On the Use of Stochastic Line-Search Methods to Solve Binary Classification Problems

Thibault Lahire*, Youssef Diouane*, Michel Salaün*
*ISAE-SUPAERO, Université de Toulouse, 31055 Toulouse, FRANCE
thibault.lahire@student.isae-supaero.fr
youssef.diouane@isae-supaero.fr
michel.salaun@isae-supaero.fr

Abstract – In many contemporary optimization problems, such as binary classification, it is computationally unfeasible to evaluate the entire objective function or its derivatives. To remedy this situation, stochastic algorithms exploiting subsampled approximations of the objective are used, but this method jeopardizes the guarantees of convergence classically obtained through deterministic techniques.

Moreover, there has been an increasing interest for non-convex formulations of data-related problems. For such instances, the aim is to develop methods that converge to second-order stationary points.

This research project contributes to this rapidly expanding field by presenting a stochastic algorithm based on second-order considerations, computed for a subsampling model of the objective.

1 Introduction

A well-known class of problems in the field of Machine Learning is binary classification problems [5]. The aim of binary classification is to separate a certain number of objects into two classes. These objects $z_i,\ i \in [\![1; n]\!]$, called data, covariates or examples, are labelled with $y_i \in \{0; 1\}$ to indicate to which class any covariate z_i belongs.

Once the algorithm has *learned* from data, it is able to assign a new covariate z to either label 1 or 0. Binary classification is often used in spam detection. For example, the user gives an email to the algorithm and the latter is able to conclude : "This mail is a spam" or "This mail is not a spam". Binary classification is then one of the branches of what is today called Artificial Intelligence.

In this paper, we first define the problem to solve (chapter II). Then, after having analyzed some data, we reformulate the problem since the latter is ill-posed in the case of a linear separation (chapter III). The three algorithms we will study in this paper are introduced in chapter IV. Then, we study the behavior of these algorithms on the toy problem for different starting points (chapter V) and for different sampling rates (chapter VI). In the last chapter, the three algorithms are compared on a real data set.

2 Problem statement

Let $D = \{(z_i, y_i), i \in [\![1; n]\!]\}$ be the training set, where $z_i \in \mathbb{R}^p$, which means that each object z_i has p features. As explained before, our problem is to assign a new example $z \in \mathbb{R}^p$ to either class C_0 (labelled 0) or C_1 (labelled 1).

To assign the new covariate z to the right class, the idea is to create a function $\hat{y}$ that takes z as an argument and such that $y = \hat{y}(z)$. In this project, only hyperplanes (i.e. a straight line in dimension 2) will be used to separate the two classes. In many cases, the data are not linearly separable, but we will restrict ourselves to this case.

A hyperplane is the kernel of a linear form. Let a be the associated linear form, called *activation function*. According to Riesz's theorem, $\exists! x \in \mathbb{R}^p,\ a(\cdot) = \langle x, \cdot \rangle$. Note that x represents the weights/coefficients of the hyperplane, it is the vector being sought to ensure a good separation.

The new observation z is categorized as C_0 if $a(z) = \langle x, z\rangle = x^\top z \leq 0$, and C_1 otherwise. In this project, we choose $\hat{y}(z) = \Phi(a(z)) = \Phi(x^\top z)$, where Φ is constructed from the sigmoid function, i.e, $\Phi(w) = \frac{1}{1+\exp(-w)}$ for all $w \in \mathbb{R}$.

This choice is coherent, because $\Phi(w) \underset{w\to-\infty}{\longrightarrow} 0$ and $\Phi(w) \underset{w\to+\infty}{\longrightarrow} 1$. Indeed, if $z \in C_0$, then $x^\top z \leq 0$, then $\Phi(x^\top z) \simeq 0$. If $z \in C_1$, then $x^\top z \geq 0$, then $\Phi(x^\top z) \simeq 1$.

Since $\hat{y}$ intrinsically depends on x, we rename $\hat{y}$ as $\hat{y}_x$ to emphasize this dependence.

A lot of decision frontiers separating a training set may exist. To classify as best as possible, the *best* hyperplane is searched, i.e. the hyperplane that guarantees the maximum distance to its nearest points.

A good way to reach this goal is to quantify the quality of the classification of a given observation z_i using a *loss function* $f_i : \mathbb{R}^p \longrightarrow \mathbb{R}$ associated to the datum (z_i, y_i). A possible choice we will use for the loss function is $f_i(x) = (y_i - \hat{y}_x(z_i))^2$. Note that, for a loss function, $\forall x \in \mathbb{R}^p$, $f_i(x) \geq 0$ and when $f_i(x) \simeq 0$, the weight vector x ensures a good classification of the datum (z_i, y_i). If the vector x does not classify well the covariate z_i, then $f_i(x) \simeq 1$.

The aim of binary classification is to find x^* such that every observation z_i belongs to the right class, i.e. $\forall i \in [\![1; n]\!]$, $f_i(x^*) \simeq 0$. In other words, the aim is to find x^* minimizing the *empirical loss* :

$$x^* \in arg\min_{x\in\mathbb{R}^p} \frac{1}{n}\sum_{i=1}^{n} f_i(x) \tag{1}$$

In this project, we will focus on problems in the following form :

$$x^* \in arg\min_{x\in\mathbb{R}^p} \frac{1}{n}\sum_{i=1}^{n} (y_i - \Phi(x^\top z_i))^2 \tag{2}$$

with Φ the function introduced some lines above. In what follows, the cost function will also be noted F. In practice, $n \gg 1$, i.e. $n \simeq 10^6$ or 10^9. Therefore, evaluating the empirical loss is computationally expensive or quasi-unfeasible. To overcome this obstacle, only a limited number of f_i is evaluated at each iteration.

All the algorithms used in this project are able to evaluate a limited number of f_i. We note s the percentage estimated of the whole sum and S_k the subset of indices selected at the iteration k. Three algorithms are tested : a second-order algorithm using line-search methods, called ALAS, the classical stochastic gradient algorithm, called SGD, and another version of SGD, called ADAGRAD. These three algorithms will be described later.

3 A reformulation of the problem

3.1 A first representation in dimension 2

First, we have to observe the data and think of the formulation of the problem again. To work on a problem computationally feasible, we create our own set of data, as featured in Fig. 1. This example is inspired from [5]. The class C_0, labelled 0, is the realization of a Gaussian random variable of mean $\begin{pmatrix}5\\10\end{pmatrix}$ and covariance matrix $\begin{pmatrix}10 & 4\\4 & 4\end{pmatrix}$. The class C_1, labelled +1, is the realization of an other Gaussian random variable of mean $\begin{pmatrix}10\\0\end{pmatrix}$ and covariance matrix $\begin{pmatrix}10 & 5\\5 & 4\end{pmatrix}$. For this data set, $n = 200$, with the same quantity of covariates for each set.

In this case, a hyperplane (i.e. a straight line in dimension 2) can separate the two classes. Approximatively, the hyperplane $\Delta_0 = \{(z_1, z_2) \in \mathbb{R}^2 : -10z_1 + 15z_2 = 0\}$ should work well.

