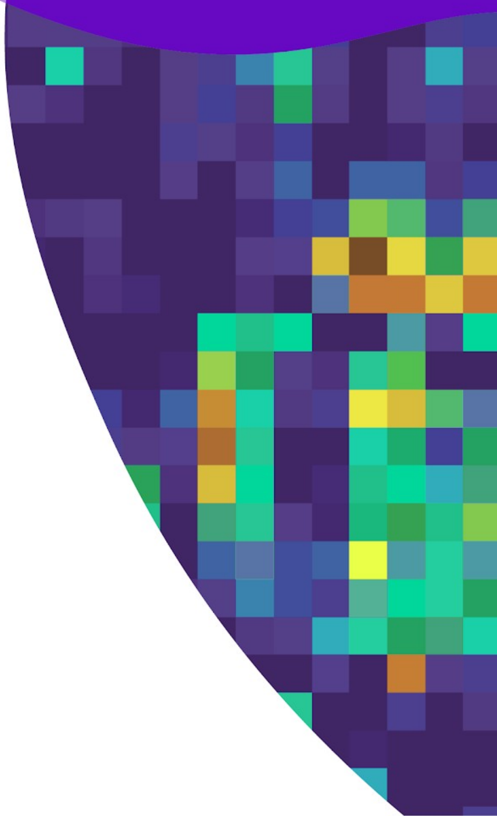
# Loss Functions

In PyTorch & how to use them



# MSE – `nn.MSELoss()`

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input $x$ and target $y$ .

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

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2 ,$$

where $N$ is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{'mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{'sum'.} \end{cases}$$

$x$ and $y$ are tensors of arbitrary shapes with a total of $n$ elements each.

The sum operation still operates over all the elements, and divides by $n$ .

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

# L1 – nn.L1Loss()

Creates a criterion that measures the mean absolute error (MAE) between each element in the input $x$ and target $y$ .

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

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = \left| x_n - y_n \right|,$$

where $N$ is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{'mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{'sum'.} \end{cases}$$

$x$ and $y$ are tensors of arbitrary shapes with a total of $n$ elements each.

The sum operation still operates over all the elements, and divides by $n$ .

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

# `nn.SmoothL1Loss()`

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:

$$\text{loss}(x, y) = \frac{1}{n} \sum_i z_i$$

where $z_i$ is given by:

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$$

$x$ and $y$ arbitrary shapes with a total of $n$ elements each the sum operation still operates over all the elements, and divides by $n$ .

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

# NLL – `nn.NLLoss()`

The negative log likelihood loss. It is useful to train a classification problem with C classes.

If provided, the optional argument `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 *target* that this loss expects should be a class index in the range $[0, C-1]$ 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 `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} x_{n,y_n}, \quad w_c = \text{weight}[c] \cdot 1\{c \neq \text{ignore\_index}\},$$

where $x$ is the input, $y$ is the target, $w$ is the weight, and $N$ is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^N \frac{1}{\sum_{n=1}^N w_{y_n}} l_n, & \text{if reduction} = \text{'mean';} \\ \sum_{n=1}^N l_n, & \text{if reduction} = \text{'sum'.} \end{cases}$$

# CE – `nn.CrossEntropyLoss()`

This criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class.

It is useful when training a classification problem with *C* classes. If provided, the optional argument `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 $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case (described later).

This criterion expects a class index in the range $[0, C-1]$ 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).

# CE – nn.CrossEntropyLoss()

The loss can be described as:

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

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

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

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 $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$, where $K$ is the number of dimensions, and a target of appropriate shape (see below).

# nn.AdaptiveLogSoftmaxWithLoss()

Efficient softmax approximation as described in Efficient softmax approximation for GPUs by Edouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou.

Adaptive softmax is an approximate strategy for training models with large output spaces. It is most effective when the label distribution is highly imbalanced, for example in natural language modelling, where the word frequency distribution approximately follows the Zipf's law.

Adaptive softmax partitions the labels into several clusters, according to their frequency. These clusters may contain different number of targets each. Additionally, clusters containing less frequent labels assign lower dimensional embeddings to those labels, which speeds up the computation. For each minibatch, only clusters for which at least one target is present are evaluated.

