# CM Project Report

Valerio Mariani. Group Id:1, Project Id:5, ML

March 2021

# Introduction/Abstract

The project consists of a Neural Network with arbitrary topology and activation function, but mandatory l1 regularization, trained with:

- (A1) A standard momentum descent approach
- (A2) An algorithm of the class of proximal bundle methods

I start with the definition of a Multi-Layer Perceprton (MLP) in chapter 1, and the computation of loss function and elements in its subgradient in chapter 2, then I introduce the two training methods in chapters 3 and 4 and test them in chapter 5.

# Contents

# Chapter 1

# Architecture of The MLP

Each network is made out of a **stack of layers of units**, in particular it has:

- $N_l$ **hidden layers**, each of which has $N_h^m$ units, with $m \in [1, N_l]$. Those have $tanh$ activation function.
- Two layers with respectively $N_u$ and $N_y$ **input and output units**. Those units have identity function as activation function.

To distinguish activation functions of the hidden layer with those of input and output ones, I refer to (pointwise) activation function of layer m as $f^m$.

To connect units one to the other, I use **weighted connections** (directed arcs). Each connection is referred to as $w_{ij}^m$, and represents the connection that goes from unit j of the layer m to unit I of layer m+1. This way I can **accumulate connections** between units of layer m and units of layer m+1 **in matrices** $w^m = \{w_{ij}^m \text{ for m} \in [0 : N_h^{m+1}),\ j \in [0, N_h^m)\}$. In more, to do regularization I need to **compute the norm** of the vector that contains all of these parameters, so I simply take all the $w^m$, I **flatten** them into a 1-dimensional vector $w$ and compute its norm.

Each unit has its own **activation and potential**, and I denote activation and potential of the units of layer m as $a^m \in \mathbb{R}^{N_h^m \times 1}$ and $v^m \in \mathbb{R}^{N_h^m \times 1}$, thereby $a^m = f^m(v^m)$, where potential $v_i^m$ of the i-th unit of layer m is the weighted sum of the activations of units connected to the unit I of layer m, indeed I **compute them using the formula** $v^m = w^{m-1}a^{m-1}$.

To wrap up the notation that I introduced up to here:

- $N_l$ is number of hidden layers
- $N_h^m$ is number of units in m-th hidden layer
- $N_u$ and $N_y$ are respectively the number of input and output units
- $f^m$ is activation function of layer m
- $w^m$ is matrix of weights connecting units in layer m-1 to those in layer m

- $w$ is the vector that contains all the weights $w_{ij}^m$
- $v^m = w^{m-1}a^{m-1}$ is potential of activation of units in layer m
- $a^m = f^m(v^m)$ is activation of units in layer m

with the basic structure of the MLP defined, I use the network feeding the input units with external input, then iteratively compute activation of units of consecutive layers up to the output layer, and finally consider the output of my network as activation of the output units.

In more, I **fix a set** $X$ **of samples** made out of inputs and relative desired outputs and compute the loss of my network using an arbitrary **loss function**, denoted as $l(w, X)$, that is composed of a term that mesures the **error** that the network does in his task **plus an optional l1-norm penality term** for regularization.

The aim of the project is to fix a set X of patterns to do supervised training and solve the optimization problem of minimizing the loss function $l(w, X) = l(w)$.

In order to do this, I need to compute the **gradient of my loss function w.r.t. w, or rather an element in the subgradient in case the loss function is not differentiable in w**, and I compute it using the standard techniques introduced in the Back-Propagation algorithm.

