Normalization and Regularization Technics for Neural Nets

(Last update 9-MAY-2020, Konstantin Burlachenko, burlachenkok@gmail.com)

Terminology

Term	Meaning
One epoch	Train Data is divide into Batches (Optimization community term) or Mini-Batches (DL community term). Once you process all data (represented by sequence of batches) people say that one epoch have been processed.
One iteration	One batch during training have been processed.

About mathematical problem and all problems in the setup

About our setup

What we're finding in ideal case is $F^* = argmin_F\left(E_{xy\sim D}L(y, F(x))\right)$

List of problems:

1. We don't know distribution *D.* Instead of distribution we have only finite samples.
2. Even non-stochastic Math Optimization problem can be extremely hard to solve.
3. Typically people consider parametrized version, where function is pulled from some specific function class. But do this you should setup *Function Class*.
4. From such setup we can predict maybe good $E[Y|X]$, but has it any connection with mapping $F(x)\rightarrow y$ at all? We hope that p.d.f. of r.v. $Y|X$ is rather picky with small variance. So $E[Y|X]$ can be used as a predictor, **maybe**.
5. What is constructed in some function $X\rightarrow Y$ even real dependency has form: $X, Z\rightarrow Y$, where Z are unobserved variables.
6. We assume that most impact into **Y** is due to **X** variables
7. We assume that distribution of **Z** will not change
8. We assume that distribution **D** will not change in future
9. We assume that variable **Z** introduce additive noise
10. We assume distribution of **Z** is not depend in *X*.

Even beside that – we should understand what we build is only approximations:

1. It's not real phenomena driven from some fundamental law
2. It's just some kind of clever approximation, no more. (But with interesting built-in tricks)
3. Even with this setup there are a lot of interesting math.

We assume that optimization setup $F^* = argmin_{F \in F}\left(E_{xy \sim D} L(y, F(x))\right)$ will construct good approximation for us.

In fact if we know all quantity from right hand side, then $F^*: X \to Y$ is called **optimal** or **target function**.

About our goal

Our goal after select function space F select one function from it. And after train said that it's our predictor model. For function selection we use scalar value function of argument which is a function.

Nice term for this in Functional Analysis is "*Functional*".

Score Function judges how good a fitted model to the data is.

Score Function in population in risk setting: $S(F) = E_{xy \sim D} L(y, F(x))$

Score Function in data in empirical risk setting: $S(F) = \frac{1}{N} \sum_{i=1}^{N} L\left(y_i, F\left(x_i\right)\right) + \lambda \cdot P(F)$

Really we will have a deal in practical setting with **Score Function in data.**

About $\lambda \cdot P(F)$ term

In fact only one powerful way to find "good presentation" is beside data, which is laughable small in all powerful schemas used in Machine Learning) is incorporate **prior knowledge**:

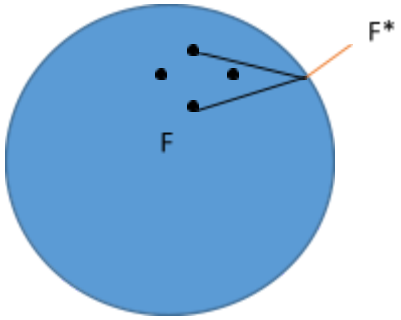
1. Choosing approximation method (Feed Forward NN, CNN, Decision Trees, NN) implicitly do it.
2. Choosing regularization schemas also implicitly do it
3. Beside Model selection it's also important how model is robust if this assumption are not so correct.

Problems with Train Data

1. High-dimensional space impossible to sample densely
2. Even if use Kolmogorov-Arnold-Theorem and transfer high-dimensional mapping into scalar function of single variable form of this simple scalar functions can be wild enough.

Bias and variance from statistical learning point of view

Scalar quantity $R(F) = E_{xy} L(y, F(x))$ **is** called Risk **or** Lack Of Accuracy**.**



1. "F" is function family which we think is good to be used as a set of possible candidate for F*
2. "F*" is target function we don't know it.
3. This function space can be considered as something like (but I think not exactly as) *Metric Space.*
4. Distance between points in this space(functions) defined as distance between two scalar values which is prediction risk on population
5. If distance is zero between functions – then let's say we do not distinguish between functions and say they are *equivalent*.
6. I don't know how this is handle, because formally metric function should be equal to 0 **iff** you measure distance between two same points in *Metric Space*. Maybe it's more precisely to think about functions from which we selecting divided into Equivalent classes. Where function from one class has the same *risk*.
7. "F*" is projected "via red line" into set **F.** Length of this red segment is **representation bias**. This is a real error – you don't know F*.
8. Black points in F is evaluated solutions by **empirical risk minimization** on finite data.
9. Because we have random samples from population we will formulate different optimization problems each time (in picture above we solve Optimization Problem 4 times) with generally speaking different optimal point for each random train sample.
10. Distance between various functions in F(black point) and projection F* on F is instance variance, i.e. it's length of black segment

11. Average of different *instance variances* is called **variance**.

Reason of Variance and Bias error

Reason of Variance. We do not know population and we use only data. Variance can also "be corrected" by more amount of data. ***It is about uncertainty which function is the best in case of limited amount of data.*** So in case of all consider just all possible functions $X \to Y$ will have no bias at all, but extremely big *variance*.

