

Understanding Cross Entropy Loss - 859251b8a

USER

CrossEntropyLoss

```
classtorch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean', label_smoothing=0.0)[source][source]
```

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

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 the unnormalized logits for each class (which do not need to be positive or sum to 1, in general). input has to be a Tensor of size

```
(
C
)
(C) for unbatched input,
```

```
(
m
i
n
i
b
a
t
c
h
,
C
)
(minibatch,C) or
```

```
(
m
i
n
i
b
a
t
c
h
,
C
,
d
1
,
d
2
,
.
.
.
,
d
K
)
(minibatch,C,d
```

```
1
,d
2
```

,...,d
K

) with
K
 \geq
1

$K \geq 1$ for the K-dimensional case. The last being useful for higher dimension inputs, such as computing cross entropy loss per-pixel for 2D images.

The target that this criterion expects should contain either:

Class indices in the range

[
0
,
C
)
[0,C) where
C

C is the number of classes; if ignore_index is specified, this loss also accepts this class index (this index may not necessarily be in the class range). The unreduced (i.e. with reduction set to 'none') loss for this case can be described as:

ℓ
(
x
,
y
)
=
L
=
{
|
1
,
...
,
|
N
}
T
,
|
n
=
-
w
y
n
log

exp

(
x
n
,
y
n
)
 \sum
c
=
1
C

exp

(
x
n
,
c
)
.
1
{
y
n
=
ignore_index
}
 $\ell(x,y)=L=\{l$
1

,...,l
N

}
T
,l
n

=-w
y
n

log
 \sum
c=1
C

exp(x
n,c

)
exp(x
n,y
n

)

·1{y
n

=ignore_index}
where

x
x is the input,
y
y is the target,
w
w is the weight,
C
C is the number of classes, and
N
N spans the minibatch dimension as well as
d

$$\ell(x, y) = \sum_{k=1}^K \sum_{d=1}^D \sum_{n=1}^N \dots$$

for the K-dimensional case. If reduction is not 'none' (default 'mean'), then

$$\ell(x, y) = \sum_{n=1}^N \sum_{k=1}^K \sum_{d=1}^D \dots$$

ignore_index

$$\ell(x, y) = \sum_{n=1}^N \dots$$

if reduction = 'mean';

$$\ell(x, y) = \frac{1}{N} \sum_{n=1}^N \dots$$

if reduction = 'sum'.

$$\ell(x, y) = \sum_{n=1}^N \dots$$

$$\sum_{n=1}^N$$

$$w_{y_n}$$

$$\cdot 1\{y_n$$

$$= \text{ignore_index}\}$$

$$1$$

$$\sum_{n=1}^N$$

$$1$$

$$,$$

if reduction='mean';
if reduction='sum'.

Note that this case is equivalent to applying LogSoftmax on an input, followed by NLLLoss.

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

$$\ell(x, y) = L = \{1, \dots, N\}^T$$

$$c = 1$$

$$C_w c \log$$

$$\exp$$

$$\left(x_n, c \right)$$

$$\sum_{i=1}^C \exp$$

$$\left(x_n, y_n, c \right)$$

$$\ell(x, y) = L = \{l_1, \dots, l_N\}$$

$$\tau, l_n$$

$$c = 1$$

$$\sum_C w_c \log \sum_{i=1}^C \exp(x_{n,i}) \exp(x_{n,c})$$

$$y_{n,c}$$

where
 x
 x is the input,
 y
 y is the target,
 w
 w is the weight,
 C
 C is the number of classes, and
 N
 N spans the minibatch dimension as well as
 d
 1
 $,$
 $.$
 $.$
 $.$
 $,$
 d
 k
 d
 1
 $,...,d$
 k

for the K -dimensional case. If reduction is not 'none' (default 'mean'), then

ℓ
 $($
 x
 $,$
 y
 $)$
 $=$
 $\{$
 \sum
 n
 $=$
 1
 N
 $|$
 n
 N
 $,$
 if reduction
 $=$
 'mean';
 \sum
 n
 $=$
 1
 N
 $|$
 n
 $,$
 if reduction
 $=$
 'sum'.
 $\ell(x,y) = \{$
 N
 \sum
 $n=1$
 N
 $|$

n

,
 $\sum_{n=1}^N$

|
n

,

if reduction='mean';
if reduction='sum'.