# Chapter 2

# BackPropagation & (sub)gradient computation

I want to minimize my loss

$$l(w) = E + \epsilon||w||_1 = (\sum_{p=1}^{size(P)} E_p) + \epsilon||w||_1 \tag{2.1}$$

where $E$ is the total error computed **summing errors $E_p$ made by the network over each pattern** $p$ of $X$. To do this I need an element in the subgradient, and I denote this element as $\nabla l(w)$. Of course, if $l$ is differentiable in w, $\nabla l(w)$ is the actual gradient (the only element in the subgradient). $\nabla l(w)$ is made out of the **derivatives of loss $l(w)$ w.r.t each $w_{ij}^m$**:

$$\frac{\partial l(w)}{\partial w_{ij}^m} = \left( \sum_{t=1}^{T} \frac{\partial E_p}{\partial w_{ij}^m} \right) + \frac{\partial\, \epsilon||w||_1}{\partial w_{ij}^m}. \tag{2.2}$$

Now I have to compute the two main components of this summation: I start with the **computation of derivative of error on pattern p**. I use the **chain rule** to derive:

$$\frac{\partial E_p}{\partial w_{ij}^m} = \frac{\partial E_p}{\partial v_i^{m+1}} \frac{\partial v_i^{m+1}}{\partial w_{ij}^m} \tag{2.3}$$

that becomes:

$$\frac{\partial E_p}{\partial w_{ij}^m} = \delta_i^m a_j^{m-1}, \tag{2.4}$$

where $\delta_i^m$ is the so-called **error-propagation-term**, $a_j^{m-1}$ is activation of the unit as described in the previous chapter, and both of them are computed feeding the network with p-th pattern of X.

I compute error-propagation term of unit I in layer m during submission of pattern p as:

$$\delta_i^{Nl+1} = \frac{\partial E_p}{\partial a_i^m} f^{m'}(v_i^m) \quad \textbf{if} \quad \mathbf{m = Nl{+}1}, \tag{2.5}$$

$$\delta_i^m = (\sum_k w_{ki}^m \delta_k^{m+1}) f^{m'}(v_i^m) \quad \textbf{otherwise.} \tag{2.6}$$

For the purpose of this report, I can assume $E_p$ is **sum of squared error** $E_p(d, y = a^{Nl+1}) = ||d-y||_2^2$ (with d desired output) because that will be always used later, so formula 2.5 becomes:

$$\delta_i^m = (d_i - a_i^m) f^{m'}(v_i^m) \quad \text{with } m = Nl + 1. \tag{2.7}$$

In **matrix terms**, the computation of the error propagation terms becomes:

$$\delta^{Nl+1} \in \mathbf{R}^{Ny \times 1} = (d - a^{Nl+1}) f^{Nl+1'}(v^{Nl+1}), \tag{2.8}$$

$$\delta^m \in \mathbf{R}^{N_h^m \times 1} = (w^{m^t} \delta^{m+1}) f^{m'}(v^m), \tag{2.9}$$

and then I can compute the **matrices of the derivatives of error** $E_p$ w.r.t. each $w_{ij}^m$ as:

$$\frac{\partial E_p}{\partial w^m} = \{\frac{\partial E_p}{\partial w_{ij}^m}\} = \delta^{m+1} a^{m^t}. \tag{2.10}$$

Now for **regularization's norm-1 penality term** in equation 2.2, the problem with this term is that $||x||_1$ is **non-differentiable** when the element(s) $x_i = 0$, so in the case of a 0-weight I would actually need a **subdifferential or rather something that belongs to it**, that I could obtain e.g. by putting $\frac{\partial ||x||_1}{\partial x_i} = 0$. Although, in the case of machine computation, an exact 0 is **practically impossible** to obtain trough floating-point-operations, so I think the **two main routes** I can take to tackle the almost-0-weights problem are:

1. I **manually put each weight to 0** when the lasso regularization sends it **below some threshold**, and then when I go and compute the gradient I actually take an element in the subdifferential, putting $\frac{\partial ||w^m||_1}{\partial w_{ij}^m} = 0$. This way, I can technically obtain exactly-0-weights through l1-regularization, but on the other hand I would need an **hyper-parameter for the threshold**.

2. I **assume that I am never going to encounter an exact-0-weight** and so I do not consider this case in the gradient computation. This way I expect the lasso regularization to **send some weights towards zero** until they reach a point where they are close in absolute value to the epsilon parameter times a constant that will be known as learning rate, and then if the training does not stop, those weights start alternatively changing sign at each iteration.