The idea is that the clusters which are accessed frequently (like the first one, containing most frequent labels), should also be cheap to compute – that is, contain a small number of assigned labels.

## BCE – `nn.BCELoss()`

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

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[ y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right],$$

where $N$ is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{'mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{'sum'.} \end{cases}$$

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets $y$ should be numbers between 0 and 1.

# KLDiv – `nn.KLDivLoss()`

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.

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

$$l(x, y) = L = \{l_1, \dots, l_N\}, \quad l_n = y_n \cdot (\log y_n - x_n)$$

where the index $N$ spans all dimensions of `input` and $L$ has the same shape as `input`. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{'mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{'sum'.} \end{cases}$$

In default `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.

## Poisson – `nn.PoissonNLLLoss()`

Negative log likelihood loss with Poisson distribution of target.

The loss can be described as:

$$\text{target} \sim \text{Poisson}(\text{input})\text{loss}(\text{input}, \text{target}) = \text{input} - \text{target} * \log(\text{input}) + \log(\text{target}!)$$

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.

# BCE – `nn.BCELoss()`

Notice that if $x_n$ is either 0 or 1, one of the log terms would be mathematically undefined in the above loss equation. PyTorch chooses to set $\log(0) = -\infty$, since $\lim_{x\to 0} \log(x) = -\infty$. However, an infinite term in the loss equation is not desirable for several reasons.

For one, if either $y_n = 0$ or $(1 - y_n) = 0$, 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 $\lim_{x\to 0} \frac{d}{dx} \log(x) = \infty$. This would make BCELoss's backward method nonlinear with respect to $x_n$, 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.

# BCE – `nn.BCEWithLogitsLoss()`

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 `reduction` set to `'none'`) loss can be described as:

$$\ell(x,y) = L = \{l_1,\dots,l_N\}^\top, \quad l_n = -w_n \left[ y_n \cdot \log \sigma(x_n) + (1-y_n) \cdot \log(1-\sigma(x_n)) \right],$$

where $N$ is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x,y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{'mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{'sum'.} \end{cases}$$

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.

# BCE – `nn.BCEWithLogitsLoss()`

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:

$$\ell_c(x, y) = L_c = \{l_{1,c}, \dots, l_{N,c}\}^\top, \quad l_{n,c} = -w_{n,c} \left[p_c y_{n,c} \cdot \log \sigma(x_{n,c}) + (1 - y_{n,c}) \cdot \log(1 - \sigma(x_{n,c}))\right],$$

where $c$ is the class number ($c > 1$ for multi-label binary classification, $c = 1$ for single-label binary classification), $n$ is the number of the sample in the batch and $p_c$ is the weight of the positive answer for the class $c$.

$p_c > 1$ increases the recall, $p_c < 1$ 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 $\frac{300}{100} = 3$. The loss would act as if the dataset contains $3 \times 100 = 300$ positive examples.

# Ranking – `nn.MarginRankingLoss()`

Creates a criterion that measures the loss given inputs $x1$ , $x2$ , two 1D mini-batch *Tensors*, and a label 1D mini-batch tensor $y$ (containing 1 or -1).

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 for each sample in the mini-batch is:

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

## nn.TripletMarginLoss()

Creates a criterion that measures the triplet loss given an input tensors $x1$ , $x2$ , $x3$ and a margin with a value greater than $0$ . 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 $(N, D)$ .

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:

$$L(a, p, n) = \max\{d(a_i, p_i) - d(a_i, n_i) + \mathrm{margin}, 0\}$$

where

$$d(x_i, y_i) = \left\|\mathbf{x}_i - \mathbf{y}_i\right\|_p$$

# nn.TripletMarginLoss()

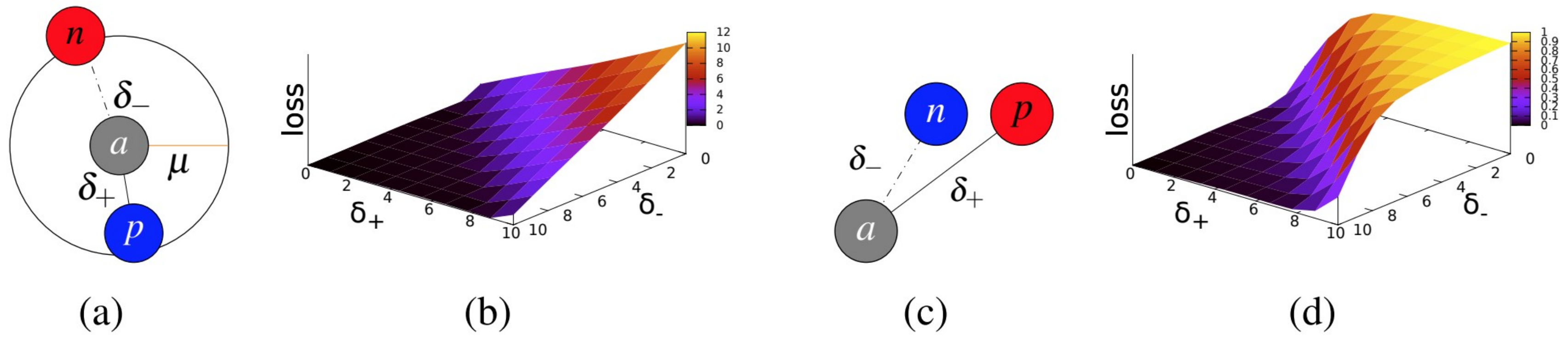


Figure 1: (a) Margin ranking loss. It seeks to push *n* outside the circle defined by the margin $\mu$, and pull *p* inside. (b) Margin ranking loss values in function of $\delta_-, \delta_+$ (c) Ratio loss. It seeks to force $\delta_+$ to be much smaller than $\delta_-$. (d) Ratio loss values in function of $\delta_-, \delta_+$

$$\boldsymbol{\delta}_+ = ||f(\boldsymbol{a}) - f(\boldsymbol{p})||_2 \text{ and } \boldsymbol{\delta}_- = ||f(\boldsymbol{a}) - f(\boldsymbol{n})||_2.$$

$$\lambda(\delta_+, \delta_*) = max(0, \mu + \delta_+ - \delta_*)$$

# nn.SoftMarginLoss()

Creates a criterion that optimizes a two-class classification logistic loss between input tensor $x$ and target tensor $y$ (containing 1 or -1).

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

# nn.MultiLabelMarginLoss()

Creates a criterion that optimizes a multi-class multi-classification hinge loss (margin-based loss) between input $x$ (a 2D mini-batch *Tensor*) and output $y$ (which is a 2D *Tensor* of target class indices). For each sample in the mini-batch:

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

where $x \in \{0, \ \cdots , \ \text{x.size}(0) - 1\}$, $y \in \{0, \ \cdots , \ \text{y.size}(0) - 1\}$, $0 \leq y[j] \leq \text{x.size}(0) - 1$, and $i \neq y[j]$ for all $i$ and $j$ .

$y$ and $x$ 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.

# nn.MultiLabelSoftMarginLoss()

Creates a criterion that optimizes a multi-label one-versus-all loss based on max-entropy, between input $x$ and target $y$ of size $(N, C)$. For each sample in the minibatch:

$$loss(x, y) = -\frac{1}{C} * \sum_i y[i] * \log((1 + \exp(-x[i]))^{-1}) + (1 - y[i]) * \log\left(\frac{\exp(-x[i])}{(1 + \exp(-x[i]))}\right)$$

where $i \in \{0, \ \cdots, \ \mathrm{x.nElement}() - 1\}$, $y[i] \in \{0, \ 1\}$.

# nn.MultiMarginLoss()

Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss) between input $x$ (a 2D mini-batch *Tensor*) and output $y$ (which is a 1D tensor of target class indices, $0 \le y \le \text{x.size}(1) - 1$):

For each mini-batch sample, the loss in terms of the 1D input $x$ and scalar output $y$ is:

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

where $x \in \{0, \cdots, \text{x.size}(0) - 1\}$ and $i \neq y$.