An important thing to be noted is that the straight line $\Delta_1 = \{(z_1, z_2) \in \mathbb{R}^2 : 10z_1 - 15z_2 = 0\}$ is the same hyperplane. However, Δ_0 classifies the data perfectly, whereas, with Δ_1, all data are in the wrong class. As a consequence, the function to be minimized to solve the problem presented in equation (2) has the shape presented in Fig. 2.

When one of the three algorithms used in this project (SGD, ADAGRAD, or ALAS) tries to minimize the empirical loss function, the optimization fails when the starting point is on the red plateau on Fig.

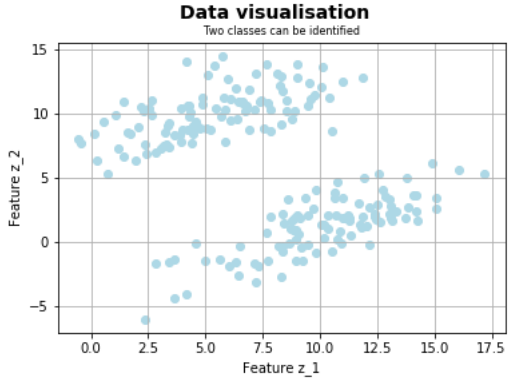

Figure 1: Illustration of the binary classification problem with a domain space of dimension 2

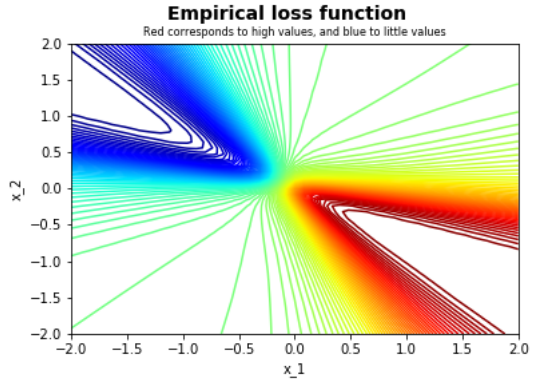

Figure 2: Empirical loss function to minimize

2. This is because the gradient and the Hessian are very close to zero.

Moreover, with this cost function, although the optimization succeeds, the classification is not perfect, as is the case in Fig. 3. Indeed, if Δ_0 is a good hyperplane associated to the weight vector $x^* = \begin{pmatrix} -10 \\ 15 \end{pmatrix}$, the hyperplane $\Delta_2 = \{(z_1, z_2) \in \mathbb{R}^2 : -100z_1 + 150z_2 = 0\}$ associated to the weight vector $x^{**} = \begin{pmatrix} -100 \\ 150 \end{pmatrix}$ is also a good separation for the data. The problem is that $F(x^{**}) < F(x^*)$. F is closer to 0 when $||x||$ has a high value. Therefore, the three algorithms will prefer $x^{***} = \begin{pmatrix} -100 \\ 200 \end{pmatrix}$, as is the case on Fig. 3, even though a vector co-linear to x^* would fit better.

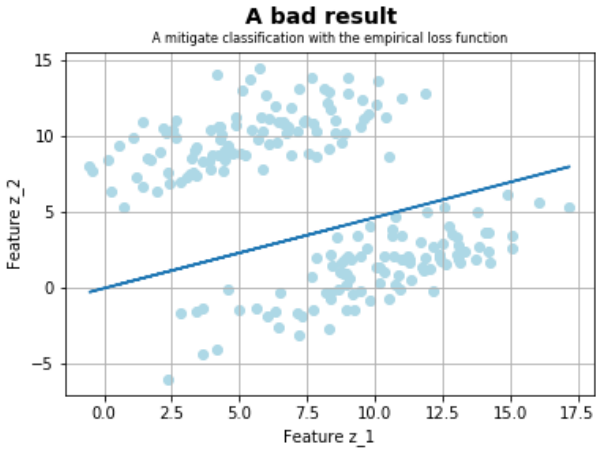

Figure 3: Example of what can be obtained with a naive formulation of the problem

3.2 A new function to minimize

To remedy this situation, the idea is to penalize the vectors of high norm. The new problem to solve is then :

$$x^* \in arg \min_{x \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} (y_i - \Phi(x^\top z_i))^2 + \frac{\delta}{2n} ||x||_2^2 \tag{3}$$

The new cost function to minimize is also noted F. δ is a constant that will equal 1 in this project, but [3] has obtained more accurate results when δ is equals to the Lipschitz constant of the empirical loss. From now on, the function to minimize has the shape presented in Fig. 4, with one global minimum.

With this cost function, the three algorithms find the global minimum wherever the starting point is located. The weight vector x has a *small* norm, and the separating hyperplane has a shape depicted in

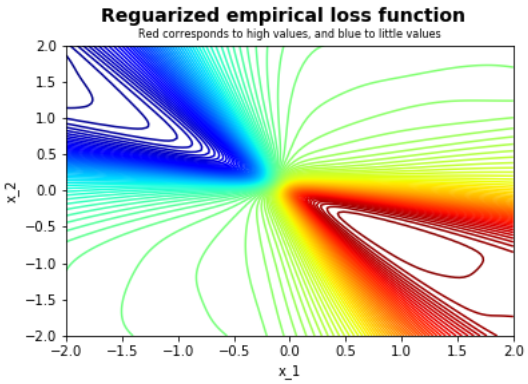

Figure 4: The new corrected empirical loss function to minimize

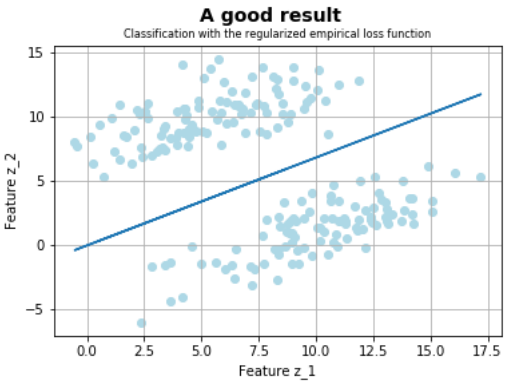

Figure 5: Example of what can be obtained with a good formulation of the problem

Fig. 5.

In this paper, the objective functions may be those presented in (2) or in (3), or they may be multiplied by n, which has no influence on the coordinates of the minimum found.

3.3 Some limitations on the datasets

As explained above, the aim of binary classification is to find x^* such that every observation z_i belongs to the right class, i.e. $\forall i \in [\![1; n]\!]$, $f_i(x^*) \simeq 0$. In other words, we introduced the *regularized empirical loss* F :

$$F(x) = \sum_{i=1}^{n} f_i(x) + \frac{1}{2}||x||_2^2 \tag{4}$$

Note that we multiplied eq. (3) by n. The word *regularized* comes from the penalization of areas of high norms by adding $||x||_2^2$. Our goal is to find x^* minimizing the F :

$$x^* \in arg \min_{x \in \mathbb{R}^p} F(x) \tag{5}$$

In our case, we use a non-convex cost function, i.e. $f_i(x) = (y_i - \Phi(x^\top z_i))^2$, where Φ is the sigmoid function. With this formulation we are trying to separate the two classes of our problem with a hyperplane parametrized by $x \in \mathbb{R}^p$.

