Optimisation Convexe Séquentielle
Projet, premier semestre 2021–2022

Optimisation convexe séquentielle : mise en oeuvre et comparaisons

---

Article: *Dual Averaging Methods for Regularized Stochastic Learning and Online Optimization (Xiao, 2009)*

Auteurs:
Bourarch Nassim
Durand Homer

Professeur:
Wintenberger Olivier

January 15, 2022

Contents

1 Structure du projet **3**
- 1.1 Les critères de comparaison . . . 4
  - 1.1.1 La précision et le temps de calcul . . . 4
  - 1.1.2 La sparsité et la forme des solutions . . . 4
  - 1.1.3 La sensibilité à l'initialisation (et aux hyperparamètres) . . . 6
  - 1.1.4 La robustesse aux attaques . . . 6

2 Les hyperparamètres **7**
- 2.1 Sélection des hyperparamètres . . . 7
- 2.2 Rôle des hyperparamètres : Le cas de l'algorithme GD . . . 8

3 Précision et temps de calcul **11**

4 Sparsité et forme des solutions **13**

5 Sensibilité à l'initialisation (aux hyperparamètres) **16**
- 5.1 Le taux d'erreur par rapport à l'initialisation . . . 16
- 5.2 sensibilité au pas de descente : Comparaison de SGDproj et SEGpm . . . 18
- 5.3 Illustrations de la dépendance à l'initialisation . . . 20

6 Robustesse aux attaques **22**

7 Méthode de moyennisation duale régularisée **25**
- 7.1 Motivations . . . 25
- 7.2 Application de la RDA aux régularisations usuelles . . . 25
- 7.3 Propriétés théoriques . . . 27
  - 7.3.1 Bornes de regret . . . 28
- 7.4 Performances . . . 28
  - 7.4.1 Sélection des hyperparamètres . . . 28
  - 7.4.2 Comparaison avec les algorithmes du cours . . . 29
- 7.5 Sensibilité aux hyperparamètres : Le cas l_1/l_2^2 . . . 32

1 Structure du projet

L'objectif de ce projet est de mettre en oeuvre et surtout de comparer différentes méthodes d'optimisation (pour calibrer une SVM permettant de classifier les '0' sur des données MNIST), il va donc nous falloir tout d'abord esquisser les critères sur lesquelles on va pouvoir comparer les différentes méthodes. C'est l'objet de cette section.

On commencera néanmoins par présenter assez brièvement 'l'incontournable' base de donnée MNIST (accessibles ici).

La base de donnée MNIST est composée de 70000 images (de 28×28 pixels en nivreau de gris) de chiffres écrits à la main ainsi que de leur étiquettes (correspondant au chiffre de l'image en question).

Notre objectif étant de classifier uniquement les '0' avec une SVM, on va retraduire le problème (tout en ajoutant une ordonnée à l'origine), On notera :

- $a_i \in \mathbb{R}^{28\times 28+1}$, $i \in [\![1, 70000]\!]$ la $i^{ième}$ image (à laquelle on a ajouté un 1).
- $b_i \in \{-1, 1\}$ sont étiquette (égale à 1 si le chiffre est un 0).

On va alors mettre en oeuvre une SVM pour classifier les 0 avec une perte charnière régularisée avec une pénalité l_2 (en fonction des algorithmes) et, pour le cas particulier de l'algorithme *Extended Kalman Filter* (EKF), une perte logistique avec une pénalité l_2. Notons que dans tout les cas on se restreindra à des solutions dans une boule l_1 (pour essayer de mettre à profit la sparsité du problème 1.1.2).
On séparera nos données en un échantillon d'entraînement de taille $n = 60000$ et un échantillon de test de taille $n' - n = 10000$.