At the end I **chose to take the first route** because I think this approach is more suited for this report, altought I believe the second approach is more used in the Machine Learning world when using lasso regularization.

All in all, at each weights update I manually put weights to 0 when they fall below $10^{-10}$, then to compute the regularization terms of $\nabla l(w)$, I use the formula in equation 2.11 for non-zero weights:

$$\frac{\partial \epsilon ||w||_1}{\partial w_{ij}^m} = \epsilon \frac{\partial ||w||_1}{\partial w_{ij}^m} = \epsilon \; sign(w_{ij}^m), \tag{2.11}$$

and put $\frac{\partial ||w||_1}{\partial w_{ij}^m} = 0$ when $w_{ij}^m = 0$.

To wrap up, to compute the partial gradient of the error function on a pattern p of X, I do the following steps:

1. **Forward Pass**: I **supply the network with the input in pattern p** and iteratively **compute activation** and potentials of units in each layer, proceeding **forward** through the network,

2. **Backward Pass**: I **use the desired output** of pattern p to compare it to the output of the network and **compute the error propagation terms** of units of each layer, proceeding backwards through the network,

3. **Actual computation of partial gradient of error on pattern p**: I use activations, potentials and propagation-terms to **compute the partial gradient**,

Then I **sum up the partial derivatives** of errors and **add the regularazion term** - computed as in equation 2.11 - like in equation 2.2 to come up with $\nabla l(w)$.

To **compute the norm of** $\nabla l(w)$ I **flatten** all the components into a 1-dimensional vector and compute its norm.

As for **computational cost** of the computation of $\nabla l(w)$: supposing that each layer has the same number of units $N_h$,

1. **both the costs of Forward and Backward pass** are dominated by the cost of $N_l$ matrix multiplications - each matrix being $R^{N_h \times N_h}$ - by column vectors - each vector being $R^{N_h \times 1}$ - so both of the passes cost $O(N_l\ N_h^2)$ flops each,

2. then there is another $O(N_l\ N_h^2)$ flops to compute **partial gradients of error w.r.t each weight**,

3. and finally the computation of the **l1-norm penalty term**, that involves taking the sign of each parameter in $O(N_l\ N_h^2)$ flops.

Overall, **computation of** $\nabla l(w)$ **takes** $O(N_l\ N_h^2)$ **flops** in total.

**Computing the loss function** involves a **forward pass** plus the computation of the **l1-norm of the vector that contains all of the parameters**, so it **can be done in** $O(N_l\ N_h^2)$ flops.

# Chapter 3

# (A1) Momentum descent

As for **momentum-based gradient descent** approach, I started from the formulae described in [9] for **Classical Momentum (CM)**:

$$d = -\alpha \nabla f(x_i) + \beta d^-, \tag{3.1}$$

$$x_{i+1} = x_i + d, \tag{3.2}$$

where $d$ and $d^-$ are the directions respectively chosen at step i and i-1, while $\alpha$ and $\beta$ are two parameters and $\nabla f$ is the gradient of the function $f$ to be optimized.

In my case, the function $f(x)$ to be minimized is the loss function $l(w)$, and $\nabla l(w)$ is **an element in the subgradient** of the loss function w.r.t w (that obviously is the actual gradient if $l$ is differentiable in w), computed as in chapter 2. Overall, my training routine is described by Algorithm 1.

