

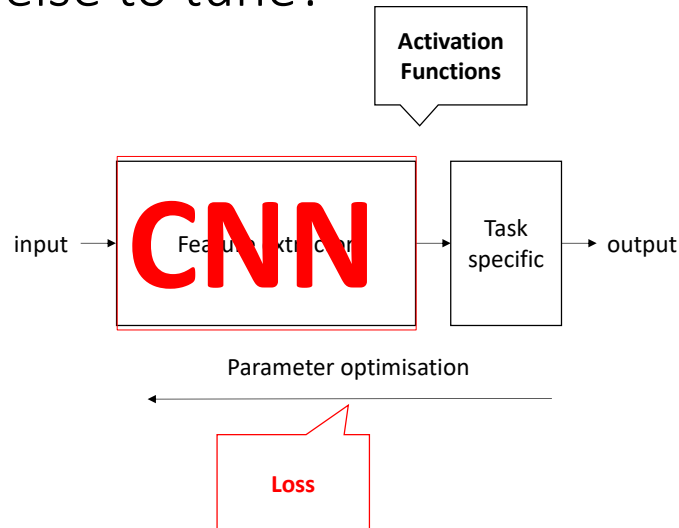
Deep Learning – Loss Functions

Bernhard Kainz

Deep Learning – Bernhard Kainz

Let's talk about a few miscellaneous but important topics.

Where else to tune?



Deep Learning – Bernhard Kainz

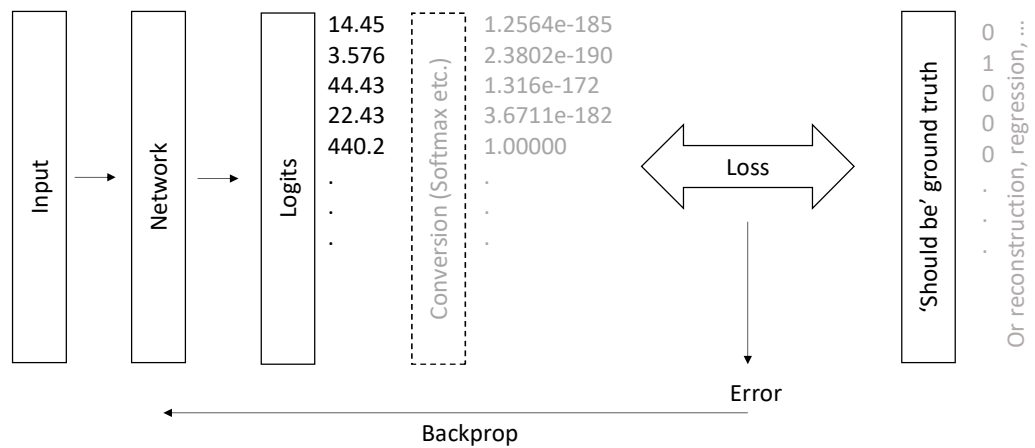
Except network depth one can also tweak two other areas.

One is how the neuronal in network activate given a certain number of inputs

The other one is how you define the Error or Loss that's backpropagated.

A lot of research has gone into these, probably a bit more and more recently into losses and a bit less and more during the early days into activation functions.

Loss functions



Deep Learning – Bernhard Kainz

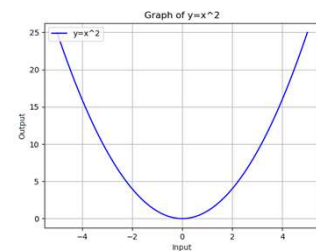
In case it is not clear what a loss or an error function does, it just compares the output of your system to the desired 'should be' output and calculates usually a single numerical value which characterises the difference.

L2 Norm, mean squared error

$$\ell(x, y) = \mathcal{L} = \{l_1, \dots, l_N\}^T, \quad l_n = (x_n - y_n)^2$$

Reduction to a single value can be either $\text{mean}(\mathcal{L})$ or $\text{sum}(\mathcal{L})$.

pytorch: `nn.MSELoss()`



Deep Learning – Bernhard Kainz

The mean square error is probably straight forward. You take the difference of the result and the ground truth for this sample and square it.

If you have a minibatch you get of course a whole list of these.

They can be reduced to a single value by either the sum or the mean.

