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 its gradient 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 arbitrary activation function.
- Two layers with respectively N_u and N_y **input and output units**. Those units usually have identity function as activation function.[1]

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** $w^m = \{w_{ij}^m$ 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

[1]Output units can have softmax activation in the case of classification tasks with cross-entropy loss function

- 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**, and i compute it using the standard techniques introduced in the Back-Propagation algorithm.

Chapter 2

BackPropagation & 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, so the gradient $\nabla l(w)$ 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 δ_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:

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

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

For the purpose of this report, i can assume E_p is **sum of squared error** $E_p(d, y = x^{Nl+1}) = ||d-y||_2^2$ (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 gradient, 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 and **add the regularazion term** - computed as in equation 2.11 - like in equation 2.2 to come up with the gradient of my loss function.

To **compute the norm of my gradient** i **flatten** all the components into a 1-dimensional vector and compute its norm.

As for **computational cost** of the gradient computation: 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 the gradient 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 [5] 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 α and β are two parameters and ∇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 $l(w)$, so $\nabla l(w)$ is the gradient of the loss function w.r.t weights of the network, computed as in chapter 2. Overall, my training routine is described by Algorithm 1.

Algorithm 1: Classical Momentum training

```
Input:
$w$: weights of the network to be trained
$train_x$: input patterns
$train_y$: desired output patterns
$\alpha$: learning rate
$\beta$: momentum parameter
$\epsilon$: regularization parameter
$\omega$: stopping treshold
$e_{max}$: maximum number of epochs to be done
1 Procedure MomentumTrain($w$, $train_x$, $train_y$, $\alpha$,$\beta$,$\epsilon$,$\omega$,$e_{max}$):
2     $e = 0$, $d\hat{}- = 0$
3     while $||\nabla l(w_t)||/||\nabla l(w_0)|| > \omega$ and $e < e_{max}$ do
4         $d$ = -$\alpha \nabla l(w_t)$ + $\beta d^-$
5         $w = w + d$
6         $e = e + 1$, $d^- = d$
```

As for **convergence** of the CM method, we know that CM (a.k.a. heavy ball method) has been proven in [1] 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 Lipshitz continuity.

Although, **convergence in the differentiable, non-convex case can be proven** if, at each iteration x_i, $\beta \in (0, 1]$ and $\alpha \in (0, 2(1-\beta)/\lambda_i^1)$, where λ_i^1 is the largest eigenvalue of the hessian matrix in x_i, assuming λ_i^1 exists and is real.

As for **computational cost** of Algorithm 1, the **complexity of each epoch of training** is **dominated by the computation of the gradient** $\nabla l(w)$ 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 [4] 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 β is set to $(\sqrt{R}-1)/(\sqrt{R}+1)$ [5].

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 of eigenvalues of the hessian matrix. Convergence is instead obtained by empirically setting the α 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.

I also tried to think of a **way to mix in** some sort of **line-search** that could replace the fixed step-sizes proposed for CM and speed up convergence, but I came to the conclusion that **adding a line-searching mechanism to the CM method could not result in much more than a brutally simplified version of Conjugate Gradient (CG)**, so i decided to simply **opt for ML-ish approach** and select the hyperparametrical fixed step-sizes by **grid-searching**.

The main differences i can identify between CM and CG are that the hyper-parameter selection for CG is generally more expensive than the one for CM because one has to set more hyper-parameters[1]. At the same time, CG theoretically promises a (possibly[2]) much faster convergence near a minimum[3] in change of an higher[4] computational cost per-iteration.

[1]I think of line-search and selection of the formula for β_{CG} at least

[2]Not always much faster because speed of theoretical convergence is strongly conditioned by the number of parameters to be optimized, and those can be a lot in the case of a neural network

[3]In the sense that CG is shown to converge n-step quadratically when a minimum is near

[4]Indefinitely higher, because the cost of an iteration depends on the time spent on each line-search, which in turn depends on the nature of the function and on the parameters of the line-search. Has to be said that if one was to quit line-searching to save some number of function/gradient-computations, he would have to pay back by erratically proceeding in the search space by probably too short/long constant step-sizes.

Chapter 4

(A2) Proximal Bundle Method

In this case i define a **bundle** β of information about my loss $l(w)$ (encoded in a series of planes that are tangent to the function to be optimized), that will be **increasingly enriched** during the flow of a **Proximal Point Algorithm** (PPA).
The bundle β is used by a **bundle function** as described in equation 4.1.

$$f_\beta(x) = max\{f_i + g_i^t(x - x_i) : (x_i, f_i, g_i) \in \beta\} \tag{4.1}$$

Equation 4.1 **defines a model of the loss function that gets better and better as the algorithm proceeds**.