**Algorithm 1:** Classical Momentum training

**Input:**
$w_0$: weights of the network to be trained
$\alpha$: learning rate
$\beta$: momentum parameter
$\epsilon$: regularization parameter
$\omega$: stopping treshold
$e_{max}$: maximum number of epochs to be done

**Procedure** `MomentumTrain`($w_0$, $\alpha$, $\beta$,$\epsilon$,$\omega$,$e_{max}$)**:**
    $w = w_0$, $e = 0$, $d^- = 0$
    **while** $||\nabla l(w)||/||\nabla l(w_0)|| > \omega$ *and* $e < e_{max}$ **do**
        $d$ = -$\alpha \nabla l(w)$ + $\beta d^-$
        $w = w + d$
        $e = e + 1$, $d^- = d$

As for **convergence** of the CM method, we know that CM (a.k.a. heavy ball method) has been proven in [3] to be **globally convergent for optimization of convex functions with Lipschitz continuous gradient** (with linear convergence if the function has also strong convexity) but unfortunately this is not the case for non-differentiable loss functions of neural networks, as we cannot assume neither convexity nor Lipschitz continuity.

Although, **local convergence** of Heavy-Ball method in the non-convex case can be proven for functions defined in [6, 8] as $C^{1+}$ functions, that is **differentiable functions whose gradient is strictly continuous**, if $\beta \in (0, 1)$ and $\alpha \in (0, \frac{2(1-\beta)}{L})$, with L Lipschitz constant of the gradient within the area in which the iterations lie [6].

In my case, **l is not differentiable** because of **l1-norm regularization**, in particular because $||x||_1$ is **non-differentiable** when the element(s) $x_i = 0$, so this convergence result cannot be applied.

Although, **if I were to give up regularization**, i would have a **twice continuously differentiable loss** because of summed-squared-error loss plus *tanh* and linear activation functions (all of which are twice continuously differentiable), and could compute second derivatives of $l$ w.r.t weights like e.g. in [1], so i could **prove local convergence** using **strict continuity of the gradient** (due to its differentiability) to provide a **limitation of the Lipschitz constant** of the gradient of the loss function **within an area in which all iterations lie**.
Mathematically, I have a real-valued-mapping $l : \mathbf{R}^n \to \mathbf{R} \in C^2$, where $n$ is the number of free parameters in the network. In this case the mapping $\nabla l : \mathbf{R}^n \to \mathbf{R}^n$ is strictly continuous on $\mathbf{R}^n$ with $lip\ \nabla l(w) = ||\nabla^2 l(w)|| < \infty$ for any $w \in \mathbf{R}^n$ [8] (Theorem 9.7) where $lip\ \nabla l(w)$ is the Lipschitz modulus of $\nabla l(w)$ as defined in [6, 8] and formally introduces the notion of strict continuity.
So, $\forall w \in \mathbf{R}^n \Rightarrow lip\ \nabla l(w) < \infty$ (because of definition of strict continuity) and if all iterations of the heavy-ball method lie in a compact, convex set $U \subset \mathbf{R}^n$ (e.g some ball centered in the initial guess $w_0$), then there exists finite $lip\ \nabla l(w)\ \forall\ w \in U$ (the function $lip\ \nabla l(w)$ is bounded from above), l is Lipschitz continuous on $U$ with Lipschitz constant $L = \sup_{w \in U} \{lip\ \nabla l(w)\}$ [8] (Theorem 9.2) and Heavy ball method is locally convergent for proper choice of the hyper-parameters $\alpha$ and $\beta$ [6].

As for **computational cost** of Algorithm 1, the **complexity of each epoch of training** is **dominated by the computation of** $\nabla l(w)$ that, as stated in Chapter 2, costs $O(N_l\ N_h^2)$ flops, and the total cost depends on how much epochs of training are required for the task at hand, although [7] showed that **CM can considerably accelerate convergence** to a local (or global) minimum **with respect to steepest descent method**, requiring $\sqrt{R}$ times fewer iterations than steepest descent to reach the same level of accuracy, where R is the condition number of the curvature at the minimum and $\beta$ is set to $(\sqrt{R}-1)/(\sqrt{R}+1)$ [9].

In practice, using the CM method to train neural networks is a very **common approach in Machine Learning**, both when optimizing differentiable and non-differentiable losses,

without theoretically ensuring convergence through computation Lipschitz constant. Convergence is instead obtained by empirically setting the $\alpha$ parameter to a value that is not too high, in such a way the algorithm does not result unstable in the tail of the convergence.