Note

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

Parameters

weight (Tensor, optional) – a manual rescaling weight given to each class. If given, has to be a Tensor of size C and floating point dtype

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

ignore_index (int, optional) – Specifies a target value that is ignored and does not contribute to the input gradient. When size_average is True, the loss is averaged over non-ignored targets. Note that ignore_index is only applicable when the target contains class indices.

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 (str, optional) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the weighted mean of the output is taken, 'sum': the output will be summed. Note: 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'

label_smoothing (float, optional) – A float in [0.0, 1.0]. Specifies the amount of smoothing when computing the loss, where 0.0 means no smoothing. The targets become a mixture of the original ground truth and a uniform distribution as described in Rethinking the Inception Architecture for Computer Vision. Default:

0.0
0.0.

Shape:

Input: Shape

(
C
)
(C),
(
N
,
C
)
(N,C) or
(
N
,
C


```
,  
d  
1  
,  
d  
2  
,  
.  
.  
.  
,  
d  
K  
)  
(N,C,d  
1
```

```
,d  
2
```

```
,...,d  
K
```

```
) with  
K  
≥  
1
```

$K \geq 1$ in the case of K-dimensional loss.

Target: If containing class indices, shape

```
(  
)  
(),  
(  
N  
)  
(N) or  
(  
N
```

```
,  
d  
1
```

```
,  
d  
2
```

```
,  
.  
.  
.
```

```
,  
d  
K  
)  
(N,d  
1
```

```
,d  
2
```

```
,...,d  
K
```

```
) with  
K  
≥  
1
```

$K \geq 1$ in the case of K-dimensional loss where each value should be between

[

```

0
,
C
)
[0,C). If containing class probabilities, same shape as the input and each value should be between
[
0
,
1
]
[0,1].

```

Output: If reduction is 'none', shape

```

(
)
(),
(
N
)
(N) or
(
N
,
d
1
,
d
2
,
.
.
.
,
d
K
)
(N,d
1

```

```

,d
2

```

```

,...,d
K

```

```

) with
K
≥
1

```

$K \geq 1$ in the case of K-dimensional loss, depending on the shape of the input. Otherwise, scalar.

where:

```

C
=
number of classes
N
=
batch size
C=
N=

```

```

number of classes
batch size

```

Examples:

```
>>> # Example of target with class indices
>>> 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()
>>>
>>> # Example of target with class probabilities
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randn(3, 5).softmax(dim=1)
>>> output = loss(input, target)
>>> output.backward()
```

can you explain how CrossEntropy works for multi class classification?

BOT

How Cross Entropy Works for Multi-Class Classification

CrossEntropyLoss measures how well a model's predicted logits (unnormalized scores) match the true classes. For multi-class classification, your network typically outputs a tensor of shape $(\text{batch size}, C)$, where C is the number of classes.

1. **Logits and Softmax**:

Instead of directly producing probabilities, most neural networks output logits—raw, unnormalized scores for each class. CrossEntropyLoss internally applies a softmax operation to these logits, turning them into a probability distribution for each batch element.

2. **Targets**:

- **Class indices**: Each sample in the batch has an integer label from $[0, C)$. CrossEntropyLoss picks out the probability of the correct class (after softmax) and computes the negative log-probability.
- **Class probabilities**: Sometimes your target is already a probability distribution (e.g., a one-hot or soft label). In this case, CrossEntropyLoss measures the cross entropy between the predicted distribution and the target distribution.

3. **Loss Computation**:

For each sample n , the loss is:

$$l_n = -\sum_{c=1}^C w_c \log(y_{n,c})$$

where

- $x_{n,c}$ is the logit for class c ,
- $y_{n,c}$ is the target distribution (1.0 for the true class in the one-hot case, or a probability in the soft-label case),
- w_c is an optional per-class weight.

4. **Reduction**:

- **Mean**: Averages the per-sample losses.
- **Sum**: Sums them up.
- **None**: Returns the full, unreduced loss for each sample.