It is important to keep in mind the following simple remarks :

- $F(0) = \sum_{i=1}^{n} f_i(0) + \frac{1}{2}||0||_2^2 = \sum_{i=1}^{n} (\frac{1}{2})^2 = n/4$
- When all the data are in the wrong class, $F(x_{bad}) > n$ because $\sum_{i=1}^{n} f_i(x_{bad}) \simeq n$
- When all the data are in the proper class, $\sum_{i=1}^{n} f_i(x_{good}) \simeq 0$

Hence, it is useless to test one of our methods on a dataset which is not linearly separable (or quasi linearly separable). Indeed, imagine a dataset in 2D where the two classes are two zero-centered circles. The first class is located on a radius r_1 and the second class is located on a radius r_2, with $r_1 < r_2$. Whatever the hyperplane we take, we always have $n/2$ data points in the proper class. So, $\sum_{i=1}^{n} f_i(x) \simeq n/2$. However, $\sum_{i=1}^{n} f_i(0) = n/4$. From the optimization point of view, it implies that the minimum is realized for $x = 0$, which does not have any sense for a hyperplane.

To conclude, it is important to evaluate our algorithms on a dataset where a hyperplane is able to ensure a quit good classification. To be more precise, we need datasets where a hyperplane is able to realize a classification where $\sum_{i=1}^{n} f_i(x_{opt}) < n/4$. For an unknown dataset found on the net, we can't be sure that this criterion is met.

4 Algorithms used : what is expected

This chapter describes the way the three algorithms used work and what is expected from them. These three algorithms share some similarities.

First, at each iteration, they never evaluate the whole sum in the empirical loss function but only a subset randomly chosen of s percent of the total data set. At a given iteration, the objective function that the algorithm tries to minimize is not the real cost function of the problem. The point found at the end of the iteration is considered as the starting point for the next iteration, that will be run with a different objective function, i.e. a function associated to another subset. The convergence of this method depends on the algorithm used.

Furthermore, if the algorithm reaches a point that satisfies the stopping criteria, it does not mean the algorithm has found a local minimum for the whole problem. It can only be derived that the point is a minimizer of the partial sum considered at the iteration. However, this point is a good candidate as a solution of the whole problem. Therefore, the algorithm will terminate if the next iterations are also successful. More precisely, the algorithm will terminate if the next $100s$ iterations are successful.

These explanations are summarized in algorithm 1.

Algorithm 1 General structure

```
procedure OPTIMIZATION(cost function F)
    Initialization : K, number of iterations, x_0 starting point, and s, percentage of the dataset
    for k = 0, 1, ..., K do
        Choose randomly a batch (or subset) S_k of the dataset
        Perform an optimization step on this subset starting from x_k to find a better point x_new
        x_k ← x_new
        if a stopping criterion is met 100s times consecutively
            return current point
    return last point
```

4.1 Classical Stochastic Gradient Descent (SGD)

The Stochastic Gradient Descent is widely used in binary classification problems [2]. If we consider a non-regularized empirical loss function, x_{k+1} is obtained from x_k with the following equation :

$$x_{k+1} = x_k - \alpha_k \frac{1}{\mid S_k \mid} \sum_{i \in S_k} \nabla_x f_i(x_k) \tag{6}$$

The SGD algorithm, as suggests its name, requires differentiable functions in the sum. Therefore, SGD can only ensure a first-order convergence. If the algorithm finds a saddle point or a small plateau, it may terminate if the next $100s$ iterations are successful.

The convergence of SGD is proven under certain conditions on the step α_k [2]. The algorithm will find a local minimum (or a saddle point...) in a finite time if :

$$\sum_{k=1}^{\infty} \alpha_k = \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \alpha_k^2 < \infty \tag{7}$$

In practice, SGD works (even if it has not yet been proven) with a constant and very small α_k, typically $\alpha_k \simeq 10^{-3}$. However, finding the optimum constant depends on the problem considered. If the constant is too high, the algorithm has difficulties finding the minimum. Indeed, even if the opposite of the gradient is a good direction, the next point is so far from the previous point that the objective function could not decrease. The evolution of the real value of the objective function (i.e. the whole sum) over iterations in this case is presented on Fig. 6.

If the constant is too small, the algorithm will find a local first-order stationary point, but it will need a large quantity of iterations. The evolution of the real value of the objective function (i.e. the whole

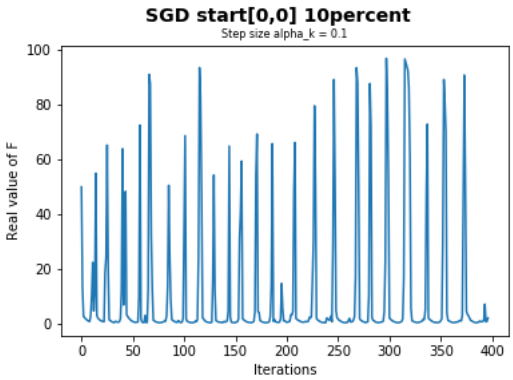

Figure 6: Example of the evolution of the value of the entire cost function with a step size too large in SGD algorithm

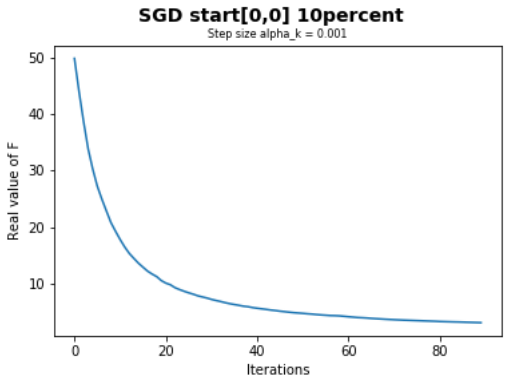

Figure 7: Example of the evolution of the value of the entire cost function with a step size too small in SGD algorithm

sum) over iterations in this case is presented on Fig. 7. NB : $F((0,0)) = n/4 = 50$.

In this project, we choose a sequence $(\alpha_k)_k$ that ensures the convergence theoretically. We choose : $\alpha_k = 1/k$. Is it a good choice ? In a more general setting, we could use $\alpha_k = \eta/k$, η the initial step size. Tuning the hyperparameter η will be discussed soon.

For SGD, the stopping criterion is a gradient norm near to zero.