Reason of bias Is that our function class not necessary contain target function. Bias can be fixed by consider more big function space.

Complete picture about supervised models

Sequence of Function Family

Complete picture is that we consider not only one *Function Family* – in fact all strong methods consider growing series sets of nested function families and try to pick the best function via cross-validation.

So it's worthwhile to consider several optimization problems in each function class and then based on some principle select the best one. For example both Neural Nets and Decision Trees allow define function class, but in the same time they allow vary size of this function space via varying meta-parameters.

We have several function subspaces $S \subset X \to Y$. What function subspace is more better to use?

Any change in *Target Function, Sample Size, Noise* completely can screw the picture what is best.

Principle which underpins *Machine Learning* we select function which will behave best in the future.

There is only one problem – we don't have a future.

But **Great Trick** is create this future artificially.

And We create artificial future which is **not train set** but some **validation set**, but under assumption that distribution will not change in future we estimate how good we're via **validation set**.

Sequence of opt. problems as a regularization path

Just for comfortable all discussion below assume that we collapsed all meta-parameters (in Homeomorphic style, i.e. in one-to-one mapping) into one single scalar parameter λ.

Also you have a deterministic train set and deterministic optimization algorithm.

You specify scalar parameter λ.

And find solution via empirical risk minimization and write it as a function $w(\lambda)$.

You change slightly scalar parameter $\lambda + d\lambda$. Find solution via empirical risk minimization $w(\lambda + d\lambda)$

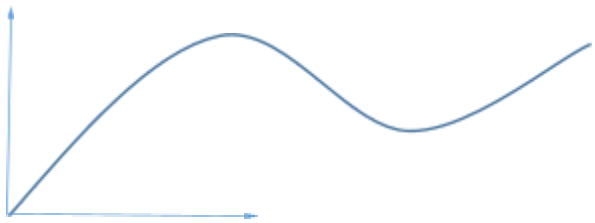
Now for each point in the path you have *parameters(weights)* for approximated function.

$w(\lambda)$ is one dimensional path in high-dimensional space of all parameters of your model.

General principle to select function in regularization path

How *functions* corresponded for each λ_i behaves in principle we can check via cross-validation. i.e. in fact just test behavior of this function in test set which we hope give us good surrogate for the future.

This one in fact is true objective.



Regularization Technics for Neural Nets

You fixed topology, so you have a fixed NN architecture.

But still, for various reasons you should not fit perfectly on the data.

First reasons you real goal is solve stochastic opt. problem, not deterministic.

Second reasons of the reason in data there is information about noise.

The only one case when you should fit perfectly – if data has no noise at all. In other cases this action is not necessary perfectly good.

Regularization for Neural Nets or any procedure is pull solution from the data into **something else**.

If this something else is targeted to ***target function*** then it will reduce variance, but to be honest it can pull into another direction.

Penalty function in objective

$min. S(F)$

$$S(F) = \frac{1}{N}\sum_{i=1}^{N} L\left(y_i, F\left(x_i\right)\right) + \lambda \cdot P(F)$$

P – is functional, but because NN is parameterized class it can be considered. Examples: *L2* norm square of weights(or parameters of NN), or Ridge Penalty.

Early Stopping. Description.

Idea – stop sequential training before solution will be obtained.

Data is divided into **Train Set** and **Validation Set**. If we start increase empirical loss in validation – **we stopped**. We assume that after validation error starts increasing will start go far from *target function*.

It's only assumption no more. There are no guarantees in form of *regularization path*.

Early Stopping. Why this have sense.

During training NN you follow some one dimensional path $w(t)$. In each moment of time in this path you have specific model. Error in **Validation Set** is you proxy of distance to target function. First of all our particular construction of this path does not say a lot. For example no guarantees that distance to **target function** will be attained in the end of the path.

Early stopping in fact can be formalized as Penalty to number of steps. But even to show it can be hard. Prove maybe can be found in some papers of Jerome H.Friedman.

Add random noise to input

Just add random noise to input from the train set.

Drop-out regularization. Description.

During learning, forward and backward propagation some part (think about half) of the network nodes **outputs** are forced to be zero. It's the same as this nodes are completely removed in this iteration.

The schema is to drop some cell in Neural Nets with some probability "p" in the whole computation graph.

One popular technic how to handle is **Inverted Dropout**.

Drop-out regularization. Why it can have sense.

Idea behind – train some subset of complete Neural Network in such way that it will force learning procedure in such way that it can not more rely on any specific feature.