Les algorithmes d'optimisations mis en oeuvre seront les suivants: [1]

Algorithme	Acronyme	perte
Gradient Descent	GDproj	charnière + l_2, sur la boule l_1
Stochastic Gradient Descent	SGDproj	charnière + l_2, sur la boule l_1
Stochastic Mirror Descent	SMDproj	charnière, sur la boule l_1
Stochastic Exponentiated Gradient +/-	SEGpm	charnière, sur la boule l_1
AdaGrad	Adaproj	charnière, sur la boule l_1
Adam	Adamproj	charnière, sur la boule l_1
Online Newton Step	ONS	charnière + l_2, sur la boule l_1
Extended Kalman Filter	EKF	logistique
Stochastic Randomized Exponentiated Gradient +/-	SREGpm	charnière, sur la boule l_1
Exp2	SBEGpm	charnière, sur la boule l_1
Regularized Dual Averaging Methods[1]	RDA...	charnière (+ l_1, l_2, $\frac{l_1}{l_2}$, ...)

Table 1: Récapitulatif des algorithmes mis en oeuvre

[1] On mettra en oeuvre plusieurs des algorithmes précédent munis d'une amélioration par RDA pour différentes pertes (dans la section 7).

1.1 Les critères de comparaison

1.1.1 La précision et le temps de calcul

Comme nous sommes dans un problème d'apprentissage supervisé, la qualité d'un classifieur donné g est facilement quantifiable, il s'agira ici de ce que nous appellerons la performance du classifieur[2] g :

$$\mathrm{Acc}(g) := \frac{\#\left\{i \in [\![n+1; n+n']\!] \,|\, g(a_i) = b_i\right\}}{n'}. \tag{1}$$

Une première considération pour comparer les algorithmes en jeu est donc de comparer la performance des différents classifieurs donnés par les algorithmes. La performance représente le nombre de classifications correctes.

Néanmoins, comme les algorithmes n'ont pas tous la même complexité, il ne nécessitent pas forcément le même nombre de calculs pour atteindre une performance donnée. Il semble judicieux de prendre en compte cela dans notre évaluation de leurs qualités. Le temps de calcul permet d'avoir une "mesure" de la complexité algorithmique.

Ainsi le premier critère de comparaison que nous allons considérer, en section 3, s'intéresse à la performance des classifieurs issus des différents algorithmes ainsi que le temps nécessaires pour obtenir de tels classifieurs. Une brève digression sur les regrets "empiriques" atteints par les différentes méthodes conclura cette section.

1.1.2 La sparsité et la forme des solutions

Les données MNIST consistant en images de chiffres centrés, il est judicieux de penser qu'un grand nombre de pixels (coordonnées de a_i pour $i \in [\![1; n+n']\!]$) sur les bords des images sont tout le temps inactifs (nulles) et ce indépendamment de l'étiquette b_i en question.

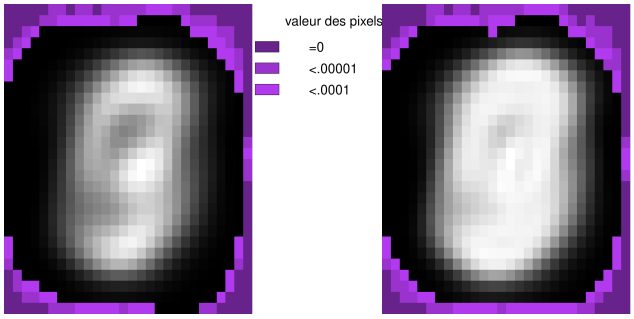

Figure 1: Moyenne empirique (gauche) et variance empirique (droite) des pixels sur l'ensemble des données

Ce point est intéressant en ce qu'il permet de potentiellement réduire la dimension du problème (si on arrive à trouver les pixels non significatifs on peut ne pas les prendres en compte).

[2]On parlera souvent de performance d'un algorithme, par abus de langage, pour parler de la performance du classifieur obtenu via la mise en oeuvre de l'algorithme.

Notons que lorsque la perte considérée est la perte charnière (ce qui est le cas pour tout les algorithmes considérés sauf EKF) la solution x suggérée suite à l'optimisation de la perte peut être interprêtée comme un vecteur de significativité des pixels.
De façon informelle, pour $j \in [\![1, 784]\!]$:

- Plus une valeur x_j est positive et grande plus ce pixel est "censé" être actif si le chiffre est un 0.
- Plus une valeur x_j est négative et grande plus ce pixel est "censé" être inactive si le chiffre est un 0.
- Plus une valeur x_j est proche de 0 moins elle est significative (l'activation ou non du $j^{ième}$ pixel n'influt pas ou presque sur la classification du chiffre).