4.2 ADAGRAD

ADAGRAD means *Adaptive Gradient*. It is a modified version of SGD, where the step size is modified at each iteration given the previous iterations. The algorithm is then able to adapt itself to the configuration of the region it explores. The initial step size is $\eta = 1$, but this is multiplied with the elements of a vector $G_{j,j}$ which is the diagonal of the outer product matrix :

$$G = \sum_{\tau=1}^{k} g_\tau g_\tau^\top \tag{8}$$

where g_τ is the gradient at iteration τ. The formula for an update is then :

$$x_{k+1} = x_k - \eta \; diag(G_{j,j})^{-1/2} g_k \tag{9}$$

Thanks to this adaptation, ADAGRAD should be more efficient than SGD. As is also the case for SGD, the stopping criterion is a gradient norm near to zero.

4.3 Tuning the initial step size ?

Hyperparameter tuning is an important issue of the machine learning / deep learning community. In deep learning, the choice of the initial step size (also called initial learning rate) has huge consequences on the training of deep neural networks. Indeed, even if the step size is updated such a way that the conditions presented in eq. (7) are satisfied, a too large initial step size may cause a divergence, and a too small initial step size increases the optimization duration.

Methods have been found to tune *in a smarter way* the initial step size in [6]. To sum up quickly what can be done, the first naive idea is to do a loop on η and to keep the η that fits best to the context studied. But this means performing a lot of optimizations from start to end, which is very long. To be more efficient, [6] proposes to perform only a few iterations on small batches and to select the initial step size that lead to the best decrease of the cost function.

Since the learning rate of ADAGRAD is updated *in a smarter way* than the learning rate of SGD, a good intuition is to think that ADAGRAD is more robust to the initial learning rate. This is true in practice. As written above, $\eta = 1$ was chosen for SGD and ADAGRAD. As it can be seen from Fig. 17

to Fig. 28, this is a catastrophic choice for SGD, whereas ADAGRAD deals with it better. For this set of experiments, another choice of η for SGD would have yielded better performance, but not an equivalence to ADAGRAD.

We now present ALAS, which is the algorithm of the main paper this research project is based on. ALAS has no initial learning rate to tune, thanks to the backtracking line-search. However, ALAS has other hyperparameters (especially those for the backtracking line-search), but their choices seem to have a lower impact than the choice of η for SGD and ADAGRAD.

4.4 ALAS

ALAS is the acronym for A Line-search Algorithm for Second-order result. ALAS was proposed in [1], and inspired from [4]. In this algorithm, a line-search step intervenes to optimize the distance between x_k and x_{k+1}, and this allows a decrease in the number of necessary iterations. Moreover, ALAS uses the Hessian, i.e. second-order information.

The advantages are a second-order convergence (meaning that the algorithm will find a real local minimizer, and not a saddle point), and an acceleration of the convergence thanks to Newton or regularized-Newton techniques.
The main disadvantage of ALAS is the computation of the Hessian, which can be computationally expensive.

The algorithm reaches the stopping criteria when the Hessian is definite-positive and the gradient near to zero. Let $\epsilon > 0$ be a tolerance on the gradient norm and $\epsilon^{1/2}$ a tolerance on the smallest eigenvalue λ_k of the Hessian H_k. The stopping criteria are a gradient smaller than ϵ and λ_k greater than $-\epsilon^{1/2}$.
x_{k+1} is obtained from x_k thanks to the relation $x_{k+1} = x_k + \alpha_k d_k$ where α_k is the result of a line-search, and d_k the direction which can be chosen in three distinct ways depending on the value of the smallest eigenvalue λ_k of the Hessian at the iteration k.

First case : $\lambda_k < -\epsilon^{1/2}$, then $d_k = -\lambda_k v_k$, where v_k is the eigenvector associated to the eigenvalue λ_k such that $||v_k|| = -\lambda_k$. This direction is called *negative curvature* direction.
Second case : $\lambda_k > ||g_k||^{1/2}$, where g_k is the gradient at the iteration k. Then a *Newton* direction is chosen, i.e. $d_k = -H_k^{-1} g_k$.
Third case : Else : $d_k = -(H_k + (||g_k||^{1/2} + \epsilon^{1/2}) I_p)^{-1} g_k$. This is the *regularized Newton* direction.

5 Comparisons of the three algorithms ($s = 100\%$) for different starting points

In this chapter, we compare the performance of ALAS, SGD and ADAGRAD from different starting points.

As indicated in the title of this chapter, we compare the three algorithms on the same objective function, i.e. the entire empirical loss function F.

The stopping criterion for this chapter is a tolerance on the gradient for SGD and ADAGRAD, to which is added a tolerance on the smallest eigenvalue of the Hessian for ALAS. We choose $g_{tol} = 10^{-5}$ and $H_{tol} = \sqrt{g_{tol}}$.

The first studied starting point is the origin, which means that the user starts the algorithms without any information on the function he has to minimize. It is the most commonly used starting point, it will be used in the other chapters.

The second studied starting point is $x_0 = (1.75, -1)$. With the data we have created thanks to the normal random variables (see chapter III), this point corresponds to an opposite gradient that indicates a direction that will not lead to the minimum.

The third starting point is far from the minimum, it is the occasion to compare the time the three algorithms take to find it.

5.1 $x_0 = (0, 0)$

The evolution of the values of the cost function F with SGD is represented on Fig. 8. The point found after one iteration is so far from the interesting region that the value of F for this point is very high. This can be explained by the evolution of the step size. At the beginning, the step size is 1, which leads the current point of the algorithm to an uninteresting region.

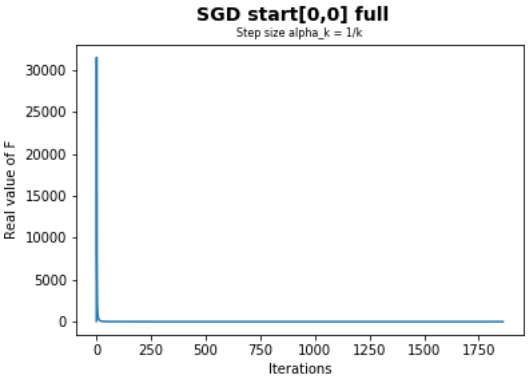

Figure 8: Evolution of the value of the entire cost function with SGD

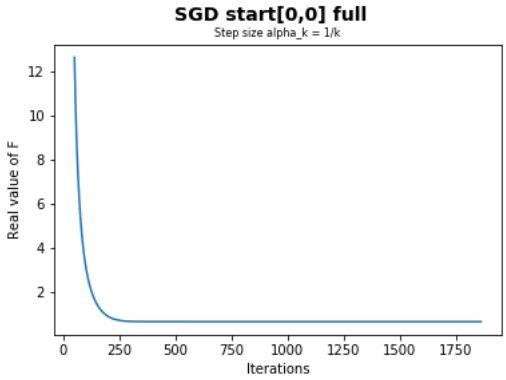

Figure 9: Evolution of the value of the entire cost function with SGD after 50 iterations

After 1000 iterations, the current points are in the region of interest, but the step size is so small that the algorithm needs a large number (1862) of iterations to reach the minimum. This can be observed in Fig. 9.

