# EE-559 – Deep learning

## 5.1. Cross-entropy loss

François Fleuret
https://fleuret.org/ee559/
Mon Feb 18 13:34:39 UTC 2019

We can train a model for classification using a regression loss such as the MSE using a "one-hot vector" encoding: given a training set

$$(x_n, y_n) \in \mathbb{R}^D \times \{1, \dots, C\}, \; n = 1, \dots, N,$$

we would convert the labels into a tensor $z \in \mathbb{R}^{N \times C}$, with

$$\forall n, \; z_{n,m} = \begin{cases} 1 & \text{if } \; m = y_n \\ 0 & \text{otherwise.} \end{cases}$$

For instance, with $N = 5$ and $C = 3$, we would have

$$\begin{pmatrix} 2 \\ 1 \\ 1 \\ 3 \\ 2 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

Training can be achieved by matching the output of the model with these binary values in a MSE sense.

However, MSE is justified with a Gaussian noise around a target value that makes sense geometrically. Beside being conceptually wrong for classification, in practice it penalizes responses "too strongly on the right side".

As we will see, the criterion of choice for classification is the cross-entropy.



We can generalize the logistic regression to a multi-class setup with $f_1, \dots, f_C$ functionals that we interpret as "logit values"

$$P(Y = y \mid X = x, W = w) = \frac{1}{Z} \exp f_y(x; w) = \frac{\exp f_y(x; w)}{\sum_k \exp f_k(x; w)},$$

from which

$$\begin{aligned}
&\log \mu_W(w \mid \mathscr{D} = \mathbf{d}) \\
&\qquad = \log \frac{\mu_{\mathscr{D}}(\mathbf{d} \mid W = w)\, \mu_W(w)}{\mu_{\mathscr{D}}(\mathbf{d})} \\
&\qquad = \log \mu_{\mathscr{D}}(\mathbf{d} \mid W = w) + \log \mu_W(w) - \log Z \\
&\qquad = \sum_n \log \mu_{\mathscr{D}}(x_n, y_n \mid W = w) + \log \mu_W(w) - \log Z \\
&\qquad = \sum_n \log P(Y = y_n \mid X = x_n, W = w) + \log \mu_W(w) - \log Z' \\
&\qquad = \underbrace{\sum_n \log \left( \frac{\exp f_{y_n}(x; w)}{\sum_k \exp f_k(x; w)} \right)}_{\text{Depends on the outputs}} + \underbrace{\log \mu_W(w)}_{\text{Depends on } w} - \log Z'.
\end{aligned}$$

If we ignore the penalty on $w$, it makes sense to minimize the average

$$\mathscr{L}(w) = -\frac{1}{N}\sum_{n=1}^{N} \log \underbrace{\left(\frac{\exp f_{y_n}(x_n; w)}{\sum_k \exp f_k(x_n; w)}\right)}_{\hat{P}_w(Y=y_n|X=x_n)}.$$

Given two distributions $p$ and $q$, their **cross-entropy** is defined as

$$\mathbb{H}(p, q) = -\sum_k p(k) \log q(k),$$

with the convention that $0 \log 0 = 0$. So we can re-write

$$\begin{aligned} -\log\left(\frac{\exp f_{y_n}(x_n; w)}{\sum_k \exp f_k(x_n; w)}\right) &= -\log \hat{P}_w(Y = y_n \mid X = x_n) \\ &= -\sum_k \delta_{y_n}(k) \log \hat{P}_w(Y = k \mid X = x_n) \\ &= \mathbb{H}\left(\delta_{y_n}, \hat{P}_w(Y = \cdot \mid X = x_n)\right). \end{aligned}$$

So $\mathscr{L}$ above is the average of the cross-entropy between the deterministic "true" posterior $\delta_{y_n}$ and the estimated $\hat{P}_w(Y = \cdot \mid X = x_n)$.



This is precisely the value of `torch.nn.CrossEntropyLoss`.

