

# 50.039 – Theory and Practice of Deep Learning

Alex

## Week 06: Loss balancing

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources. ]


### Key takeaways:

For unbalanced problems you can

- use sample weights in losses at testing and at training time
- use control over your dataloader to create mini-batches that are balanced
- see how an unweighted loss can be turned into a weighted loss

## 1 unbalanced problems - the importance of losses and dataloaders

Sometimes you have to deal with unbalanced problems, that is there is a subset of interest which has very small numbers.

**Scenario / Use case:** You want to predict  on some features which customer will buy a certain insurance. Unfortunately only 0.1% of all your customers will buy the insurance. Suppose you model it as a two-class problem (buy/not buy). By manager decision, your goal is to get as many of the potential customers as you can, even if you spam a few more people who won't consider to buy it.

If you consider zero-one loss (or below its accuracy), you will see, that it is not the right way for above problem. Why?

$$acc = 1 - \frac{1}{n} \sum_{i=1}^n 1[f(x_i) \neq y_i] = \frac{1}{n} \sum_{i=1}^n 1[f(x_i) = y_i]$$

You can resort to, for example:

- at test time: use a different, weighted loss for measuring
- at training time: put weights in your loss function for samples, to give higher weights to rare samples
- at training time: use your dataloader to upsample a rare class

## 1.1 Measuring final performance at test time

One alternative is in classification settings to use for example a weighted performance measure, for example a weighted sum of  $TPR$  and  $TNR$ .

$$\begin{aligned} ws(data) &= \alpha TPR + (1 - \alpha) TNR \\ &= \alpha \sum_{i:y_i=+1} \frac{1}{n_+} 1[f(x_i) = y_i] + (1 - \alpha) \sum_{i:y_i=-1} \frac{1}{n_-} 1[f(x_i) = y_i] \end{aligned}$$

This comes with a weight ratio, telling how important it is to get a positive right ( $\alpha$ ) versus a negative sample right ( $1 - \alpha$ ).

What can one use as weights if one has no idea on how to choose  $\alpha$ ?

Compare this to the vanilla accuracy

$$\begin{aligned} s &= \sum_{i=1}^n \frac{1}{n} 1[f(x_i) = y_i] \\ &= \sum_{i:y_i=+1} \frac{1}{n} 1[f(x_i) = y_i] + \sum_{i:y_i=-1} \frac{1}{n} 1[f(x_i) = y_i] \end{aligned}$$

Above means, that the zero one loss is a sum over positive and negative samples with weight always  $\frac{1}{n}$ .

Now look at a weighted loss:

$$\begin{aligned}\alpha TPR + (1 - \alpha) TNR &= \alpha \frac{1}{n_+} \sum_{i: y_i = +1} 1[f(x_i) = y_i] + (1 - \alpha) \frac{1}{n_-} \sum_{i: y_i = -1} 1[f(x_i) = y_i] \\ &= \frac{\alpha n}{n_+} \sum_{i: y_i = +1} \frac{1}{n} 1[f(x_i) = y_i] + \frac{(1 - \alpha)n}{n_-} \sum_{i: y_i = -1} \frac{1}{n} 1[f(x_i) = y_i] \\ ws &= \sum_{i: y_i = +1} \frac{1}{n} 1[f(x_i) = y_i] + \sum_{i: y_i = -1} \frac{1}{n} 1[f(x_i) = y_i]\end{aligned}$$

The weighted loss differs from zero-one loss by multiplicative factors of  $\frac{\alpha n}{n_+}$  and  $\frac{(1-\alpha)n}{n_-}$ .



They have an interpretation:  $\frac{\alpha n}{n_+} = \frac{\alpha}{(n_+/n)}$  is the ratio between  $\alpha$  and the fraction of positive samples ( $n_+/n$ ).



What happens to the balanced loss if  $\alpha = 0.5, n_+ = n_-$ ?

## 1.2 weighting losses at training time

Any loss function, which is a sum of samples can be appropriately weighted.