Au vue de ce fait, on va considérer qu'une solution x est d'autant plus bonne qu'elle présente une sparsité assez proche de la sparsité globale des données.

Aussi, pour les algorithmes se basant sur la perte charnière on pourra aussi juger (de manière informelle) de la qualité d'une solution en ce que la solution x présente un 'profil d'activation' "vraisemblable" pour un '0'.
Ce que l'on appelle 'profil d'activation' est la représentation de x comme une image 28×28 pixels où les luminosités des pixels représentent l'intensité de leur valeur et les couleurs représentent leur signe (bleu si positif et rouge si négatif).

On peut essayer de se faire une idée "d'à quoi pourrait ressembler" l'activation pour la classification d'un nombre en calculant la moyenne empirique (pixel par pixel) des a_i associés à ce chiffre à laquelle on soustrait la moyenne empirique (pixel par pixel) de tout les autres chiffres. On obtient, par exemple, les profils d'activation suivant :

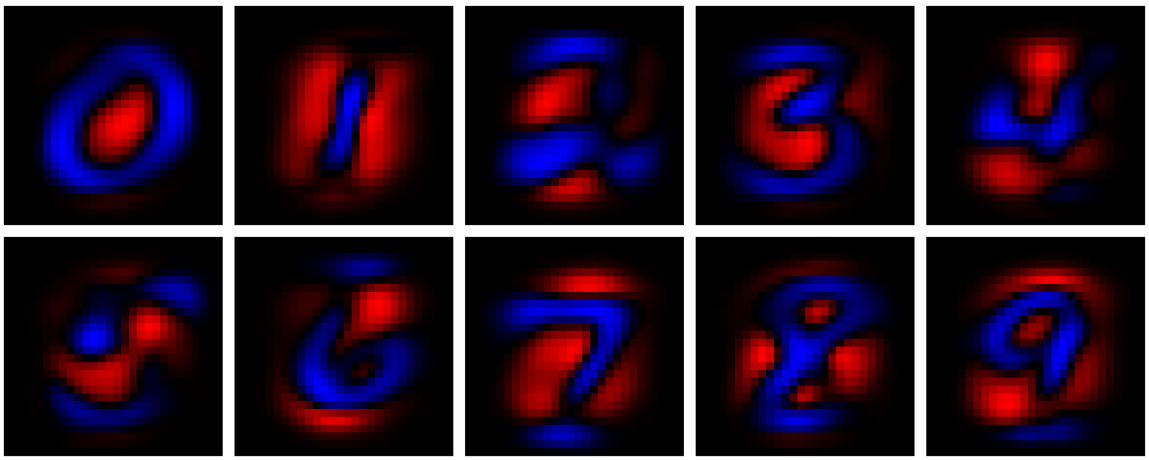

Figure 2: Exemples de profils d'activations pour chaque chiffre

Ces considérations seront l'objet de la section 4.

1.1.3 La sensibilité à l'initialisation (et aux hyperparamètres)

Un autre point essentiel, commun à de nombreux algorithmes, est celui de l'initialisation et des hyperparamètres. En effet, pour la plupart des algorithmes que nous allons voir[3] le choix de l'initialisation peut sembler arbitraire. Il paraît alors vital d'avoir des algorithmes dont le résultat ne dépend pas ou peu du point d'initialisation.
Nous allons donc mettre en oeuvre en section 5 un moyen de comparer (de façon non exhaustive...) la sensibilité des différents algorithmes à l'initialisation.

Pour ce qui est de la sensibilité aux hyperparamètres, elle ne fera pas l'objet d'une section à proprement parler, mais il en sera question dans la section 2.1 lors de la recherche des hyperparamètres "optimaux".