# Chapter 4

# (A2) Proximal Bundle Method

In this case I define an algorithm that makes use of a **bundle** $\beta$ of information about my loss function $l(w)$, encoded in a series of tuples $(w_i, l_i, g_i)$, where $w_i, l_i, g_i$ are respectively weights at iteration i, $l(w_i)$, **and an element in the subdiferential of** $l(w_i)$**, that I denote as** $\nabla l(w)$ **and compute like described in Chapter 2**, so each tuple can be used to obtain a plane that is tangent to the loss function in $w_i$.
The bundle will be **increasingly enriched** during the flow of the algorithm, so (if l is convex) it **defines a model of the loss function that gets better and better as the algorithm proceeds**.

In a generic bundle method, at iteration i the bundle is used by a **bundle function** as described in equation 4.1.

$$l_\beta(w) = max\{l_i + g_i^t(w - w_i) : (w_i, l_i, g_i) \in \beta\}. \tag{4.1}$$

In the Cutting Plane Method (CPM), **To find the next iterate, one has to solve the optimization problem** of finding $min\{l_\beta(w)\}$, and this problem is known as Master Problem (MP). Because equation 4.1 defines a **polyhedral function**, MP can be written as a **linear program**, formulated like in equation 4.2, making it easy to solve:

$$min\{l_\beta(w)\} = min\{v : v \geq l_i + g_i^t(w - w_i), (w_i, l_i, g_i) \in \beta\}. \tag{4.2}$$

**To turn the CPM into Proximal Bundle Method (PBM)** I need to **stabilize** the MP adding a stabilization term with relative stability parameter $\mu$, obtaining equation 4.3:

$$min\{l_\beta(w)+\mu||w-\bar{w}||_2^2/2\} = min\{v+\mu||w-\bar{w}||_2^2/2, v \geq l_i+g_i^t(w-w_i), (w_i, l_i, g_i) \in \beta\} \tag{4.3}$$

Because MP is now a **convex quadratic program** to be optimized by **constrained optimization**, one can just use an off the shelf solver to (relatively, in the sense that ad-hoc strategies exist, anyhow remaining in polynomial time because the hessian is positive semidefinite) efficiently **find the optimal solution (stability center) of MP at each iteration**.

All in all, my training routine for PBM is described in algorithm 2:

**Algorithm 2:** Proximal Bundle Method

**Input:**
$w$: initial values of weights of the network to be trained
$\epsilon$: regularization parameter
$\omega$: stopping treshold
$m1$: parameter for SS/NS decision
$\mu$: initial stability parameter
$\tau$: rescaling coefficent for stability parameter, $< 1$
$i_{max}$: maximum number of epochs to be done

1 **Procedure** PBM($w$, $\epsilon$, $\omega$, $m1$, $\mu$, $\tau$, $i_{max}$)**:**
2  $i = 0$, $\bar{w} = w$
3  $\beta = \{(\bar{w}, l(\bar{w}), \nabla l(\bar{w}))\}$
4  **while** *true* **do**
5  $w^* = argmin\{l_\beta(\bar{w}) + \mu||w - \bar{w}||_2^2/2\}$
6  **if** $\mu||w^* - \bar{w}||_2^2 < \omega$ ***or*** $i > i_{max}$ **then**
7  **break**
8  **if** $l(w^*) \leq l(\bar{w}) + m1(l_\beta(w^*) - l(\bar{w}))$ **then**
9  $\bar{w} = w^*$
10  $\mu = \tau\,\mu$
11  **else**
12  $\mu = (1 + \tau)\,\mu$
13  $\beta = \beta \bigcup \{(w^*, l(w^*), \nabla l(w^*))\}$
14  $i = i + 1$

**Convergence of the PBM** in his standard version can be proven **only in the case of convex optimization** (Although non-convex versions exist, e.g working on a local convexification of the objective function [4]), and it intuitively relies on the fact that the value $l_\beta(w^*) - l(\bar{w}) < 0$ because if the function is convex then $l_\beta(w) \leq l(w)\ \forall w$, and so the direction at each SS is of descent and the series $\{l(\bar{w})\}$ is a decreasing series, so the stopping condition in line 6 practically reduces to deciding if the distance $||w^* - \bar{w}||$ is small enough that algorithm can terminate with **a solution that has been provided by a good enough model of the objective function**.

