Little insights on learning

- As we know, SGD is not guaranteed to find a local minimum if we dont have a convex function, so someone might find the result of these techniques a bit aleatoric, however, they are not: in fact they are *surprisingly stable* (the reason for this is not 100% understood as of now). However, 3 possible explanation for this can be related to:
  1. The *initializations* of the weights: usually, they are very close to 0 and chosen in a randomic way (essentially, it's a confined area of values).
  2. The *convexity* of the loss function: it's not rare to have loss functions which have multiple local minimum with values very much close to the global minimum.

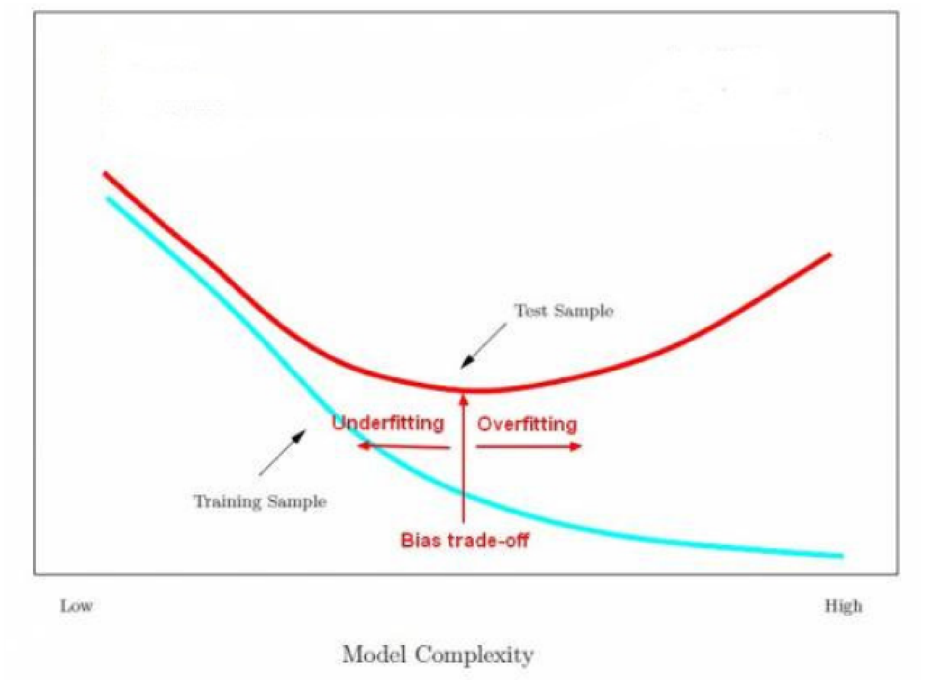
  3. (not proven though) *Redundancy* in the internal neural network: redundancy of the number of neurons and parameters inside the NN.
     - A lot of redundancy allows a NN to ignore the parts which give a bad result.
- In dense layers, in which each neuron is connected to every neuron of the previous layer, the order of the nodes (and thus the weights) is not relevant.

Overfitting and underfitting

As we know: - **overfitting**: the model is too *complex* and *specialized* over the *peculiarities* of the samples in the training set. - **underfitting**: the model is too *simple* and does not allow to express the complexity of the observations.

Overfitting essentially tells us to keep in mind that the training data we have is just a small subset of the bigger, more general set of real data.

> [!remark] Deep models are good at fitting, but the real goal is generalization - With NN models, we see mainly overfitting problems.



Ways to reduce overfitting

- **Collect more data:**
  - the more the test data, the most likely the predictions will be similar to real data -> the more likely you can capture the variability of the real world.
- **Reduce the model capacity**
  - Capacity refers to the ability of a model to fit a variety of functions; more capacity, means that a model can fit more types of functions for mapping inputs to outputs. Increasing the capacity of a model is easily achieved by *changing the structure of the model*, such as adding more layers and/or more nodes to layers.
- **Early stopping**:
  - If the results (i.e. validaiton of the training set) are *not improving, I simply stop.*
    - Easily implemented using callbacks. You can use callbacks to save weights or compare result at specific times of the run-times (i.e. the end of this epoch)
- **Regularization**, e.g. Weight-decay
  - l_2 regularization: you penalize the weights of the model using a quadratic penalty, to keep them close to 0.

