BCELOSS

CLASS `torch.nn.BCELoss(weight=None, size_average=None, reduce=None, reduction='mean')` [SOURCE]

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

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.

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 multiplying 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.

Parameters

- **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 `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 `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 `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `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: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: $(*)$, where $*$ means any number of dimensions.
- Target: $(*)$, same shape as the input.
- Output: scalar. If `reduction` is `'none'`, then $(*)$, 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()
```

❮ Previous Next ❯

Docs

Access comprehensive developer documentation for PyTorch

View Docs

Tutorials

Get in-depth tutorials for beginners and advanced developers

View Tutorials

Resources

Find development resources and get your questions answered

View Resources

PyTorch

Get Started

Features

Ecosystem

Blog

Contributing

Resources

Tutorials

Docs

Discuss

Github Issues

Brand Guidelines

Stay Connected