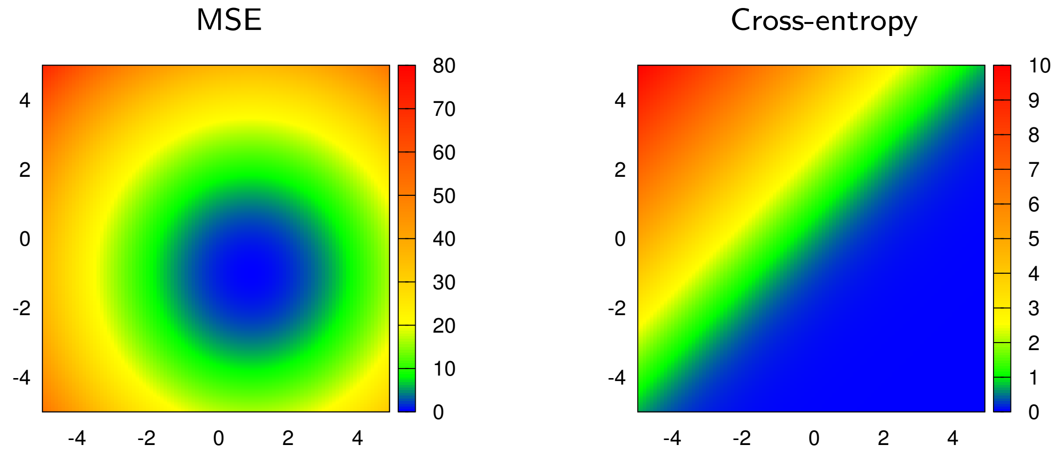
```
>>> f = torch.tensor([[-1., -3., 4.], [-3., 3., -1.]])
>>> target = torch.tensor([0, 1])
>>> criterion = torch.nn.CrossEntropyLoss()
>>> criterion(f, target)
tensor(2.5141)
```

and indeed

$$-\frac{1}{2}\left(\log \frac{e^{-1}}{e^{-1} + e^{-3} + e^{4}} + \log \frac{e^{3}}{e^{-3} + e^{3} + e^{-1}}\right) \simeq 2.5141.$$

The range of values is 0 for perfectly classified samples, $\log(C)$ if the posterior is uniform, and up to $+\infty$ if the posterior distribution is "worst" than uniform.

Let's consider the loss for a single sample in a two-class problem, with a predictor with two output values. The $x$ axis here is the activation of the correct output unit, and the $y$ axis is the activation of the other one.




$$\mathscr{L} = (x-1)^2 + (y+1)^2 \qquad \mathscr{L} = -\log\left(\frac{e^x}{e^x + e^y}\right)$$

MSE incorrectly penalizes outputs which are perfectly valid for prediction, contrary to cross-entropy.



The cross-entropy loss can be seen as the composition of a "log soft-max" to normalize the score into logs of probabilities

$$(\alpha_1, \dots, \alpha_C) \mapsto \left(\log \frac{\exp \alpha_1}{\sum_k \exp \alpha_k}, \dots, \log \frac{\exp \alpha_C}{\sum_k \exp \alpha_k}\right),$$

which can be done with the `torch.nn.LogSoftmax` module, and a read-out of the normalized score of the correct class

$$\mathscr{L}(w) = -\frac{1}{N} \sum_{n=1}^{N} f_{y_n}(x_n; w),$$

which is implemented by the `torch.nn.NLLLoss` criterion.

```
>>> f = torch.tensor([[-1., -3., 4.], [-3., 3., -1.]])
>>> target = torch.tensor([0, 1])
>>> model = nn.LogSoftmax(dim = 1)
>>> criterion = torch.nn.NLLLoss()
>>> criterion(model(f), target)
tensor(2.5141)
```

Hence, if a network should compute log-probabilities, it may have a `torch.nn.LogSoftmax` final layer, and be trained with `torch.nn.NLLLoss`.

The mapping

$$(\alpha_1, \dots, \alpha_C) \mapsto \left( \frac{\exp \alpha_1}{\sum_k \exp \alpha_k}, \dots, \frac{\exp \alpha_C}{\sum_k \exp \alpha_k} \right)$$

is called soft-max since it computes a “soft arg-max Boolean label.”

```
>>> y = torch.tensor([[-10., -10., 10., -5. ],
...                   [  3.,   0.,  0.,  0. ],
...                   [  1.,   2.,  3.,  4. ]])
>>> f = torch.nn.Softmax(1)
>>> f(y)
tensor([[ 2.0612e-09,  2.0612e-09,  1.0000e+00,  3.0590e-07],
        [ 8.7005e-01,  4.3317e-02,  4.3317e-02,  4.3317e-02],
        [ 3.2059e-02,  8.7144e-02,  2.3688e-01,  6.4391e-01]])
```



PyTorch provides many other criteria, among which

- `torch.nn.MSELoss`
- `torch.nn.CrossEntropyLoss`
- `torch.nn.NLLLoss`
- `torch.nn.L1Loss`
- `torch.nn.NLLLoss2d`
- `torch.nn.MultiMarginLoss`