Deep learning

5.1. Cross-entropy loss

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ DE GENÈVE

The usual form of a classification training set is

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

We can train on such a data-set with a regression loss such as the MSE using a "one-hot vector" encoding: that converts labels into a tensor $z \in \mathbb{R}^{N \times C}$, with

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

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

$$N \begin{pmatrix} 2 \\ 1 \\ 1 \\ 3 \\ 2 \end{pmatrix} \xrightarrow{\text{One-hot encoding}} \underbrace{\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}}_{C}$$

This can be done with `F.one_hot`.

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".

Consider this example with correct class 1, and two outputs $\hat{y}$ and $\hat{y}'$.

$$\begin{array}{ccc} y & \hat{y} & \hat{y}' \\ \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 2 & -1 & -1 \end{pmatrix} & \begin{pmatrix} 0 & 1 & 1 \end{pmatrix} \end{array}$$

Both $\hat{y}$ and $\hat{y}'$ have a MSE of 1, even though the $\hat{y}$ leads to a perfect prediction, and the $\hat{y}'$ to a perfectly wrong one.

The criterion of choice for classification is the **cross-entropy**, which fixes these inconsistencies.

Notes

MSE is justified in an Euclidean space where the distance to a target location is consistent with the objective. But it does not really makes sense in a classification context, because the class values do not have any topological structure: we cannot say that class "dog" is closer to class "school bus" than to class "building".
In this simple three-class example, the real class is 1, corresponding to the one-hot vector y, and we consider two predictions $\hat{y}$ and $\hat{y}'$.
The MSE between y and $\hat{y}$ is

$$\frac{(2-1)^2 + (-1-0)^2 + (-1-0)^2}{3} = 1,$$

and the MSE between y and $\hat{y}'$ is

$$\frac{(0-1)^2 + (1-0)^2 + (1-0)^2}{3} = 1.$$

Hence in a MSE sense, $\hat{y}$ and $\hat{y}'$ are "as bad", although the prediction with argmax, i.e. the "winner takes all" decision rule, is class 1 for $\hat{y}$ and either class 2 or 3 for $\hat{y}'$.

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

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

from which

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

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

$$\mathbb{H}(p, q) = -\mathbb{E}_p\left[\log q\right] = -\sum_k p(k) \log q(k),$$

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 δ_{y_n} and the estimated $\hat{P}_w(Y = \cdot \mid X = x_n)$.

This is what `torch.nn.CrossEntropyLoss` computes.

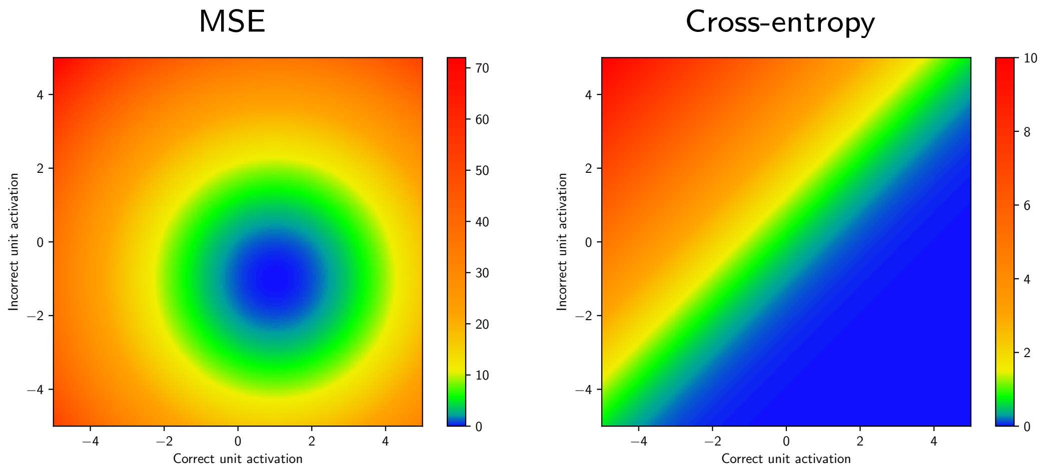
```
>>> f = torch.tensor([[-1., -3., 4.], [-3., 3., -1.]])
>>> target = torch.tensor([0, 1])
>>> criterion = torch.nn.CrossEntropyLoss()
>>> criterion(f, target)
tensor(2.5141)
```

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 “worse” than uniform.

Let's consider the loss for a single sample in a two-class problem, with a predictor with two output values.



$$\mathscr{L} = (x-1)^2 + (y+1)^2$$

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

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

Notes

We illustrate both the MSE loss and the cross-entropy loss in a 2d binary problem: the x axis represents the prediction for the correct class, and the y axis is for the incorrect class. The ideal case for MSE is to respond $+1$ for the correct class, and -1 for the incorrect one.
As soon as the prediction goes away from $(+1, -1)$, which is center of the blue area, the loss increases in an isotropic manner.
In particular, for a prediction of $(4, -4)$ at the bottom right, the loss is high, although the prediction is very good: strongly positive for the positive class, strongly negative for the negative class. And yet, such a prediction is "penalized" by the MSE loss, as much as e.g. $(-2, 2)$, which is a very bad prediction.
In contrast with the MSE loss, cross-entropy has a nice behavior. As soon as there is a sufficient difference between the response for the correct class and that for the wrong one, the loss is small.

The cross-entropy loss can be seen as the composition of a "log soft-max" to normalize the [logit] scores 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`.

Notes

In practice, if one wants to train a network so that its output are normalized logs of probabilities, then the last module of the network should be `nn.LogSoftmax`, in which case the loss used for training can be `nn.NLLLoss`.
If one just need to have logits, that is logs of non-normalized probabilities, then the network can be trained with `nn.CrossEntropyLoss`.

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`