Only one thing. Once you will apply this technic scalar score function is not necessary decrease in every step of stochastic gradient descend.

Algorithm behavior in Train Time

d is random matrix with {0,1} values with **1** value in (i,j) position with probability ***keepProbability***

(which is a scalar like 0.8)

$a^{[l,new]} := \frac{a^{[l]} * d}{keepProbability}$ (here "*" is elementwise matrix multiplication)

Particularly this division means the following $E\left[a^{[l,new]}\right] = E\left[\frac{a^{[l]} * d}{keepProbability}\right] = a^{(l)}\frac{E[d]}{keepProbability} = a^{[l]}$

Test Time

The most popular strategy in test time – **do nothing**. As alternative you can run inference several times with different **drop-out** cells and average result. But empirical study provides intuition is that it gives approximately the same result.

Data Augmentation

One typical thing that if you know method how from valid input data generate new valid input train data – you can do it. If from valid data you will generate new valid data it's a good thing always - it will always reduce variance. Typical approaches when input is image are *mirroring, image cropping, image rotation, some kind of slightly image distortion*.

All typical schemas of regularizations technics

Assume that architecture of computation graph have been fixed.

After this step *space of parameters* is fixed and form of X; $w \to Y$ is fixed too.

We use optimization strategy to "optimize" model.

We modify objective into some other form $w = argmin_w E_{x,y\sim D} f(x, y; w, \lambda)$ **for fixed** λ

During execution of algorithm we follow path $w(\lambda)$ and we produce sequence of parameters for such parameterized class of functions.

In fact **<u>any modification</u>** in generating path trajectory can be considered as **<u>regularization</u>**:

1. You can append some another term into objective
2. You can modify optimization algorithm and make it more smooth
3. You can append some kind of constraint set
4. **What is fun that - any noise in fact DL community can consider as regularization**

Some scientists in the field for example *Andrew Ng* like in fact another situation, when there is a decoupling and orthogonalization between two things:

- ***Optimization Technic*** – which solves or approximately solves setup problem
- ***Regularization Technic*** – which purpose to reduce variance in some way.

Normalization Technics for Neural Nets

Input Normalization

$\mu = \frac{1}{m}\sum_i X^{(i)}$ (Evaluate in train set)

$\sigma^2 = \frac{1}{m}\sum_i \left(X^{(i)} - \mu\right)^2$ (Evaluate in train set)

Apply scaling as pre-processing for **input** and **test** step. At this moment μ and σ^2 are just some constant.

$X^{(i)}_{norm} = \frac{X^{(i)}-\mu}{\sqrt{\sigma^2+e}}$ (Apply both for train and test set)

Level curves of objective after this preprocessing will be more roundish and for first-order optimization methods it's pretty important.

Batch Normalization. Description for training time.

Algorithm in Learning Time:

BN do normalization in the following way.

1. Evaluate elementwise mean and variance of input to activation functions Z for all units in some hidden layer across Batch.

$\mu = \frac{1}{m}\sum_i z^{(i)}$

$\sigma^2 = \frac{1}{m}\sum_i \left(z^{(i)} - \mu\right)^2$

2. Elementwise normalize values via remove estimated mean and divide by square root of standard variance. e is only for numerical stability

$z^{(i)}_{norm} = \frac{z^{(i)}-\mu}{\sqrt{\sigma^2+e}}$

Due to such normalization technic (of course if it's applied) applied to intermediate variables in layer **L** which by themselves are obtained from affine transformation happened in level ***L*** via using activations from previous layer ***L-1***.

$$z^{(i)[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$

It can be observed that $b^{[l]}$ in fact will be take into account during calculating μ in **step-1**. But then in **step-2** it is removed.

So, In particularly value of learnable parameter(variable) $b^{[l]}$ does not matter in case of using BN.

3. Use intermediate variable $z^{(i)}_{norm}$ (before applying activation function) to obtain $z^{(i)}$.

Apply finally scaling and shifting with two learnable parameters: $\tilde{z} = \gamma z_{norm} + \beta$

γ, β are vectors with dimensions equal to number of hidden units in layer.

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$

4. Parameters to be trained/optimized during train time:

γ, β (per layer vectors with dimensions equal to number of hidden units in layer)

Batch Normalization. Description for inference time.

At test/inference time you have no access to mini-batch because you typically have only single example. One possible way is estimate of μ, σ via exp. weighted average when exp.weight average compution happens across mini-batches which are feeded during training. Any other reasonable way to estimate this should work fine.

TensorFlow - https://www.tensorflow.org/api_docs/python/tf/nn/batch_normalization

Batch Normalization. Why it can make sense

1. Intermediate variables scale is in some control.
2. At least *Mean* and *Variance* is under some control.
3. Mean and variance is estimated on specific mini-batch and it has small regularization effect.
4. β and γ are trainable for whole network, not only for single Batch.
5. If use more big size of Mini-Batches – regularization effect is reduced
6. Empirically BN makes easily train deeper models