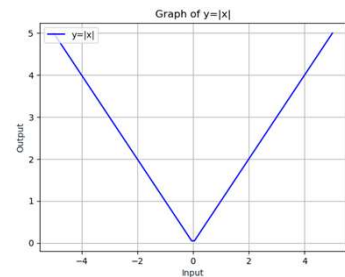
L1 Norm

$$\ell(x, y) = \mathcal{L} = \{l_1, \dots, l_N\}^T, \quad l_n = |x_n - y_n|$$

Reduction to a single value can be either $\text{mean}(\mathcal{L})$ or $\text{sum}(\mathcal{L})$.

Use for robust regression (noisy data)

pytorch: `nn.L1Loss()`



Deep Learning – Bernhard Kainz

The L1 loss is basically the Absolut value of the difference between the current sample's actual output and the desired output.

This is used for robust regression. This means that you want small errors to count a lot and large errors to count but not as much as for example for a L2 norm.

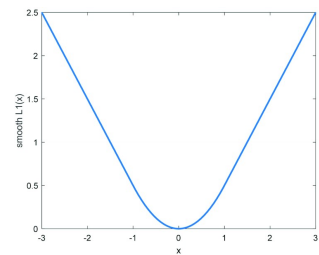
This is good when the training data is noisy.

The problem with this is that it is not differentiable at the bottom. To mitigate this you could use a soft version called SoftShrink or SmoothL1Loss.

Smooth L1

$$\text{loss}(x, y) = \frac{1}{n} \sum_i z_i$$
$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$$

pytorch: `nn.SmoothL1Loss()`



Deep Learning – Bernhard Kainz

This one is behaving like an L2 loss at the bottom and like an L1 norm for large errors.

This is again useful for protecting against outliers.

One problem with this loss is that it has a scale, e.g. 0.5 in this case. This depends on the scale of your errors, which is difficult to know in advance.

Negative log likelihood loss

- Assumption: Network output represents log likelihoods.
- Make the desired output as large as possible and all others as small as possible

$$\begin{aligned}\ell(x, y) &= \mathcal{L} = \{l_1, \dots, l_N\}^T, & l_n &= -w_{y_n} x_{n, y_n}, \\ w_c &= \text{weight}[c] \cdot 1\{c \neq \text{ignore_index}\}\end{aligned}$$

$$\ell(x, y) = \begin{cases} \sum_{n=1}^N \frac{1}{\sum_{n=1}^N w_{y_n}} l_n, & \text{if reduction = mean} \\ \sum_{n=1}^N l_n, & \text{if reduction = sum} \end{cases}$$

pytorch: `nn.NLLLoss()`

Implementation valid for all such problems, not only likelihoods.

Deep Learning – Bernhard Kainz

The Negative log likelihood loss is based on the idea that every output represents a likelihood for example a particular class.

It aims to make the output for the correct class as high as possible and those for the others as small as possible.

Nothing restricts this here to likelihoods, so really the name is just there to convey the idea.

The equation says: pick the x that happens to be the correct one for one sample in the batch and make that score as large as possible.

This one here also allows you to give a different weight to different classes, can be important if you expect a significant class imbalance between the samples in your training data. For example if a few categories are rare.

The alternative is to increase the frequency of the class that is rare during training.

Cross Entropy (CE) Loss

- Combines LogSoftmax and NLLLoss
- Useful for classification problems with C classes

$$\text{loss}(x, \text{class}) = -\log\left(\frac{e^{x[\text{class}]}}{\sum_j e^{x[j]}}\right) = -x[\text{class}] + \log\left(\sum_j e^{x[j]}\right)$$

Classes can also be weighted.

The losses are averaged across observations for each minibatch.

`pytorch: nn.CrossEntropyLoss()`

Deep Learning – Bernhard Kainz

The cross entropy loss is very popular for classification problems. It combines Logsoftmax and negative log likelihood loss.

Basically it says: I have several weighted sums, I pass them through Softmax, then take the log of those, and then I want to make the output for the Logsoftmax as large as possible for the correct class. When you backpropagate through the logSoftmax, it makes as a consequence the scores of all other classes as small as possible.

If you simplify the combination, you just get the negative score of the correct class and then add a log of a sum of the exponentials of the scores of all the other classes to make this small.