Whereas SGD needs 1862 iterations (or function calls, and CPU time = 12.6) to reach the stopping criteria, ADAGRAD needs 440 iterations (or function calls, and CPU time = 2.8) and ALAS only 27 iterations (or function calls, and CPU time = 1.2) to get the same result. It is interesting to note that ALAS only uses *Newton* directions to solve this problem : ALAS exploits the second-order information to be faster. On Fig. 10, the different approaches of the three algorithms are represented. Note that the first one hundred iterations of SGD are out of bounds.

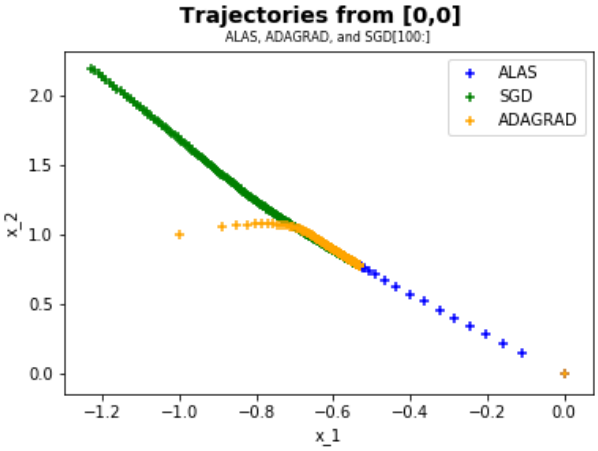

Figure 10: Trajectories of the three algorithms. ALAS is the most efficient thanks to *Newton*

5.2 $x_0 = (1.75, -1)$

This point is interesting since the direction suggested by the opposite of the gradient will not lead to the minimum without a direction change. The starting point, the point found at the first iteration by ADAGRAD and the point found at the first iteration by SGD are on the same straight line (the SGD-point is not represented on Fig. 11). However, as it can be seen on Fig. 11, the direction indicated by the methods used by ALAS is the direction of the minimum. The first iteration of ALAS uses a

negative-curvature-type direction, i.e. a direction based on the Hessian.

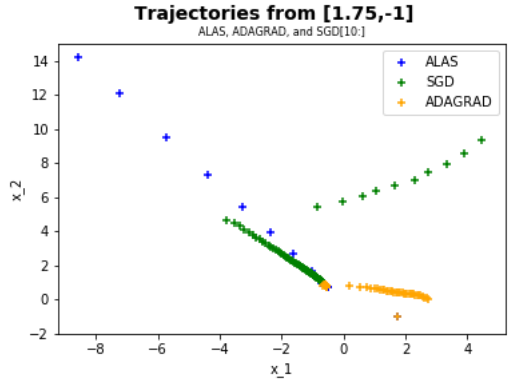

Figure 11: Trajectories of the three algorithms. ALAS is the most efficient thanks to *negative-curvature*-type direction

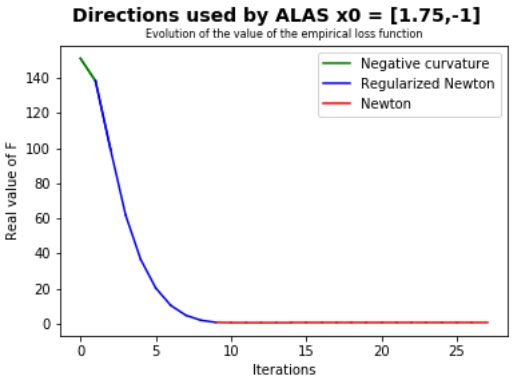

Figure 12: Evolution of the objective function and the directions used over iterations

The directions used by ALAS are represented on Fig. 12. ALAS needs 28 iterations (37 function calls, and CPU time = 0.8) to reach the stopping criteria, whereas ADAGRAD needs 224 iterations (or function calls, and CPU time = 1.4) and SGD needs 882 iterations (or function calls, and CPU time = 5.8) for the same result.

5.3 $x_0 = (10, -10)$

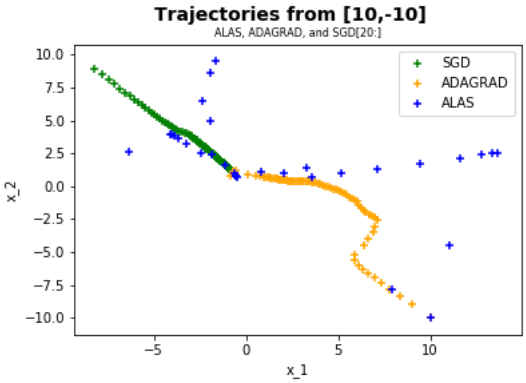

Figure 13: Trajectories of the three algorithms. ALAS is the most efficient thanks to its use of second-order

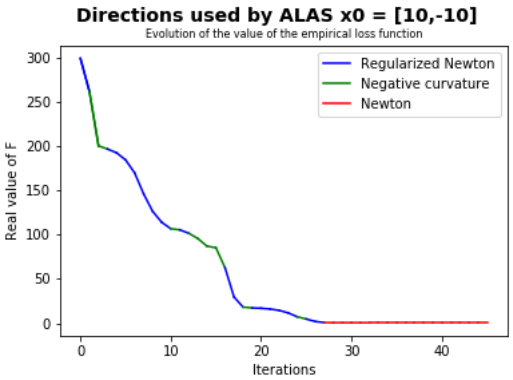

Figure 14: Evolution of the objective function and the directions used over iterations

As represented on Fig. 13, SGD and ADAGRAD go in the direction indicated by the opposite of the gradient. The minimum is found after a large amount of iterations since the step size becomes small over time. The first twenty iterations of SGD are out of bounds of the picture. It takes 1790 iterations (or function calls, CPU time = 11.8) for SGD and 309 iterations (or function calls, CPU time = 1.9) for ADAGRAD to reach the stopping criteria, whereas ALAS only needs 46 iterations (and 131 function calls, CPU time = 1.3) to obtain the same result.

When the start is far from the minimum, ALAS remains the most efficient algorithm thanks to the use of second-order information, as represented in Fig. 14 . ALAS uses alternatively *regularized Newton* and *Negative Curvature* to find a trajectory more efficient than a gradient descent.

6 Toy problem in dimension 2

In this chapter, the three algorithms are tested on the same toy problem in dimension 2 introduced in chapter III. The starting point $x_0 = (0, 0)$ is the same for ALAS, SGD and ADAGRAD. The idea is to compare the number of iterations, the number of function calls, and the time it takes to reach the stopping criteria. Comparing the number of iterations makes a study of the directions used by ALAS possible, however, it is more interesting to show the number of function calls, since it is the longer step of the program. Indeed, for ALAS only, the number of function calls is not equal to the number of iterations, due to the line-search step.

For all the algorithms, the number of gradient calls is equal to the number of iterations. For ALAS, the number of Hessian calls is equal to the number of iterations. Let c_p be the time it takes to get a function call. p is the dimension of the problem. It is commonly found in the literature that the time required to evaluate the gradient is equal to pc_p, and the time it takes to evaluate the Hessian is equal to $\frac{p^2}{2}c_p$. However, the computation of the function loss can be optimized. For instance, the scalar product $x^\top z_i$ can be saved in a variable and reused for the computation of the function, the gradient, and the Hessian.