To find the next iterate, i have to solve the optimization problem of finding $min\{f_\beta(x)\}$, and because equation 4.1 defines a **polyhedral function**, my problem can be written as a **linear program** formulated like in equation 4.2, making it easy to solve:

$$min\{f_\beta(x)\} = min\{v : v \geq f_i + g_i^t(x - x_i), (x_i, f_i, g_i) \in \beta\}. \tag{4.2}$$

To turn the method described - that is known as Cutting Plane Method (CPM) - **into Proximal Bundle Method (PBM)** i need to **stabilize** the linear program described in equation 4.2 adding a stabilization term with relative stability parameter μ, obtaining the so called **Master Problem (MP)**, as in equation 4.3:

$$min\{f_\beta(x) + \mu||x - \bar{x}||_2^2/2\} = min\{v + \mu||x - \bar{x}||_2^2/2, v \geq f_i + g_i^t(x - x_i), (x_i, f_i, g_i) \in \beta\} \tag{4.3}$$

Because MP is a **convex quadratic program** to be optimized by **constrained optimization**, one can just use an off the shelf solver (in my case CVXPY) to (relatively[1]) efficiently **find the optimal solution (stability center) of MP at each iteration**.

[1]In the sense that ad-hoc strategies exist, anyhow remaining in polynomial time because the hessian is positive semidefinite.

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

Algorithm 2: Proximal Bundle Method

```
Input:
$w$: 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, $< 1$
$e_{max}$: maximum number of epochs to be done
1  Procedure PBM($w$, $train_x$, $train_y$, $\epsilon$,$\omega$,$m1$,$\mu$,$\tau$,$e_{max}$):
2      $e = 0,\ \bar{w} = w$
3      $\beta = \{(\bar{w}, l(\bar{w}), \nabla l(\bar{w}))\}$
4      while $true$ do
5          $w^* = argmin\{f_\beta(\bar{w}) + \mu||w - \bar{w}||_2^2/2\}$
6          if $\mu||w^* - \bar{w}||_2^2 < \omega$ or $e > e_{max}$ then
7              break
8          if $l(w^*) \leq l(\bar{w}) + m1(f_\beta(w^*) - l(\bar{w}))$ 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         $e = e + 1$
```

Convergence of the PBM in his standard version can be proven **only in the case of convex optimization**[2], and it intuitively relies on the fact that the value $f_\beta(w^*) - l(\bar{w}) < 0$ because if the function is convex then $f_\beta(w) \leq l(w)\ \forall w$, 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 $O(1/\epsilon^3)$ [3], 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 β becomes a good approximation of the objective function**.

[2]Although non-convex versions exist, e.g working on a local convexification of the objective function [2].

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 some applications.

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/Test

In this chapter i **report the results of some experiments i did to test the training algorithms** that i introduced in the previous chapters.
Both the sessions of experiments involve some MLPs doing **regression**: in the first session of experiment i do **regression on univariate polynomal functions** because i can do **plots to visualize how the networks fit the functions**. In the second session, i test the algorithms on the **ML-CUP20 dataset** to see if they are able to get nice results w.r.t the classical approaches described in the ML course.

5.1 session 1: polynomial regression

For this session of experiment i **prepared myself a dataset** for univariate regression: i **defined a polynomial function** and added some **random white noise** ϵ_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 X composed of 300 (x, d) patterns (d=desired input). The nice property of this dataset is that i can plot the function to fit and the actual output of the network, as shown in Figure 5.1:

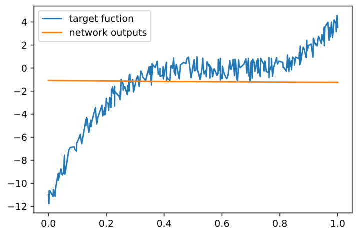


Figure (5.1) function to fit and output of the network **before training**

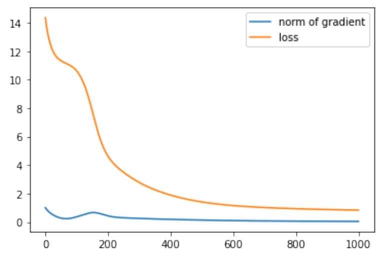
5.1.1 experimental setup

For this task i use **two network configurations**:

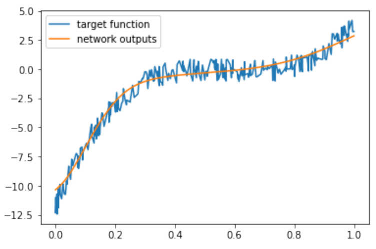
1. n1 is a three-layer network with respectively 4,8,4 *tanh* neurons and random initial values of weights in $[-0.7, 0.7]$,
2. n2 is a single-layer network with 10 *tanh* neurons and random initial values of weights in $[-0.7, 0.7]$.

5.1.2 A1

When training the networks n1 and n2 with CM, i choose for the hyper-parameters the values $\alpha = 10^{-2}, \beta = 10^{-3}, \omega = 10^{-2}, \epsilon = 10^{-12}$, then trained for a maximum of 2000 epochs. I report an example of a **learning curve** and the **output of a sample n1 network vs. the desired output** in Figure 5.2, and then a table with some **statistics on the results of training** in Table 5.1



(a) Learning curve.



(b) desired output vs actual output of the network. MSE=0.57

Figure (5.2) Training results of an n1 network with CM.