By combining a softmax with a negative log-likelihood, CrossEntropyLoss directs the network to assign higher logits to the correct classes and lower logits to the others. This is the standard approach for multi-class classification tasks in PyTorch.

USER

Instead of directly producing probabilities, most neural networks output logits—raw, unnormalized scores for each class. CrossEntropyLoss internally applies a softmax operation to these logits, turning them into a probability distribution for each batch element.

so for a batch size of 4 and 27 classes

we would take raw inputs and convert them to probs using softmax

where for each of those 4 examples we will have 27 classes where each value in 27 classes is prob and the sum of those 27 classes in each of 4 examples will be equal to 1?

BOT

Yes. Each of the 4 examples gets its own 27-dimensional probability distribution

When you apply `softmax` across the 27 classes for a single example, all 27 probabilities sum to 1. So, if you have a batch size of 4, you effectively get 4 separate softmax calculations (one per example). Each of those 4 results is a 27-dimensional probability distribution summing to 1.

USER

Targets:

Class indices: Each sample in the batch has an integer label from

```
[  
0  
,
```

```
)
```

[0,C). CrossEntropyLoss picks out the probability of the correct class (after softmax) and computes the negative log-probability.

Class probabilities: Sometimes your target is already a probability distribution (e.g., a one-hot or soft label). In this case, CrossEntropyLoss measures the cross entropy between the predicted distribution and the target distribution.

Loss Computation:

For each sample

n, the loss is:

=

-

\sum

=

1

,

log

(

exp

(

,

)

\sum

=

1

exp

(

,

)

$$- \sum_{c=1}^C w_c \log \left(\frac{\exp(x_{n,i})}{\sum_{c=1}^C \exp(x_{n,c})} \right)$$

),
where

,

$x_{n,c}$

is the logit for class

c ,

,

$y_{n,c}$

is the target distribution (1.0 for the true class in the one-hot case, or a probability in the soft-label case),

w_c

c

is an optional per-class weight.

so for each example we are picking out which label index is 1 and then for that we get probability, then we compute the loss.

i understand the idea. the thing i dont understand is when we calculate the loss we are only doing it for positive class and ignore the rest. so when we update with new grads we are only pushing weights in the directly that increase the prob of positive class. so how does negative class weights decrease to ensure their probability reduces?

BOT

Why negative class probabilities also decrease

It might seem at first glance that when you compute the loss for a single *correct* class, you only “reward” that correct class, and somehow ignore the others. In reality, the softmax + cross-entropy formulation affects *all class logits* in the gradient step. Here’s the gist:

1. **Cross-entropy with softmax**

The loss for a single example n (assuming class indices) is

$$L = -\log \text{Bigl} \left(\frac{\exp(x_{n,y})}{\sum_{i=1}^C \exp(x_{n,i})} \right) \text{Bigr},$$

where y is the true class for that example. Although the formula looks like it only uses the true class’s logit $x_{n,y}$, the gradient depends on the entire softmax distribution.

2. **Gradient w.r.t. logits**

When you do backprop, the partial derivative of the loss with respect to each logit $x_{n,i}$ has a neat form:

- For the *correct* class y :

$$\frac{\partial L}{\partial x_{n,y}} = p_{n,y} - 1$$

where $p_{n,y}$ is the softmax probability your model predicted for the correct class.

- For every *other* class $i \neq y$:

$$\frac{\partial L}{\partial x_{n,i}} = p_{n,i}$$

Notice that each $p_{n,i}$ for $i \neq y$ explicitly appears in the gradient. This means if the network thinks “class i ” has high probability when it’s not really correct, that logit gets pushed down (negative update for $x_{n,i}$).

3. **Competition among classes**

Because the softmax normalizes all classes to sum to 1, increasing one class’s logit effectively decreases the relative probability of the others. Meanwhile, the gradient with respect to the logit for each *incorrect* class pushes those logits lower if they’re too large.

Bottom line: Even though the cross-entropy formula takes the log of the *correct* class probability, the gradient uses the entire predicted distribution. Incorrect classes with high predicted probability get penalized just as surely as the correct class is rewarded.