1.1.4 La robustesse aux attaques

Enfin, un dernier enjeu important que nous avons décidé de prendre en compte est celui de la "robustesse" aux attaques. En effet, pour des algorithmes qui remplissent des fonctions critiques pour des enjeux de sécurité il est essentiel d'être résistant à de potentielles "attaques". Heureusement, dans notre cas, la mauvaise classification d'un '0' ne causera pas (on l'espère) de catastrophe.
Il n'en reste pas moins intéressant d'essayer de discriminer les algorithmes les plus résistants des moins résistants.

Pour ce faire, une première idée que nous avons décidé de mettre en oeuvre dans la section 6 est celle de l'empoisonnement des données. Un empoisonnement plus réfléchi et/ou d'autres méthodes plus pointilleuses sont disponibles dans : Biggio et al. (2013) qui mets en oeuvre une approche par ascension de gradient pour trouver des données "optimales pour leurrer une SVM"; ou dans le très bon site d'introduction au Machine Learning adversariel !

[3] À l'exception de SEGpm, SREGpm et SBEGpm

2 Les hyperparamètres

Nous nous proposons dans cette section d'étudier les hyperparamètres (i.e. les paramètres qui ne sont pas optimisés durant l'entraînement) de chacun des algorithmes. Dans un premier temps, nous cherchons à déterminer empiriquement des hyperparamètres optimaux en procédant par une recherche en grille par validation croisée (grid search cross validation). Nous étudierons dans un second temps l'impact des hyperparamètres sur les résultats obtenus.

2.1 Sélection des hyperparamètres

Un algorithme d'optimisation vient souvent avec un certain nombre d'hyperparamètres pouvant être dûs à l'ajout d'une fonction de régularisation à la perte utilisée (hyperparamètre λ pour la régularisation l_2 ou la taille z de la boule pour les algorithmes utilisant une projection dans l_1) ou bien venir de l'algorithme en lui même (poids β_1 et β_2 associés au 'momentum' pour l'algorithme Adam). Il n'existe pas de manière explicite de calculer les valeurs optimales de ces hyperparamètres et nous utilisons donc une méthode empirique, la recherche par grille, pour trouver les valeurs optimales sur un ensemble de valeurs finies en fonction d'un critère d'optimalité objectif.

Definition 2.1 (Recherche par grille)**.** *Soit $\Theta \subset \mathbb{R}^n$ l'ensemble des hyperparamètres et $\{\mathcal{G}_{\Theta_i}\}_{i=1}^n$ l'ensemble des grilles finies d'hyperparamètres. On définit par $g(\theta; \mathcal{D}_u, \mathcal{D}_v)$ la fonction qui associe à un jeu de paramètre θ le critère d'optimalité sur le jeu $\mathcal{D}_v$ de l'algorithme g après T itérations d'entraînement sur le jeu $\mathcal{D}_u$. Les paramètres $\hat{\theta}$ sélectionnés par la recherche par grille sont alors définis par,*

$$\hat{\theta} = \text{argmin}_{\theta \in \{\mathcal{G}_{\Theta_i}\}_{i=1}^n} CV_K(\theta)$$

où la fonction CV_K est la fonction de moyenne par validation croisée sur K blocs définis par

$$CV_K(\theta) = \frac{1}{K} \sum_{u=1}^{K} g(\theta; \bar{\mathcal{D}_u}, \mathcal{D}_u)$$

où $\mathcal{D}_u$ est une partition du jeu d'entraînement $\mathcal{D}_{train}$ de taille $\lfloor \frac{|\mathcal{D}_{train}|}{K} \rfloor$ et $\bar{\mathcal{D}_u}$ est son complémentaire dans $\mathcal{D}_{train}$.

Par la suite, nous utilisons pour la sélection des hyperparamètres le taux d'erreur comme critère d'optimalité car nous cherchons les paramètres permettant d'obtenir la meilleure précision sur le jeu de test. Nous notons toutefois qu'il était envisageable d'utiliser d'autres critères d'optimalité comme la fonction de perte avec par exemple une régularisation l_1 qui tiendrait ainsi compte de la sparsité des solutions obtenues.