**Iteration complexity** of the PBM is $O(1/\epsilon^3)$ [5], that is much worse than the optimal $O(1/\epsilon^2)$ for convex optimization of non-differentiable functions, but this method can - extreme aggregation apart - produce **satifying convergence speeds in the tail of the convergence**, when the **bundle** $\beta$ **becomes a good approximation of the objective function**.

Obviously, when the bundle gets to contain huge amounts of information, the oracle **solver can become slow at solving the MP**, but still remaining in polinomial time due to convexity. In more, **brutally reducing the bundle** pushes the method towards the classical subgradient method and **can cancel the reduction of iteration complexity** in the tail of the convergence, so it turns out it is better to keep the bundle as rich as possible in at least in some applications [2].

Finally, I think it is worth noting that this method can theoretichally count on a **working stopping criterion**, unlike other methods in his class.

# Chapter 5

# Experiments/Tests

In this chapter I **report the results of some experiments I did to test the training algorithms** that I introduced in the previous chapters.

These experiments involve some MLPs doing **regression on a univariate polynomial function and on ML-CUP20 dataset**.

## 5.1 experimental setup

For both the tasks I use a **single-layer network with 10** *tanh* **neurons** and random initial values of weights in $[-0.7, 0.7]$, then select hyperparameters for the algorithms differently for the two tasks.

To **compare the two methods** in terms of efficiency/efficacy (that is final loss values, iterations, time), I launched both the algorithms on 10 random initializations and collected some statistics, among which the **best value obtained during optimization**, that I denote as $l^*$ and use as my best approximation of a global optimum.

## 5.2 session 1: polynomial regression

First I **prepared myself a dataset** for univariate regression: I **defined a polynomial function** and added some **random white noise** $\epsilon_x$ to its output:

$$f(x) = x^3 - x^2 + \epsilon_x \tag{5.1}$$

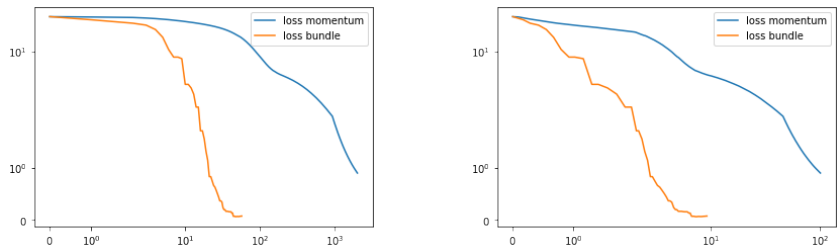
then I took 300 random points in the interval $[-1, 1]$, fed them to the function f and got my dataset composed of 300 $(x, d)$ patterns (d=desired output).

When training the network with **CM**, I choose for the **hyper-parameters** the values $10^{-2}, \beta = 5 \times 10^{-1}, \omega = 5 \times 10^{-2}, \epsilon = 10^{-5}$, then trained for a maximum of 2000 epochs.

When training with **PBM**, I used a value for $\epsilon = 10^{-5}$, and initial value of $\mu = 10$, with $\tau = 0.3$, $\omega = 10^{-}3$.

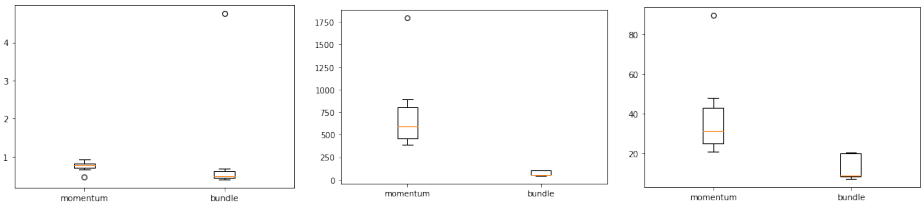
### 5.2.1 Results

I reported two plots to show how the gap $|l_i - l^*|/l^*$ varies w.r.t iterations and time in seconds during optimization in semi-logarithmic scale in Figure **??**; then some **boxplots with statistics on the results of optimization** in Figure 5.2.