USER

i understand this statements intuition:

Because the softmax normalizes all classes to sum to 1, increasing one class’s logit effectively decreases the relative probability of the others. Meanwhile, the gradient with respect to the logit for each incorrect class pushes those logits lower if they’re too large.

but i dont really understand

Cross-entropy with softmax
The loss for a single example

n (assuming class indices) is

=
-
log

(
exp

(

,

)

$$L = -\log \left(\frac{\exp(x_{n,y})}{\sum_{i=1}^C \exp(x_{n,i})} \right)$$

where

y is the true class for that example. Although the formula looks like it only uses the true class's logit

, the gradient depends on the entire softmax distribution.
 Gradient w.r.t. logits
 When you do backprop, the partial derivative of the loss with respect to each logit

has a neat form:
 For the correct class

$$\frac{\partial L}{\partial x_{n,y}}$$

$$=$$

$$=$$

$$=$$

$$=$$

$$= \frac{-\exp(x_{n,y})}{\sum_{i=1}^C \exp(x_{n,i})}$$

$\frac{\partial x}{\partial n,y}$

$\frac{\partial L}{\partial n,y}$

$= p_{n,y}$

-1
where

,

$p_{n,y}$

is the softmax probability your model predicted for the correct class.
For every other class

\neq

i

$= y:$
 $\frac{\partial}{\partial}$

$\frac{\partial}{\partial}$

,

$=$

,

$\frac{\partial x}{\partial n,i}$

$\frac{\partial L}{\partial n,i}$

$= p_{n,i}$

Notice that each

,

$p_{n,i}$

for

\neq

i

$= y$ explicitly appears in the gradient. This means if the network thinks “class

i ” has high probability when it’s not really correct, that logit gets pushed down (negative update for

,

x
n,i
)

can u explain it in simpler terms. be detailed.

BOT

A More Intuitive Explanation

Think of the cross-entropy + softmax process as follows:

1. **Softmax Creates a Probability Distribution**

For a single data example (n) with logits $(x_{n,0}, x_{n,1}, \dots, x_{n,C-1})$, softmax turns these raw scores into probabilities that sum to 1:

$$p_{n,i} = \frac{\exp(x_{n,i})}{\sum_{k=0}^{C-1} \exp(x_{n,k})}.$$

Each $(p_{n,i})$ is how much the model believes “class (i) ” is the correct answer.

2. **Cross-Entropy Loss Rewards the Correct Class**

If the correct class is (y) , the cross-entropy loss can be written as

$$L = -\log(p_{n,y}).$$

This means that if the model assigns a high probability to the correct class $(p_{n,y})$, the loss is small (since (\log) of a number close to 1 is 0). If $(p_{n,y})$ is low, the loss is large.

3. **Gradients: Correct Class Gets Pushed Up, Others Get Pushed Down**

When you do backpropagation, you take partial derivatives of (L) with respect to each logit $(x_{n,i})$. Despite the formula looking like it only uses $(\log(p_{n,y}))$, *all classes* appear in the softmax, so *all* logits affect the final probabilities. The most important piece is:

- **Gradient w.r.t. correct class** $(x_{n,y})$:

$$\frac{\partial L}{\partial x_{n,y}} = p_{n,y} - 1.$$

If $(p_{n,y})$ is less than 1, then $(p_{n,y} - 1)$ is negative. In gradient descent, a negative derivative means we *increase* $(x_{n,y})$ next update, strengthening the correct class's logit.

- **Gradient w.r.t. incorrect classes** $(x_{n,i})$ for $(i \neq y)$:

$$\frac{\partial L}{\partial x_{n,i}} = p_{n,i}.$$

Because $(p_{n,i})$ is positive (between 0 and 1), the sign of the gradient makes the network *decrease* $(x_{n,i})$ next update, pushing down the model's confidence in the wrong classes.

4. **Why This Lowers Wrong-Class Probabilities**

Since softmax probabilities all sum to 1, making the correct class more probable automatically reduces the probability of the others. And the explicit gradient for *every* wrong class ensures that if the model is giving too much probability to the wrong classes, their logits get moved down.