As a consequence, the number of function calls is not best way to compare algorithms, even if it remains better than counting the number of iterations. The best comparison basis is the processor time (or CPU time).

Note that the comparison has always been realized in the previous chapter for a sampling size of $s = 100\%$. For this toy problem, we will now compare the three algorithm for a sampling size of $s = 10\%$, because the number of data is small ($n = 200$). Moreover, with $s = 10\%$, randomness is introduced in the three algorithms, that's why histograms are required.

Besides the stopping criteria introduced in the previous chapter, a limit on the number of iterations is also imposed. Indeed, as it will be seen, the algorithms may not terminate without this condition. The maximum number of iterations is 5000.

However, the data created are linearly separable. If the algorithms find a minimum, we know that this minimum corresponds to a good classification. Note that, when data are not linearly separable, the algorithms may found a minimum but it will not correspond to a good classification.

The objective function is a sum of loss functions, the latter equal 1 if the corresponding covariate is not well classified, and 0 otherwise. Then we have $0 \leq \sum_{i=1}^{n} f_i(x) \leq n$. Therefore, if this sum is lower than $\beta < 1$, it means that all data are well classified. We exploit this information by adding a new stopping criterion, e.g. $\sum_{i=1}^{n} f_i(x) \leq 0.4$.

6.1 Comparisons of the three algorithms in terms of iterations

Running ALAS, SGD and ADAGRAD with this stopping criteria leads to the bar graph presented in Fig. 15. ALAS is the most efficient algorithm from the point of view of iterations. A proportion of 20% of all runs have terminate in 12 iterations, which means that a good result was found after 2 iterations only, the following ones were controlling the redundancy of the satisfaction of the stopping criteria with different subsets.

6.2 Comparisons of the three algorithms in terms of function calls

The results presented in Fig. 16 have few differences with those presented in Fig. 15. Indeed, for ADAGRAD and SGD, the number of iterations equals the number of function calls. It is not the case for ALAS due to the line-search step. For this example, ALAS uses very few line-search and accepts most of the time the first point it finds.

6.3 Comparisons of the three algorithms in terms of time

For this toy problem, ALAS remains the most efficient algorithm in terms of time (see table 1). ADAGRAD and SGD are similar, and take a longer time to find the minimum. The standard deviations are related to the standard deviations presented in Fig. 15 and 16, since the number of iterations equals to the number of function calls for ADAGRAD and SGD.

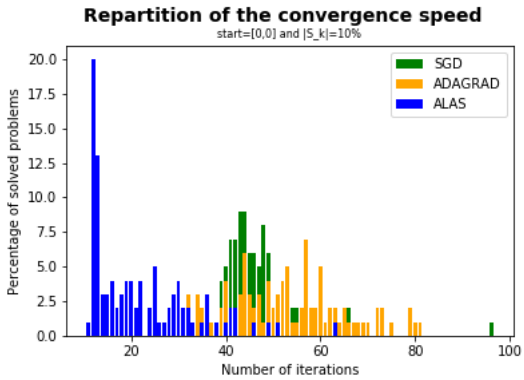

Figure 15: Statistical distribution of the three algorithms in terms of iterations

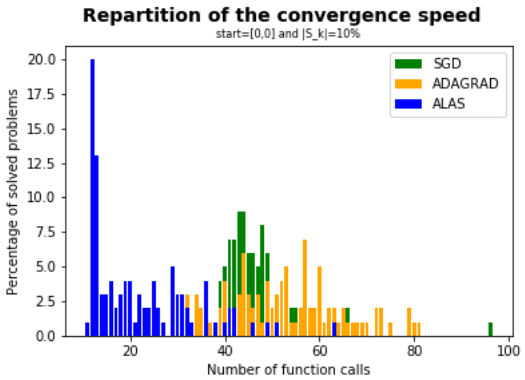

Figure 16: Statistical distribution of the three algorithms in terms of function calls

Table 1: Comparisons of the three algorithms in terms of time

CPU-time	ALAS	SGD	ADAGRAD
Mean	0.12	0.87	0.77
Standard deviation	0.05	0.77	0.49

7 With real data : LIBSVM ijcnn1

To finish this report, an optimization is run on a real data set ("ijcnn1") where $p = 22$ and $n = 49990$. Three cases are studied for ALAS, ADAGRAD and SGD : $s = 100\%$, $s = 5\%$, and $s = 1\%$. For each case, we will compare the three algorithms on different objective functions : the function defined in equation (2), i.e. the *classical* empirical loss function, and the function defined in (3), i.e. the *regularized* empirical loss function.

In all configurations, SGD is less efficient than ALAS and ADAGRAD (see Fig. 17 to 28). The step size $\alpha_k = 1/k$ ensures theoretically the convergence, but it is here a bad choice. A constant and small step size ($\alpha_k = 10^{-3}$) would fit better [1].

In this chapter, the concept of *epoch* will be used. One epoch of a solver that uses $s\%$ of the sum is equivalent to running $100/s$ iterations of the given method.

7.1 $s = 100\%$

In this case, 20 iterations are presented, which is equivalent to 20 epochs. On the *classical* empirical loss, ALAS is more efficient than ADAGRAD. This is all the more remarkable with the evolution of the gradient norm in this case, where ALAS is going to meet the stopping criteria faster than ADAGRAD on Fig. 18.

With the *regularized* empirical loss, ADAGRAD ensures a decrease faster than ALAS over the first iterations, but ALAS becomes more efficient after a certain time. See Fig. 19 and Fig. 20.

The first iterations of ALAS are *regularized-Newton*-type directions. Whether with the *classical* empirical loss or the *regularized* empirical loss, ALAS (CPU-time : 3.5) needs twice as much time than ADAGRAD (CPU-time : 7) to achieve one epoch or iteration.

7.2 $s = 5\%$

In this case, 200 iterations are presented, which is equivalent to 10 epochs. On the *classical* empirical loss, ALAS is more efficient than ADAGRAD. After a certain number of iterations, the norm of the gradient oscillates, as represented on Fig. 22, since it is very difficult to find the exact minimum with the use of samples. However, the points found by the algorithms are in the region of interest.