Example: the unweighted cross-entropy loss for two classes is:

$$\begin{aligned}p(x_i) &= P(Y = +1|x_i) = \frac{1}{1 + e^{-f(x_i)}} \\ L(\{(f(x_i), y_i), i = 1, \dots, n\}) &= \sum_{i=1}^n y_i \log p(x_i) + (1 - y_i) \log(1 - p(x_i))\end{aligned}$$



where  $f(x_i)$  is the output of the fullyconnected layer (not the softmax).

Design approach:

- This has a term  $l_i = y_i \log p(x_i) + (1 - y_i) \log(1 - p(x_i))$  per sample.

$$\begin{aligned}L(\{(f(x_i), y_i), i = 1, \dots, n\}) &= \sum_{i=1}^n l_i \\ l_i &= y_i \log p(x_i) + (1 - y_i) \log(1 - p(x_i))\end{aligned}$$

- We can weight the term per sample dependent on the label of the sample:

$$l_i \longrightarrow t(y_i)l_i$$

such that

$$t(y_i) = a \mathbb{1}[y_i = +1] + b \mathbb{1}[y_i = -1] = \begin{cases} a & \text{if } y_i = +1 \\ b & \text{if } y_i = -1 \end{cases}$$

- this works for all losses which decompose into one term per sample (IoU, hinge loss, ...)

Note the convenience of using logical indicator function  $\mathbb{1}[T]$  for a logical expression  $T$ . if-else can be slow in code.

If one wants to have equal weights for the smaller set of samples for the positive class as for larger set of samples of the negative class, then

$$\begin{aligned} a &= C \frac{1}{n_+} \\ b &= C \frac{1}{n_-} \end{aligned}$$



Note that such weights have an impact on the scale of the gradient, thus one may need to adapt the learning rate, as well. To see this consider the case that the whole loss is scaled up or down by some constant  $K$

$$\begin{aligned} T_1 &= \sum_i l_i \\ T_2 &= \sum_i K l_i \\ \Rightarrow \frac{\partial T_2}{\partial w} &= K \sum_i \frac{\partial l_i}{\partial w} = K \frac{\partial T_1}{\partial w} \end{aligned}$$

Class weights for equal weight of both classes:

if you original loss has the structure

$$\sum_i l_i(f, x_i, y_i)$$

Then you can use classweights

$$\begin{aligned} a &= C \frac{1}{n_+} \\ b &= C \frac{1}{n_-} \\ t(y_i) &= a \mathbb{1}[y_i = +1] + b \mathbb{1}[y_i = -1] \\ \sum_i t(y_i) l_i(f, x_i, y_i) \end{aligned}$$



### 1.3 upsampling dataloaders at training time


There is another option, if your class is very rare, say 0.0001% of all samples. Balancing losses will make training very unstable, bcs seeing a sample from your rare class is a rare event, and the order of elements seeing in the epoch can matter a lot!



The thought is the following: you have a positive sample one in every ten thousand, and run a batchsize of 32. How often are you going to see a positive sample?

**The idea here is:**

You want to have in your minibatch 50% positive and 50% negative sample (you can choose in practice another percentage, its just an exemplary number.)

- You can create a dataloader which maintains the subset of positive samples and the subset of negative samples. 
- additionally you maintain a pointer at which sample you are currently - for positive and for negative
- what to do in `def__getitem__(self,idx)` : so that 50% (or 30%) of your minibatch are positive samples? There is a probabilistic and a deterministic approach!



## 2 start in class

- consider `sampletr.txt` and `sampleste.txt` (v6 folder) for train and test data. every line is 2-dimensional input data with 1 label.
- plot the training data
- implement a neural net having only one fully connected layer with 2 inputs and one output.
- train always with 100 iterations, learning rate 0.1, batchsize 128, no weight decay, SGD
- now train with a standard dataloader and an standard loss, e.g. cross-entropy with logits or hingeloss
- now write a dataloader which ensures that every minibatch consists of 50% of the smaller class and 50% of the larger class, and which gives everytime a random order of samples.
- compare for both choices: accuracy, balanced accuracy ( $\alpha = 0.5$ ) in the weighted mean, and true positive rate

- if you are curious/not required: observe that you can find a good solution also with the standard loss when you reduce the batch size.