Binary Cross Entropy (BCE) Loss

- CE loss for only two classes

$$\begin{aligned}\ell(x, y) = \mathcal{L} &= \{l_1, \dots, l_N\}^T, \\ l_n &= -w_n[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)]\end{aligned}$$

Reduction to a single value can be either $\text{mean}(\mathcal{L})$ or $\text{sum}(\mathcal{L})$.

pytorch: `nn.BCELoss()`

Requires [0,1] probabilities. If this cannot be guaranteed, use

`nn.BCEWithLogitsLoss()`

Deep Learning – Bernhard Kainz

This does not contain logsoftmax, it is just entropies and models a Bernoulli distribution.

Probabilities have to be between 0 and 1

This loss is a special case of cross entropy for when you have only two classes so it can be reduced to a simpler function. This is used for measuring the error of a reconstruction in, for example, an auto-encoder. This formula assumes x and y are probabilities, so they are strictly between 0 and 1.

Kullback-Leibler Divergence Loss

- Measures distance between distributions

$$\ell(x, y) = \mathcal{L} = \{l_1, \dots, l_N\}^T, \quad l_n = y_n \cdot (\log y_n - x_n)$$

Pytorch: `nn.KLDivLoss()`

Deep Learning – Bernhard Kainz

This is simple loss function for when your target is a one-hot distribution (i.e. y is a category). Again it assumes x and y are probabilities. It has the disadvantage that it is not merged with a softmax or log-softmax so it may have numerical stability issues.

This is a simplified version if you have a one-hot representation for the target. It is prone to numerical issues but very useful for example in variational auto encoders.

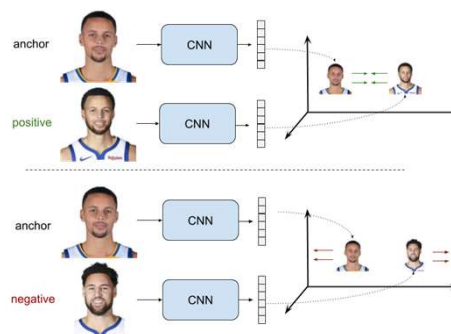
Margin Ranking Loss/Ranking Losses/Contrastive loss

$$\text{loss}(x, y) = \max(0, -y \cdot (x_1 - x_2) + \text{margin})$$

Useful to push classes as far away as possible and for metric learning

Practical: take category that scores is closest or higher than correct one
change until difference is at least the margin

pytorch: `nn.MarginRankingLoss()`



Deep Learning – Bernhard Kainz

<https://gombru.github.io/>

Margin ranking losses are an important category of losses.

Unlike other loss functions, such as Cross-Entropy Loss or Mean Square Error Loss, whose objective is to learn to predict directly a label, a value, or a set or values given an input, the objective of Ranking Losses is to predict relative distances between inputs. This task is often called metric learning.

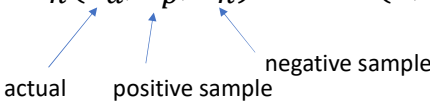
Ranking Losses functions are very flexible in terms of training data: We just need a similarity score between data points to use them. That score can be binary (similar / dissimilar).

To use a Ranking Loss function we first extract features from two (or three) input data points and get an embedded representation for each of them. Then, we define a metric function to measure the similarity between those representations, for instance Euclidian distance. Finally, we train the feature extractors to produce similar representations for both inputs, in case the inputs are similar, or distant representations for the two inputs, in case they are dissimilar.

We don't even care about the values of the representations, only about the distances between them. However, this training methodology has demonstrated to produce powerful representations for different tasks.

So Margin losses are an important category of losses. If you have two inputs, this loss function says you want one input to be larger than the other one by at least a margin. You want the score for the correct category larger than the score for the incorrect categories by at least some margin. $y*(x_1 - x_2)$ is larger than margin, the cost is 0. If it is smaller, the cost increases linearly. If you were to use this for classification, you would have x_1 the score of the correct answer and x_2 the score of the highest scoring incorrect answer in the mini-batch. If used in energy based models (discussed later), this loss function pushes down on the correct answer x_1 and up on the incorrect answer x_2 .

Triplet Margin Loss

- $\ell(x, y) = \mathcal{L} = \{l_1, \dots, l_N\}^T$,
$$l_n(x_a, x_p, x_n) = \max(0, m + |f(x_a) - f(x_p)| - |f(x_a) - f(x_n)|)$$


Make samples from same classes close and different classes far away.