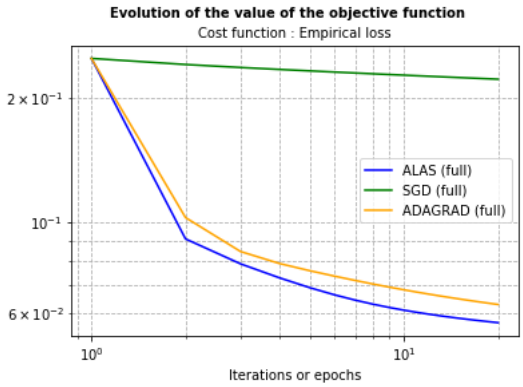

Figure 17: Evolution of the objective function with ALAS, SGD and ADAGRAD for $s = 100\%$ on the classical empirical loss

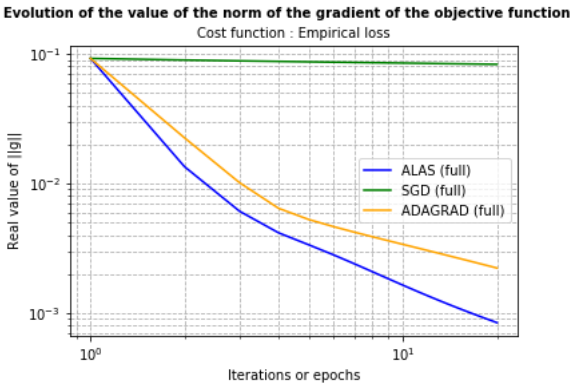

Figure 18: Evolution of the norm of the gradient of the objective function with ALAS, SGD and ADAGRAD for $s = 100\%$ on the classical empirical loss

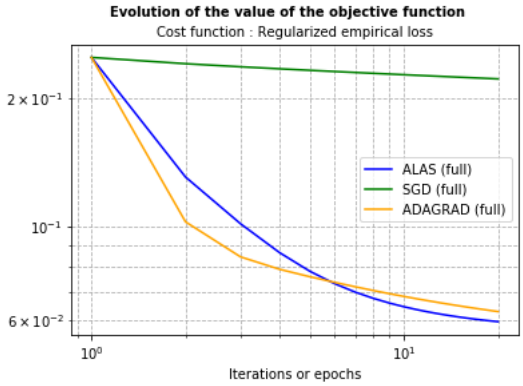

Figure 19: Evolution of the objective function with ALAS, SGD and ADAGRAD for $s = 100\%$ on the regularized empirical loss

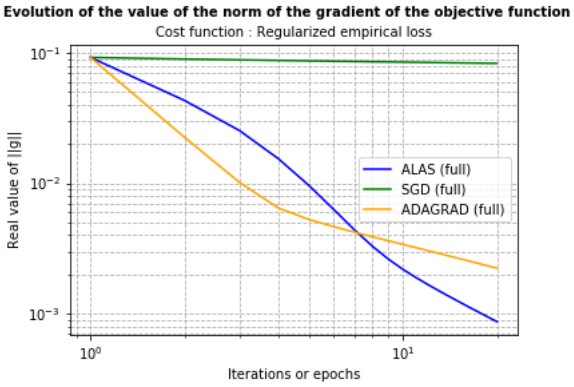

Figure 20: Evolution of the norm of the gradient of the objective function with ALAS, SGD and ADAGRAD for $s = 100\%$ on the regularized empirical loss

On the *regularized* empirical loss, the performance of ALAS and ADAGRAD are equivalent in terms of iterations.

The first iterations of ALAS are *regularized-Newton*-type directions. Whether with the *classical* empirical loss or the *regularized* empirical loss, ALAS (CPU-time : 40) needs as much time as ADAGRAD (CPU-time : 39) to achieve one epoch or 20 iterations.

7.3 $s = 1\%$

In this case, 1000 iterations are presented, which is equivalent to 10 epochs. On the *classical* empirical loss, ALAS is more efficient than ADAGRAD. After a certain number of iterations, the norm of the gradient oscillates, as represented on Fig. 26, since it is very difficult to find the exact minimum with the use of samples. However, the points found by the algorithms are in the region of interest.

With the *regularized* empirical loss, ADAGRAD ensures a decrease faster than ALAS over the first iterations. Then, ADAGRAD and ALAS oscillate around a certain value. See Fig. 27 and Fig. 28.

The first iterations of ALAS are *regularized-Newton*-type directions. Whether with the *classical* empirical loss or the *regularized* empirical loss, ALAS (CPU-time : 191) needs as much time as ADAGRAD (CPU-time : 192) to achieve one epoch or 100 iterations.

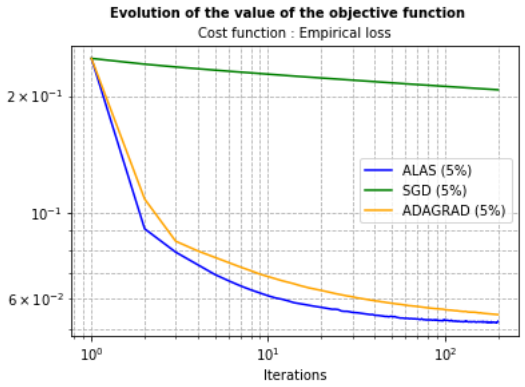

Figure 21: Evolution of the objective function with ALAS, SGD and ADAGRAD for $s = 5\%$ on the classical empirical loss

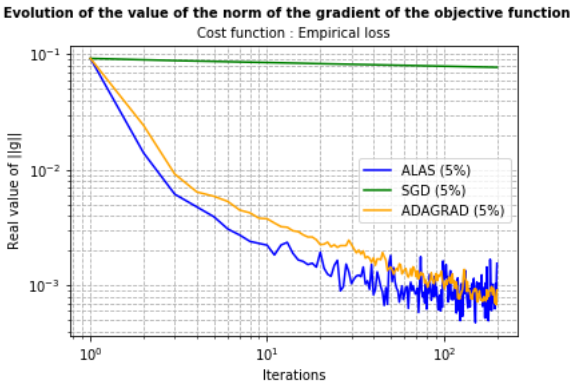

Figure 22: Evolution of the norm of the gradient of the objective function with ALAS, SGD and ADAGRAD for $s = 5\%$ on the classical empirical loss

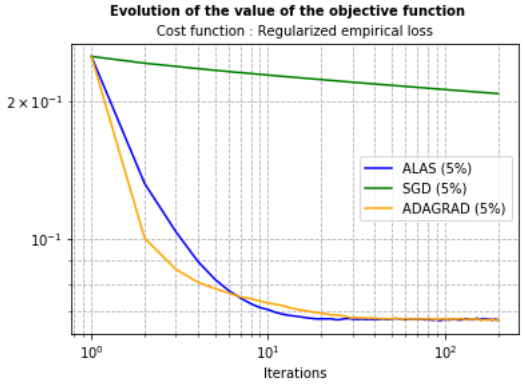

Figure 23: Evolution of the objective function with ALAS, SGD and ADAGRAD for $s = 5\%$ on the regularized empirical loss

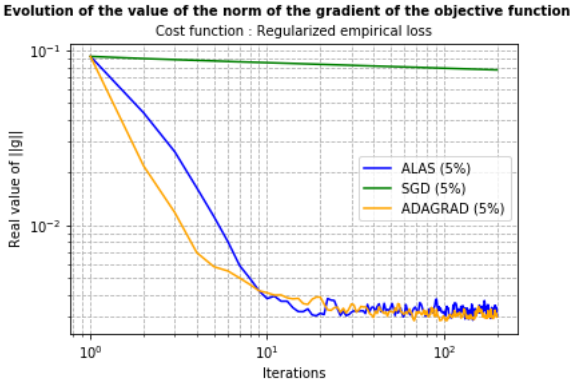

Figure 24: Evolution of the norm of the gradient of the objective function with ALAS, SGD and ADAGRAD for $s = 5\%$ on the regularized empirical loss

7.4 Comment on the results

When working on an exact function (which means $s = 100\%$ in our project), ALAS, with the use of second-order information and line-search, finds the minimum in fewer iterations than first-order methods (namely ADAGRAD and SGD). Moreover, the point found is a minimum, and not a saddle-point.