Le taille de la grille de recherche évoluant exponentiellement avec le nombre d'hyperparamètres nous sommes ici limités par la puissance de calcul à notre disposition et nous utiliserons donc des grilles de petite taille (moins de 30 points) bien que nous sommes conscients que les hyperparamètres obtenus pourront de fait être éloignés de leurs valeurs optimales. Par ailleurs nous avons pu remarquer que certains hyperparamètres pouvaient rendre les algorithme plus ou moins sensible à l'initialisation. Nous avons ici effectué la

Optimiseur	paramètres
GD	$\lambda = \{1/6, 1/4, 1/3, 1/2\}, z = \{1, 10, 20, 50, 100\}$
SGD	$\lambda = \{1/5, 1/4, 1/3, 1/2, 1\}, z = \{10, 50, 100, 200, 500\}$
SMDproj	$z = \{10, 50, 100, 200, 500\}$
SEGpm	$z = \{10, 50, 100, 200, 500\}$
Adaproj	$z = \{10, 50, 100, 200, 500\}$
Adamproj	$\beta_1 = \{0.5, 0.9, 0.99\}, \beta_2 = \{0.99, 0.999, 0.9999\}, z = \{50, 100, 200\}$
ONS	$\lambda = \{1/8, 1/4, 1/2\}, z = \{50, 100, 200\}, \gamma = \{1/16, 1/8, 1/4\}$
SREGpm	$z = \{10, 50, 100, 200, 500\}$

Table 2: Grille de recherche pour chacun des algorithmes

Optimiseur	λ	z	γ	β_1	β_2
GD	0.25	50	–	–	–
SGD	0.2	50	–	–	–
SMDproj	–	200	–	–	–
SEGpm					
Adaproj	–	200	–	–	–
Adamproj	–	200	–	0.99	0.999
ONS	0.25	200	0.25	–	–
SREGpm	–	–	–	–	–

Table 3: Hyperparamètres sélectionnés par recherche par grille avec validation croisée

recherche par grille avec des poids initialisés à 0 mais une autre initialisation aurait pu nous donner des résultats différents. Nous reparlerons de ces considérations dans la section 5.

En utilisant les grilles définies par Tab[2.1] avec $K = 3$ partitions pour la validation croisée et pour 100 itérations pour l'algorithme de descente de gradient, 1000 pour ONS et 10000 itérations pour les autres algorithmes nous obtenons les hyperparamètres définis par Tab[3].

Nous pouvons voir sur la figure Fig[3](droite) que la fonction qui associe le taux d'erreur à un jeu d'hyperparamètres semble particulièrement régulière (proche d'une fonction linéaire) pour l'algorithme de Descente de Gradient Stochastique. Cela semble en revanche moins le cas pour l'algorithme Online Newton Step (ONS) (voir Fig[3](gauche)) où la fonction qui associe le taux d'erreur à un jeu de paramètre semble moins régulière. Les résultats obtenus par grille de recherche sont dans ce cas moins fiables car une faible variation de paramètre pourrait modifier significativement le taux d'erreur.

2.2 Rôle des hyperparamètres : Le cas de l'algorithme GD

On peut voir sur la figure Fig[4] que la sparsité des solutions est positivement corrélée à λ et négativement corrélée à z. Ces résultats sont cohérents puisque le paramètre λ est un paramètre de régularisation l_2 et qu'il va donc shrinker les solutions à mesure qu'il