Objective: Distance for the good pair has to be smaller than distance to the bad pair. Actual distance does not need to be small, just smaller.

Used for metric learning and Siamese networks

pytorch: `nn.TripletMarginLoss()`

Deep Learning – Bernhard Kainz

When we use pairs of training data points or triplets of training data points.

This loss is used for measuring a relative similarity between samples. This is used a lot for metric learning.

For example, you put two images with the same category through a CNN and get two vectors.

You want the distance between those two vectors to be as small as possible.

If you put two images with different categories through a CNN, you want the distance between those vectors to be as large as possible.

This loss function tries to send the first distance toward 0 and the second distance larger than some margin.

However, the only thing that matter is that the distance between the good pair is smaller than the distance between the bad pair.

This was originally used to train an image search system for Google.

At that time, you would type a query into Google and it would encode that query into a vector.

It would then compare that vector to a bunch of vectors from images that were previously indexed.

Google would then retrieve the images that were the closest to your vector.

Triplet Margin Loss

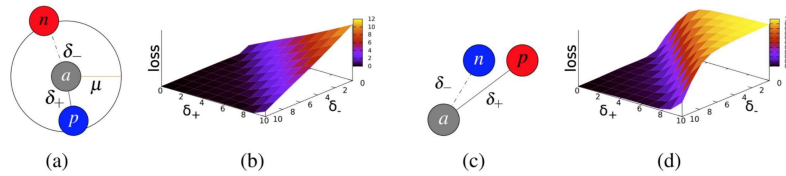


Figure 1: (a) Margin ranking loss. It seeks to push n outside the circle defined by the margin μ , and pull p inside. (b) Margin ranking loss values in function of δ_- , δ_+ (c) Ratio loss. It seeks to force δ_+ to be much smaller than δ_- . (d) Ratio loss values in function of δ_- , δ_+

$$\hat{\delta}_+ = \|f(a) - f(p)\|_2 \text{ and } \hat{\delta}_- = \|\hat{f}(a) - f(n)\|_2.$$

$$\lambda(\delta_+, \hat{\delta}_*) = \max(0, \mu + \delta_+ - \hat{\delta}_*).$$

<http://www.bmva.org/bmvc/2016/papers/paper119/index.html>

Deep Learning – Bernhard Kainz

In the original paper they had a nice figure illustrating this behaviour. Positive examples are pulled closer and negative examples pushed further away.

Cosine Embedding Loss

$$\text{loss}(x, y) = \begin{cases} 1 - \cos(x_1, x_2), & \text{if } y = 1 \\ \max(0, \cos(x_1, x_2) - \text{margin}), & \text{if } y = -1 \end{cases}$$

Measure whether two inputs are similar or dissimilar

Basically a normalised Euclidean distance

pytorch: `nn.CosineEmbeddingLoss()`

Deep Learning – Bernhard Kainz

This loss is used for measuring whether two inputs are similar or dissimilar, using the cosine distance, and is typically used for learning nonlinear embeddings or semi-supervised learning.

Thought of another way, 1 minus the cosine of the angle between the two vectors is basically the normalised Euclidean distance.

The advantage of this is that whenever you have two vectors and you want to make their distance as large as possible, it is very easy to make the network achieve this by making the vectors very long. Of course this is not optimal. You don't want the system to make the vectors big but rotate vectors in the right direction so you normalise the vectors and calculate the normalised Euclidean distance.

For positive cases, this loss tries to make the vectors as aligned as possible. For negative pairs, this loss tries to make the cosine smaller than a particular margin. The margin here should be some small positive value.

In a high dimensional space, there is a lot of area near the equator of the sphere. After normalisation, all your points are now normalised on the sphere. What you want is samples that are semantically similar to you to be close. The samples that are dissimilar should be orthogonal. You don't want them to be opposite each other because there is only one point at the opposite pole. Rather, on the equator, there is a very large amount of space so you want to make the margin some small positive value so you can take advantage of all this area.

What do we learn from this

- The choice of loss depends on the desired output (e.g., classification vs. regression)
- Loss functions are a hot topic of research.
- It informs how the overall system behaves during training
- Don't get scared by the equations. If you look closely the underlying ideas are very simple.