  - Weight-decay: the weights decay towards 0.

  - We do this to improve smoothness of the model.
  - In each layer you add regularizers, and you can choose the kind of regularizers that you want.
- **Model averaging**
  - i.e. in Random forests, we compute an average of the results obtained from each tree.
- **Data augmentation**
  - We add data that we presume that is consistent with the real data (i.e. images with modifications, flipped images).
  - Can have dangerous repercussions.
- **Dropout**

Dropout

Idea: "*cripple*" the neural network stocastically removing hidden units (we disconnect neurons from the network, disabling them). - during training, at each iteration (not each epoch) *hidden units are disabled* with probability p (e.g. 0.5) - hidden units cannot co-adapts with other units (they are disconnected). - inputs not set to 0 are scaled up by $1/(1-p)$ - similar to train many networks and averaging between them.

This operation makes our network much *more robust*, since the network has to adapt to this strong change. Another way to make the model more robust is to inject noise into the data, and check the results with very noisy data (i.e. adding salt and pepper noise).

Activation and loss functions for classification

In this topic we'll talk about activation functions on the *final layer* of the NN. We know that on the hidden layers we should always use *rectified linear units* (or variants of it).

Sigmoid

When the result of the network is a value between 0 and 1, e.g. a probability for a binary classification problem, it is customary to use the *sigmoid function*:

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

as activation function. If

$$P(Y=1|x) = \sigma(f(x)) = \frac{e^{f(x)}}{1+e^{f(x)}}$$

then

$$P(Y=0|x) = 1-\sigma(f(x)) = \frac{1}{1+e^{f(x)}}$$

In general, sigmoid functions are used when we have to define a probability of a certain choice, in particular they are used when our network has a *single output*.

Softmax

[useless stuff he said] When using the softmax activation function, we are trying to generalize the situation in which we have a multiclass classification problem, in which for each class we want to output the probability that input belongs to each class. So, the output is the probability distribution of the input wrt to all the categories of the output that we are interested in.

In this case, since we are outputting a probability distribution over the categories, our output must satisfy 2 constraints: 1. The output for each class should be between 0 and 1. 2. The sum of the outputs should be 1. [end of useless stuff he said]

When the result of the network is a *probability distribution*, e.g. over K different categories, the **softmax function** is used as activation:

$$\text{softmax}(j, x_1, ..., x_k) = \frac{e^{x_j}}{\sum_{i=1}^{k} e^{x_i}}$$

It is easy to see that

$$0 < \text{softmax}(j, x_1, ..., x_k) < 1$$

and most importantly

$$\sum_{j=1}^{k} \text{softmax}(j, x_1, ..., x_k)$$

Softmax vs Sigmoid

An interesting property of the softmax function is that it is *invariant for translation*: if we add a value c to the inputs, the output *remains unchanged*.

$$\text{softmax}(j, x_1, ..., x_k) = \text{softmax}(j, x_1 + c, ..., x_k + c)$$

We could see the softmax function as a *generalization* of the sigmoid case:

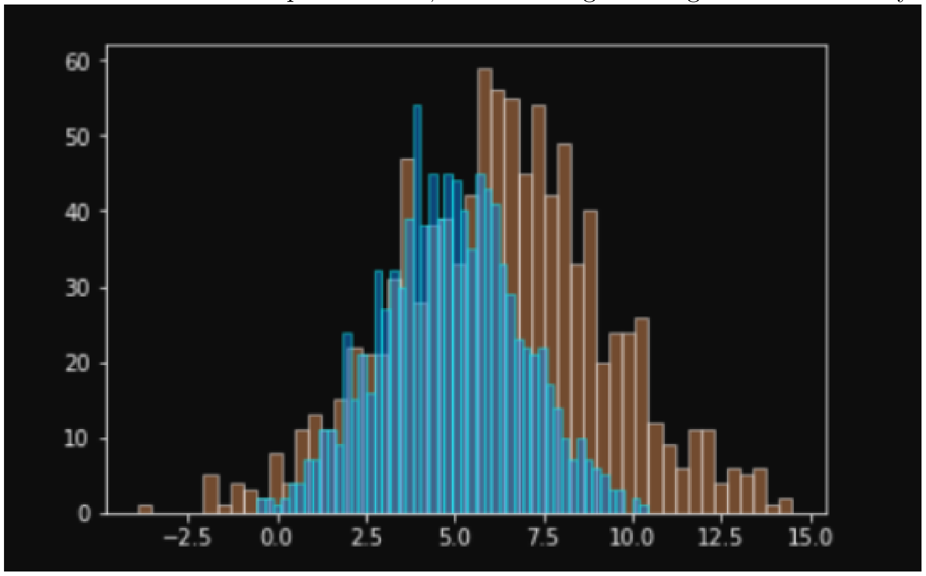
$$\sigma(x) = \text{softmax}(x, 0)$$

Cross entropy

Comparing loss functions

The output computed, usually, is a probability distribution. - Also, the ground truth can be seen as a *categorical distribution*, since we have a category for each one of the inputs. So, if an input belongs to one of the categories, its probability wrt to that category is 1, while the probability wrt to the others is 0.

As we know, the **loss function** is the *difference* between the *actual output* of the model and the *ground truth.* The problem is, what loss functions should we use for *comparing probability distributions*? - We could treat them as "normal functions", and use e.g. *quadratic distance* between true and predicted probabilities. - Can we do better? For instance, in logistic regression we do not use mean squared error, but use negative log-likelihood. Why?



Probability distributions can be compared according to many different metrics. There are two main techniques: - you consider their *difference* $P - Q$ (e.g. Wasserstein distance, it tries to measure the amount of "work" needed to reshape the curve to the one of the ground truth) - you consider their *ratio* P/Q (e.g. **Kullback Leibler divergence**)

Kullback-Leibler divergence

The **Kullback-Leibler divergence** $DKL(P||Q)$ between two distributions Q and P, is a *measure of the information loss due to approximating* P *with* Q.

$$
\begin{aligned}
DKL(P\|Q) &= \sum_i P(i) log \frac{P(i)}{Q(i)} \\
&= \sum_i P(i)(logP(i) - logQ(i)) \\
&= -\underbrace{\mathcal{H}(P)}_{entropy} - \sum_i P(i) logQ(i)
\end{aligned}
$$

We call **Cross-Entropy** between P and Q the quantity:

$$\mathcal{H}(P, Q) = -\sum_i P(i) logQ(i) = \mathcal{H}(P$$

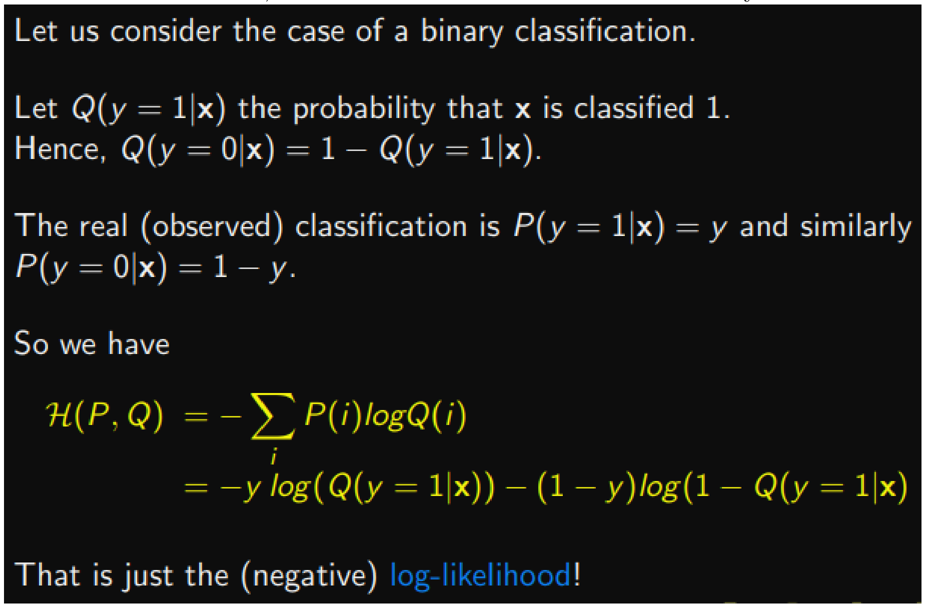
Since, given the training data, their entropy $H(P)$ is constant, minimizing $DKL(P||Q)$ is equivalent to minimizing the cross-entropy $H(P, Q)$ between P and Q.