Optionally, you can give non-equal weighting on the classes by passing a 1D `weight` tensor into the constructor.

The loss function then becomes:

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

# nn.HingeEmbeddingLoss()

Measures the loss given an input tensor $x$ and a labels tensor $y$ (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 $x$ , and is typically used for learning nonlinear embeddings or semi-supervised learning.

The loss function for $n$ -th sample in the mini-batch is

$$l_n = \begin{cases} x_n, & \text{if } y_n = 1, \\ \max\{0, \Delta - x_n\}, & \text{if } y_n = -1, \end{cases}$$

and the total loss functions is

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{'mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{'sum'.} \end{cases}$$

where $L = \{l_1, \dots, l_N\}^\top$ .

## nn.CosineEmbeddingLoss()

Creates a criterion that measures the loss given input tensors $x_1$ , $x_2$ 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.

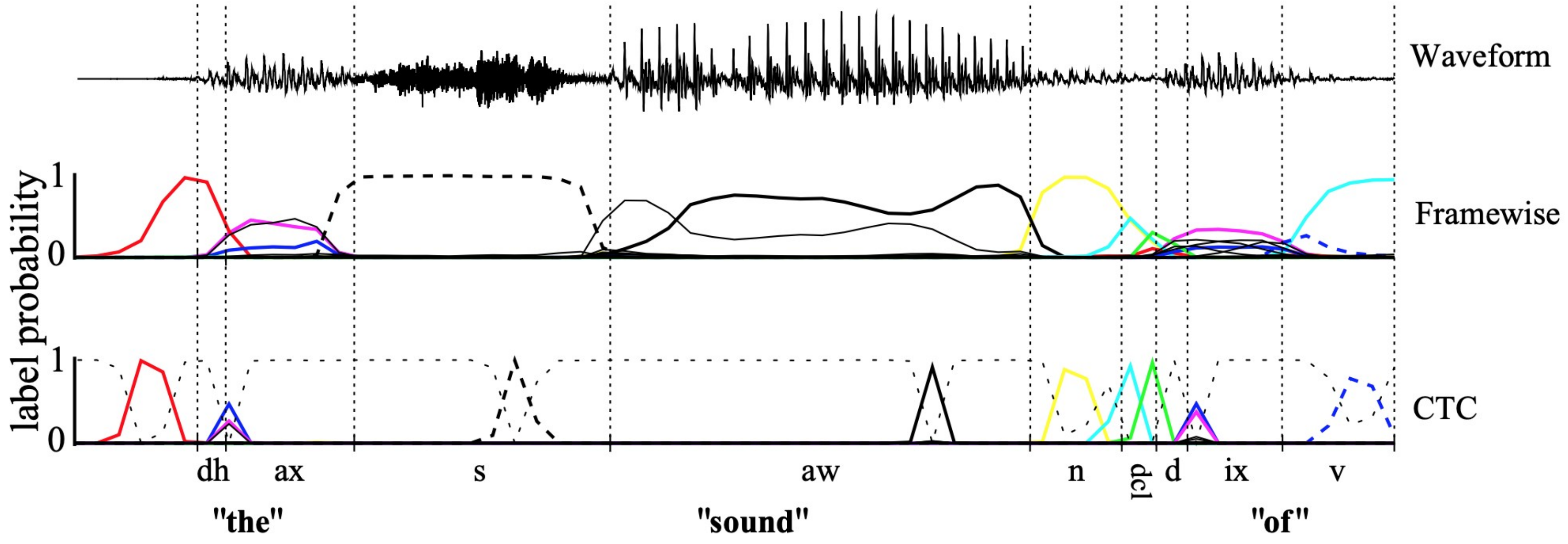
The loss function for each sample is:

$$\text{loss}(x, y) = \begin{cases} 1 - \cos(x_1, x_2), & \text{if } y = 1 \\ \max(0, \cos(x_1, x_2) - \text{margin}), & \text{if } y = -1 \end{cases}$$

## CTC – `nn.CTCLoss()`

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 $\leq$ the input length.


Waveform
Framewise
CTC
label probability
dh
ax
s
aw
n
dcl
d
ix
v
"the"
"sound"
"of"