When working on inexact functions (which means $s < 100\%$ in our project), two cases must be separated. When working on the *classical* empirical loss, it is better to use ALAS for the first iterations, because it ensures an efficient decrease. When working on the *regularized* empirical loss, ADAGRAD is also a good candidate, it depends on which percentage of the total data set we use.

8 Conclusion

The aim of this research project was to compare a new second-order method using backtracking line-search (ALAS) with classical first-order algorithms (SGD and ADAGRAD) on the problem of binary classification using a non-convex cost function.

The main advantages of ALAS boils down to the following points :

- ALAS guarantees second-order conditions on the point found. Hence, the point cannot be a saddle point or a plateau, but it is a true local minimum.
- ALAS uses the Hessian matrix to compute the next iterate. Hence, when near to a minimum, the convergence is quadratic (similar to Newton's method).

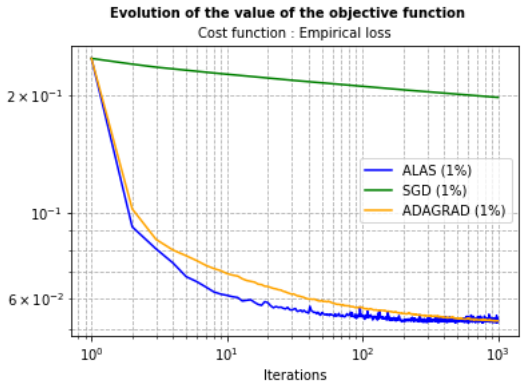

Figure 25: Evolution of the objective function with ALAS, SGD and ADAGRAD for $s = 1\%$ on the classical empirical loss

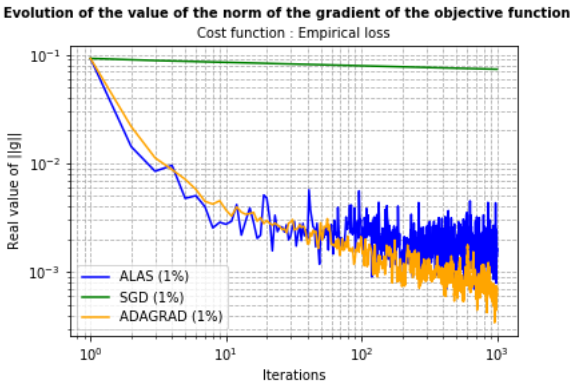

Figure 26: Evolution of the norm of the gradient of the objective function with ALAS, SGD and ADAGRAD for $s = 1\%$ on the classical empirical loss

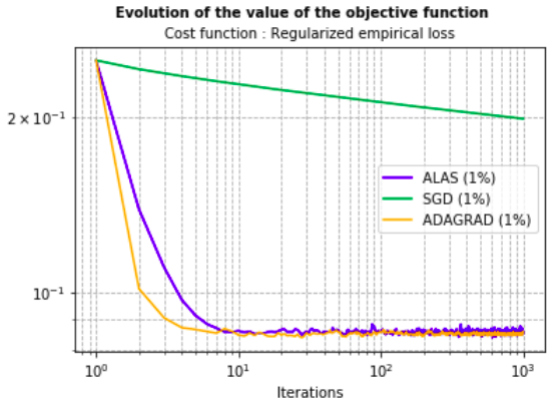

Figure 27: Evolution of the objective function with ALAS, SGD and ADAGRAD for $s = 1\%$ on the regularized empirical loss

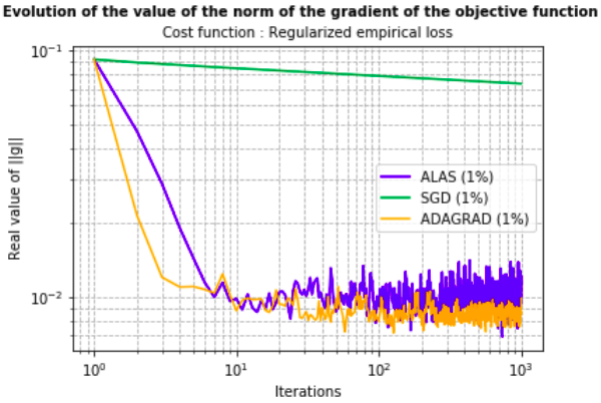

Figure 28: Evolution of the norm of the gradient of the objective function with ALAS, SGD and ADAGRAD for $s = 1\%$ on the regularized empirical loss

- The backtracking line-search ensures a more stable decrease in the cost function. Moreover, contrarily to SGD and ADAGRAD, ALAS seems less sensitive to hyperparameters tuning.

However, the main disadvantage of ALAS is the cost of the computation of the Hessian matrix. This is the well-known problem of *the curse of dimensionality*. In practice, when the dimension is above a few hundred, computing the Hessian matrix is too costly. Hence, this method is not suitable for training deep neural networks for example, for which first-order methods seem to remain the most powerful paradigm.

Given this main disadvantage, the following ideas may nonetheless be interesting. First, if the dimension is *not too high*, a program that combines ALAS for the first iteration and first-order methods for the next iterations could be efficient. Indeed, as seen in the last chapter, ALAS can be more efficient on the first iteration than first-order methods. Moreover, since ALAS is less sensitive to hyperparameters tuning than first-order methods, we might wonder if the difference between x_0 (starting point) and x_1 (first point found by ALAS) can give interesting information on a good initial step size of the first-order method that will be applied to find x_2. This question remains open.

Acknowledgment

The author would like to thank Youssef Diouane and Michel Salaün for their constant support and their availability.

References

[1] E. Bergou, Y. Diouane, V. Kungurtsev, and C.W. Royer. A subsampling line-search method with second-order results. Technical report, ISAE-SUPAERO, arXiv preprint arXiv:1810.07211, 2017.

[2] L. Bottou. *Stochastic Gradient Descent Tricks*, volume 7700, pages 421–436. Montavon G., Orr G.B., Müller KR. (eds) Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science, 2012.

[3] R. Chen, M. Menickelly, and K. Scheinberg. Stochastic optimization using a trust-region method and random models. *Mathematical Programming*, 169:447–487, 2018.

[4] C. W. Royer and S. J. Wright. Complexity analysis of second-order line-search algorithms for smooth nonconvex optimization. *SIAM Journal on Optimization*, 28:1448–1477, 2018.

[5] O. Simeone. A brief introduction to machine learning for engineers. *CoRR*, abs/1709.02840, 2017.

[6] Leslie N. Smith. No more pesky learning rate guessing games. *CoRR*, abs/1506.01186, 2015.