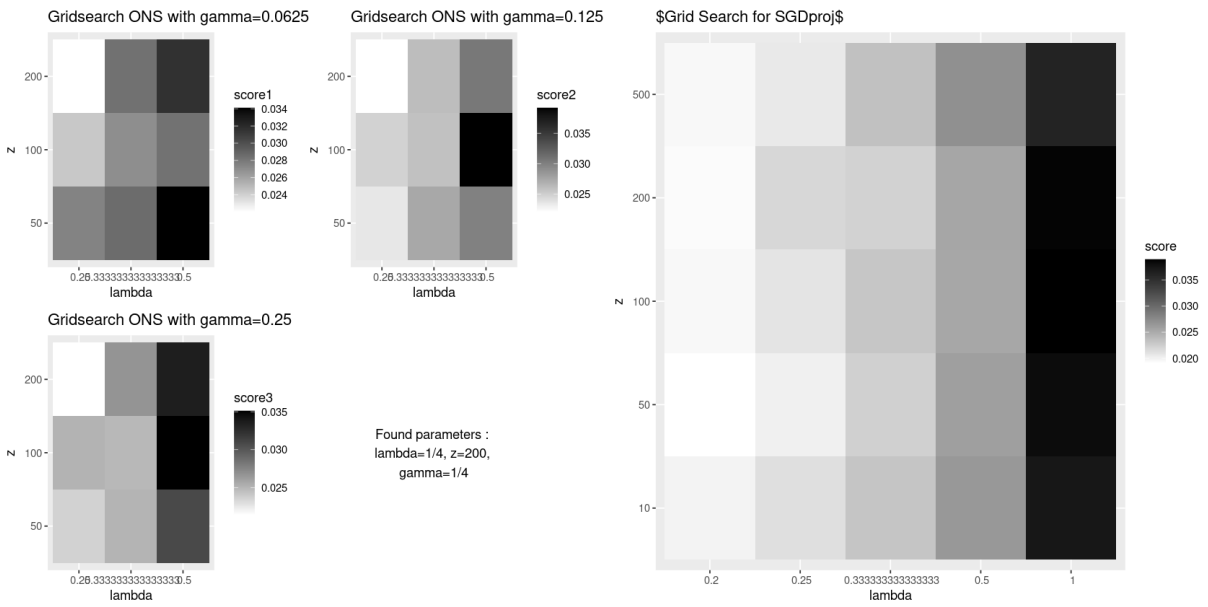

Figure 3: Recherche par grille avec validation croisée (K=3) pour l'algorithme Adam (gauche) et SGD (droite)