(a) loss vs iterations. (b) loss vs time (seconds)

Figure (5.1) convergence plots in semi-logarithmic scale.




(a) loss (b) iterations (c) time (seconds)

Figure (5.2) boxplots with statistics of optimization with the two methods. Data was obtained running each algorithm on 10 different initializations.

Figure 5.1 and 5.2 show how, during this session of experiments, the **PBM** was usually able to come up with **better minima** than the ones obtained by CM, **in much less time**.

It can be noted from Figure 5.2a though, that **PBM happens to get stuck at poor local minima** from time to time, although I computed from the 10 initializations used to plot figure 5.2 that while the **best value of loss obtained from CM is 0.46**, **50% of the loss values obtained from PBM are below 0.49**.

Also, while the CM method **always stopped on its own thanks to its stopping condition, PBM reached its stopping condition only 6 out of 10 runs** and quit optimization after 100 iterations when running on the other initializations.

## 5.3 session 2: ML-CUP20

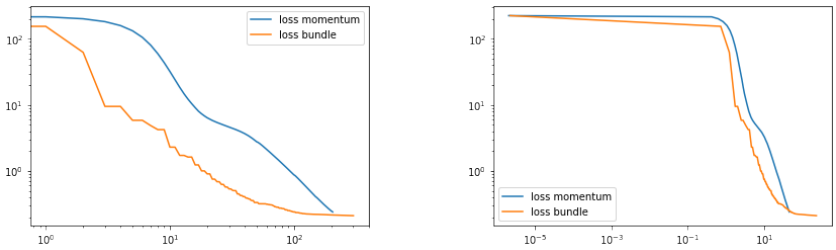
For this session of experiments I use the **dataset from the ML course**. This dataset requests regression of a function $f : \mathbf{R}^{10\times1} \to \mathbf{R}^{2\times1}$.

When training the network with CM, I choose for the hyper-parameters the values $\alpha = 1 \times 10^{-2}, \beta = 5 \times 10^{-1}, \omega = 10^{-1}, \epsilon = 10^{-12}$, then trained for a maximum of 2000 epochs.

When training with PBM, I used as hyper-parameters $\epsilon = 10^{-12}$, and initial value of $\mu = 10$, with $\tau = 0.3$.

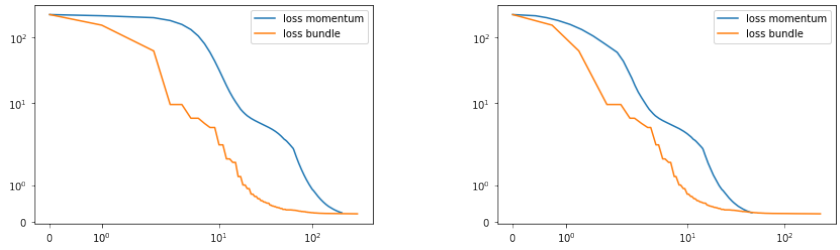
### 5.3.1 Results

I reported two plots to show how the gap $|l_i - l^*|/l^*$ varies w.r.t iterations and time in seconds during optimization in logarithmic scale in Figure 5.3 and in semi-logarithmic scale in Figure 5.4; then some **boxplots with statistics on the results of optimization** in Figure 5.5.