A Quick Numerical Example

Say we have 3 classes, and the model is predicting:

- $(p_{n,0} = 0.1)$
- $(p_{n,1} = 0.7)$
- $(p_{n,2} = 0.2)$

If the true class is (0) :

- Gradient w.r.t. $x_{n,0}$ is $(0.1 - 1 = -0.9)$. In gradient descent, a negative gradient on $x_{n,0}$ will *increase* $x_{n,0}$ after the update, so the probability (for class 0) will go up.
- Gradient w.r.t. $x_{n,1}$ is (0.7) . This positive gradient means the weight update will *decrease* $x_{n,1}$, lowering the probability for class 1.
- Gradient w.r.t. $x_{n,2}$ is (0.2) . Similarly, $x_{n,2}$ also goes down.

Over many iterations, class 0's logit gets bigger, while classes 1 and 2 get smaller, causing the softmax distribution to favor class 0 more strongly.

USER

we get

$\frac{\partial x}{\partial n, y}$

$\frac{\partial L}{\partial n, y}$

$= p_{n, y}$

$- 1.$

and
Gradient w.r.t. incorrect classes (

,

$x_{n, i}$

for

\neq

i

$= y):$
 $\frac{\partial}{\partial}$

$\frac{\partial}{\partial}$

,

$=$

,

$\frac{\partial x}{\partial n, i}$

$\frac{\partial L}{\partial n, i}$

$= p_{n, i}$

$-$

from softmax inside cross entropy?

BOT

Yes. Those partial derivatives are obtained by applying the chain rule to the cross-entropy loss when combined with (or derived from) the softmax function. Specifically, if we denote the softmax probabilities by:

$$p_{n,i} = \frac{\exp(x_{n,i})}{\sum_{k=0}^{C-1} \exp(x_{n,k})},$$

then for a single training example with the true class (y) , the cross-entropy loss is:

$$L = -\log p_{n,y}.$$

Taking partial derivatives of (L) with respect to each logit $(x_{n,i})$ leads to:

1. **For the correct class** $(i = y)$:

$$\frac{\partial L}{\partial x_{n,y}} = p_{n,y} - 1.$$

2. **For each incorrect class** $(i \neq y)$:

$$\frac{\partial L}{\partial x_{n,i}} = p_{n,i}.$$

This difference arises from both how the softmax is normalized across all classes and the log in cross-entropy.

USER

same thing for using sigmoid in crossentropy would be - * (ylogy + (1-y log (1-y)) correct?

BOT

Yes, that is the standard binary cross-entropy formula. For a single logit (x) , you pass it through the sigmoid function:

$$p = \sigma(x) = \frac{1}{1 + e^{-x}},$$

and the binary cross-entropy loss for one sample is

$$L = -\bigl[y \log(p) + (1 - y) \log(1 - p)\bigr].$$

Here, (y) is either 0 or 1, and (p) is the probability assigned to the positive class.

USER

so binary cross-entropy uses sigmoid where as multiclass cross entropy would use softmax hence the difference in formulas?

BOT

Yes, exactly. In **binary classification**, you only have two classes (often denoted 0 or 1), so it's sufficient to pass a single logit through a ***sigmoid***, which outputs a probability in $[0, 1]$. The corresponding **binary cross-entropy** formula is:

$$\frac{\partial L}{\partial x}$$

$$L = - \text{Bigl}[y \log(\text{sigma}(x)) + (1 - y)\log\text{bigl}(1 - \text{sigma}(x)\text{bigr)}\text{Bigr}]$$

In **multi-class classification** with (C) classes, you need a (C) -dimensional output (one logit per class). You then apply a **softmax** operation to form a probability distribution over these (C) classes. The corresponding **multi-class cross-entropy** loss is:

$$L = - \sum_{c=1}^C y_c \log \text{bigl}(\text{softmax}(x)_c \text{bigr}),$$

where $\text{softmax}(x)_c = \frac{\exp(x_c)}{\sum_{k=1}^C \exp(x_k)}$.

Because sigmoid outputs a single probability and is used for binary tasks, whereas softmax produces a vector of probabilities that sum to 1 for multi-class tasks, the two formulas look slightly different.