Some properties: - The cross entropy is minimal when Q *is equal to* P. - The KL divergence is always *positive*, and it is equal to 0 when Q is equal to P.

A learning objective can be the *minimization* fo the Kullback-Leiber divergence.

Cross-Entropy and Log-likelihood

To better understand, let us consider the case of the binary classification.



Let us consider the case of a binary classification.

Let $Q(y=1|\mathbf{x})$ the probability that $\mathbf{x}$ is classified 1.
Hence, $Q(y=0|\mathbf{x}) = 1 - Q(y=1|\mathbf{x})$.

The real (observed) classification is $P(y=1|\mathbf{x}) = y$ and similarly $P(y=0|\mathbf{x}) = 1-y$.

So we have

$$\begin{aligned}\mathcal{H}(P,Q) &= -\sum_i P(i) log Q(i) \\ &= -y\, log(Q(y=1|\mathbf{x})) - (1-y) log(1 - Q(y=1|\mathbf{x})\end{aligned}$$

That is just the (negative) log-likelihood!

If $x = 1$, then the probability of $P(y=1|x)$ is 1.

Calculating the cross entropy is the same as computing the log-likelihood of Q w.r.t. to the real distribution P.

Log-likelihood Predicted log-likelyhood that X has label Y is $logQ(Y|X)$. We want to split it according to the possibile labels l of Y:

$$logQ(l_1|X) + logQ(l_2|X)...logQ(l_n|X)$$

but weighted in which way?

According to the actual probability that X has label l:

$$P(l_1|X)logQ(l_1|X) + P(l_2|X)logQ(l_2|X)...P(l_n|X)logQ(l_n|X)$$

or

$$\sum_l P(l|X)log(Q(l|X)$$

Cross-entropy is really how likely is the probability of a probability distribution Q given P. So essentially, we want to measure how likely is the distribution of the prediction w.r.t. to the training data that we have.

Summing up... For binary classification use: - sigmoid as activation function - binary crossentropy (aka log-likelihod) as loss function

For multinomial classification use: - softmax as activation function - categorical crossentropy as loss function

Understading Log-likelihood

(this part was in the next lesson)

$< x_i, y_i >$, with categories $\{k_1, ..., k_n\}$ Let's assume that we have a binary distribution, so we have k_1 and k_2. We have to estimate the probability of x given that y is y_i, (that is, the value Q). We also have a *ground truth probability* P, i.e. $P(y = k|x = x_i)$. - We could have that $k = y_i$, and in this case P = 1. Otherwise $k \neq y_i$, and P = 0.

We assume that all the data that is in the training set are *independent from each other*, and so their probability is just the *product of the probabilities*. The likelihood, then, is: $L = \prod_{i \in \text{DATA}} L_i = \prod_{i \in \text{DATA}} Q(y = y_i|x = x_i)$

We can prove that ==[1]==: $Q(y = y_i|x = x_i) = Q(y = k_1|x = x_i)^{P(y=k_1|x=x_i)} \cdot Q(y = k_2|x = x_i)^{P(y=k_2|x=x_i)}$

Let's examine the case in which $y_i = k_1$: - In this case, $P = 1$ in the first part and $P = 0$ in the second term of the multiplication. So, the final result is $Q(y = y_i|x = x_i) = Q(y = k_1|x = x_i)$ Same thing if $y_i = k_2$.

Let's compute the whole logarithm of the expression ==[1]==. We'll get:

$$P(y = k_1|x = x_i) \cdot log(Q(y = k_1|x = x_i)) + P(y = k_2|x = x_i) \cdot log(Q(y = k_2|x = x_i))$$

It's basically just the *sum of the possibility*, for all the categories that we have, that the label of associated with x is y_i. And this is also the definition of the cross-entropy between P and Q.