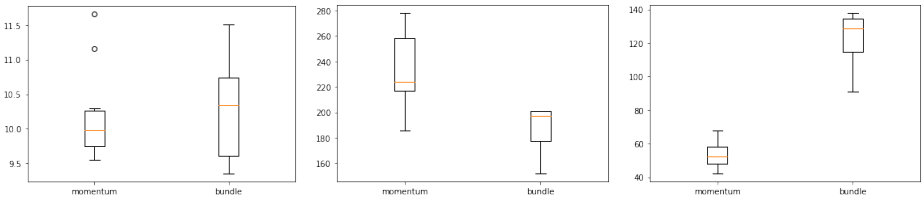

(a) loss vs iterations. (b) loss vs time (seconds)

Figure (5.3) convergence plots in logarithmic scale.




(a) loss vs iterations. (b) loss vs time (seconds)

Figure (5.4) convergence plots in semi-logarithmic scale.




(a) loss (b) iterations (c) time (seconds)

Figure (5.5) boxplots with statistics of optimization with the two methods. Data was obtained running each algorithm on 10 different initializations.

Figures 5.3 and 5.4 show how this second session confirms the improved convergence speed in the initial phase of the optimization, Although the boxplots in Figure 5.5 reveal that this time the bundle method took much more time to optimize and did not always return the best loss values (even if it gave the absolute best one I got). In more, also this time the bundle method did not reach his stopping conditions all the times, and had to quit optimization after 200 iterations 5 out of the 10 runs, even if I lowered the $\epsilon$ parameter by 10 times w.r.t the previous session of tests. Even the times PMB reached his natural stopping condition, Figure 5.5c shows that it always took more time than CM to optimize.

# Chapter 6

# Conclusions

For this project I implemented a model for artificial neural networks simulation, in particular for simulation of **Multi-Layer-Perceptron type networks, with l1 regularization**, plus two algorithms to train them.

The algorithms were implemented with a focus on the **optimization aspect**, that is keeping in mind that the **aim was non-convex optimization of (theoretically) non-differtiable functions**.

Because of non-convexity (and non-differentiability in the case of Classical Momentum), I was **not able to theoretically assume global convergence to global (unique) minima**, although both of the algorithms proposed **shown convergence to relatively satisfying accumulation points**.

The **Classical-Momentum/Heavy-Ball method** always resulted **convergent with appropriate parameters**, but requested **long times to converge** and did not always show the best quantitative results.

Also in the case of **Proximal-Bundle-Method**, **non-convexity did not allow me to assume global convergence**. Although, not only the algorithm was **usually able to find nice accumulation points**, but it also managed to **produce very well-fit networks**, converging in **very little time** if compared to Classical Momentum method during the first session of experiments.

# Bibliography


[1] Chris Bishop. *Exact calculation of the Hessian matrix for the multilayer perceptron.* 1992.

[2] Antonio Frangioni. "Standard bundle methods: untrusted models and duality". In: *Numerical Nonsmooth Optimization.* Springer, 2020, pp. 61–116.

[3] Euhanna Ghadimi, Hamid Reza Feyzmahdavian, and Mikael Johansson. "Global convergence of the heavy-ball method for convex optimization". In: *2015 European control conference (ECC).* IEEE. 2015, pp. 310–315.

[4] Warren Hare and Claudia Sagastizábal. "A redistributed proximal bundle method for nonconvex optimization". In: *SIAM Journal on Optimization* 20.5 (2010), pp. 2442–2473.

[5] Krzysztof C Kiwiel. "Efficiency of proximal bundle methods". In: *Journal of Optimization Theory and Applications* 104.3 (2000), pp. 589–603.

[6] Peter Ochs. "Local convergence of the heavy-ball method and ipiano for non-convex optimization". In: *Journal of Optimization Theory and Applications* 177.1 (2018), pp. 153–180.

[7] Boris T Polyak. "Some methods of speeding up the convergence of iteration methods". In: *Ussr computational mathematics and mathematical physics* 4.5 (1964), pp. 1–17.

[8] R Tyrrell Rockafellar and Roger J-B Wets. *Variational analysis.* Vol. 317. Springer Science & Business Media, 2009.

[9] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *International conference on machine learning.* PMLR. 2013, pp. 1139–1147.