CONFIG	Mean	Variance
N1	**0.575**	0.006
N2	0.631	0.000

Table (5.1) Mean and variance of MSE after CM training on 10 networks.

5.1.3 A2

When training with PBM, i used a value for $\epsilon = 10^{-12}$, and initial value of $\mu = 10$, with $\tau = 0.1$.
I noticed that **this type of training does not work much well for networks of configuration n1 (in general with many layers)** probably because the addition of many layers to the structure makes the loss function very non-convex, and in general the PBM algorithm **happens to fail because the assumption $f_\beta(w^*) - l(\bar{w}) < 0$ does not hold in all iterations**. In my case this fact results in the **PBM algorithm being non-monotone and getting stuck on poor local minima almost all of the time when used to train networks of n1-configuration**. Although, i noticed that **on the n2-type networks the PBM algorithm most of the times manages to obtain a lower loss than the ones obtained by CM** and, most noticeably, when it reaches a nice accumulation point it **terminates on its own thanks to the stopping condition being reached after a series of NSs**, all of this happening in a **much briefer time** than the one requested from CM to

train (seconds vs minutes).

Figure 5.3 shows the **output of an n2 network trained with PBM vs the function to fit** and table 5.2 reports **some statistics on the PBM training**.

Comparing Figure 5.3 with 5.2b graphically shows how the n2-type network trained with PBM (Figure 5.3) **fits my dataset much better** than the n1-type network trained with CM (Figure 5.2b) despite the fact that **the second network is more complex and took more time to train**. In more, I think it is worth noting that of the **10 networks trained with PBM method, 9 got a final MSE lesser than 0.45**.

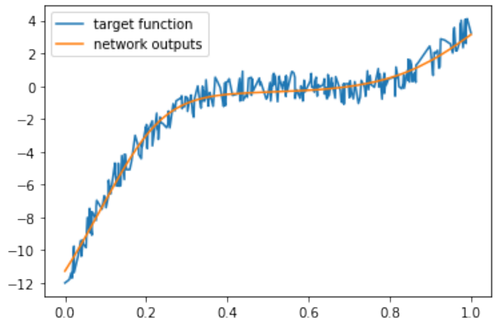


Figure (5.3) function to fit and output of an n1 network trained with PBM. MSE=0.33

CONFIG	Mean MSE	Variance
N1	5.340	**13.28**
N2	**0.767**	**1.211**

Table (5.2) Mean and variance of MSE after CM training on 10 networks.

5.2 session 2: ML-CUP20

For this session of experiments i use the **dataset from the ML course**. This dataset requests regression of a function $f : \mathbf{R}^{10\times1} \rightarrow \mathbf{R}^{2\times1}$.
For the purpose of this report i think it is enough to see if the algorithms i wrote manage to fit the training set, so i am not testing the networks on any validation/test set.

5.2.1 experimental setup

This time too i chose **two network configurations to test the algorithms**,

1. n1 is a two-layer network with respectively 20,10 *tanh* neurons and random initial values of weights in $[-0.7, 0.7]$,
2. n2 is a single-layer network with 20 *tanh* neurons and random initial values of weights in $[-0.7, 0.7]$.

5.2.2 A1

When training the networks n1 and n2 with CM, i choose for the hyper-parameters the values $\alpha = 5\,10^{-3}, \beta = 10^{-3}, \omega = 5\,10^{-3}, \epsilon = 10^{-5}$, then trained for a maximum of 2000 epochs.
I reported some **statistics on the results of CM training** in table 5.3, obtained from 10 networks of each type.

CONFIG	Mean MSE	Variance
N1	**6.549**	0.104
N2	8.184	0.041

Table (5.3) Mean and variance of MSE after CM training on 10 networks, ML-CUP dataset

5.2.3 A2

When training with PBM, i used as hyper-parameters $\epsilon = 10^{-5}$, and initial value of $\mu = 10$, with $\tau = 0.1$.
This time the **results i got are completely in favour of the PBM method**, as it **requested much less time to obtain better results on both configurations n1 and n2**. As before, some **statistics on the results of training** are reported in table 5.4.

CONFIG	Mean MSE	Variance
N1	**6.147**	0.652
N2	**6.24**3	0.031

Table (5.4) Mean and variance of MSE after PBM training on 10 networks, ML-CUP dataset

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, 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** has been compared to Conjugate Gradient method, noticing that the first has fixed parameters instead of line-search/closed-formulae but, depending on the task at hand, this does not necessarily represent a weakness. In my case, the 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**, and in fact I have frequently seen the algorithm **having to give away his monotonicity assumptions**, in particular **in the case of relatively deep architectures**. Although, not only the algorithm was **usually able to find an accumulation point**, but it also managed to **produce very well-fit small networks in the first session of experiment**s, converging in **very little time** if compared to Classical Momentum method, and **generally better results in the second session of experiments** w.r.t the Classical Momentum method.

Bibliography

[1] 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.

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

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

[4] 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.

[5] 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.