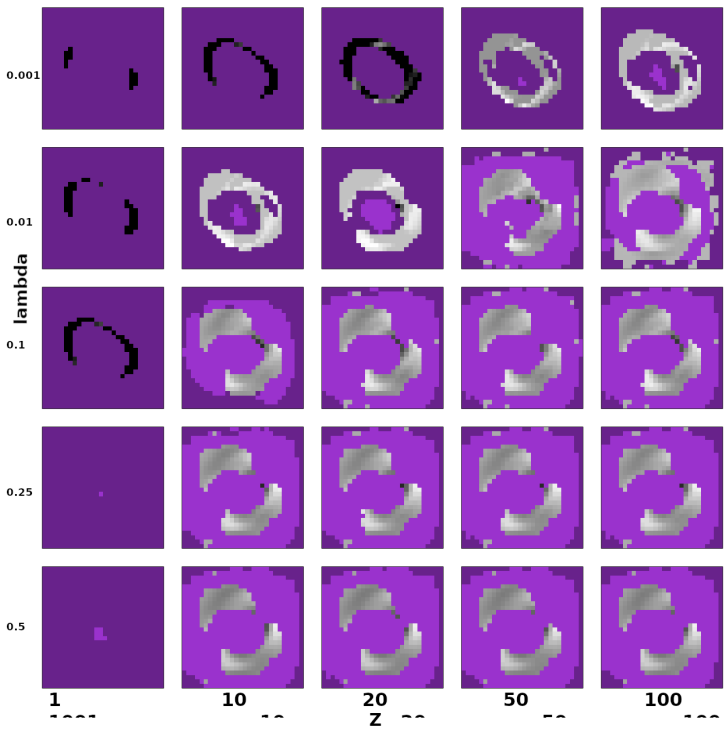

Figure 4: Sparsité de GD en fonction de z et lambda

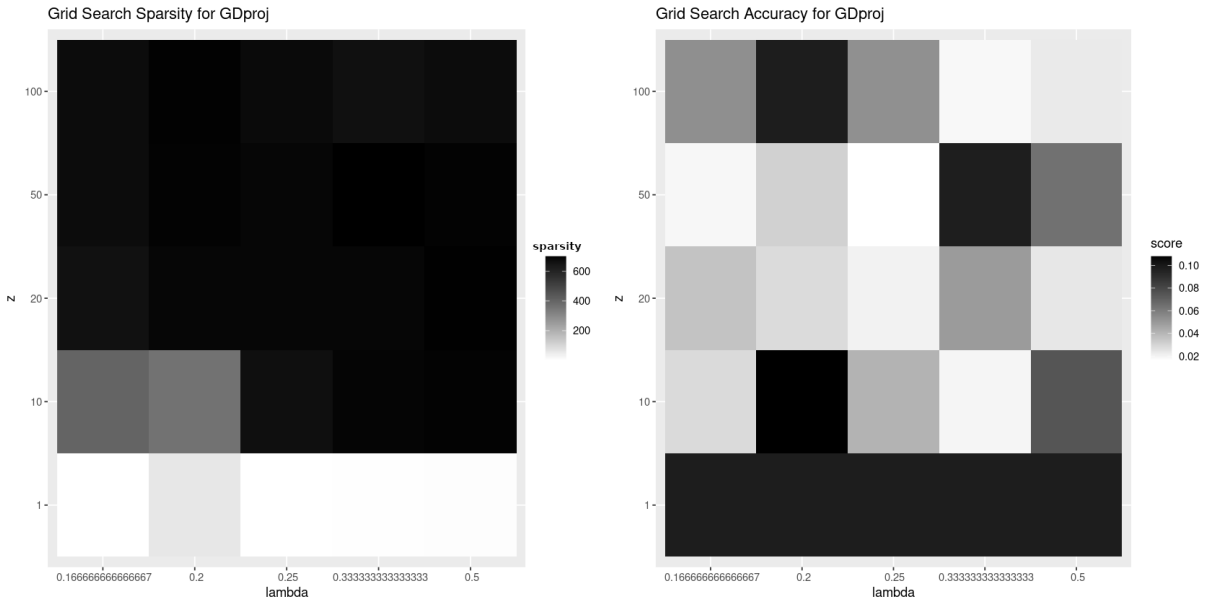

Figure 5: Evolution du score et de la sparisté en fonction de z et λ pour descente de gradient

augmente, de même qu'à mesure que le rayon z de la boule l_1 diminue, la projection sur cet ensemble à plus de chance de tomber sur un coin de la boule et donc d'être sparse.

Par ailleurs, il semblerait (voir Fig[5]) qu'il soit difficile de concilier des hyperparamètres qui entraînent des solutions à la fois sparses et performantes en ce qui concerne la descente de gradient. On voit en effet que si le rayon de la boule l_1 utilisée pour la projection est trop petit (inférieur à 20) la précision de la descente de gradient diminue significativement. Par ailleurs des valeurs de z trop grandes ne permettent pas d'obtenir des solutions sparses.

Globalement il semble donc difficile d'obtenir des solutions ayant un bon score de précision et qui soient en même temps sparses. Nous verrons dans la section 7 comment l'auteur de Xiao (2009) propose une méthode qui permet d'obtenir des solutions qui concilient les deux.

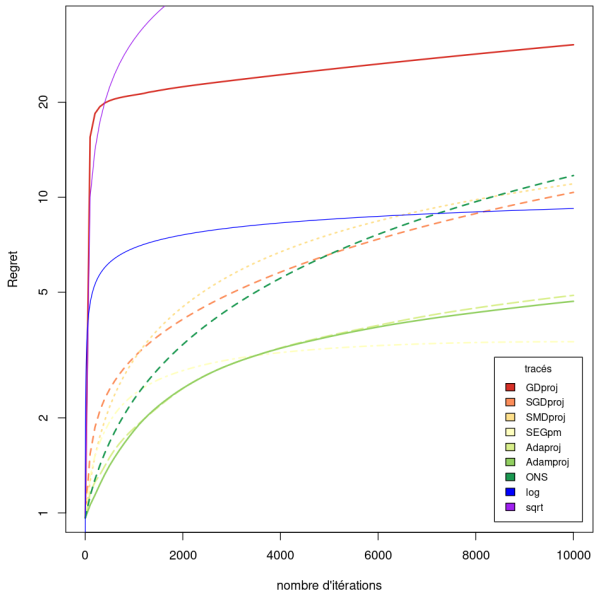

Figure 6: Estimation du regret pour l'ensemble des méthodes

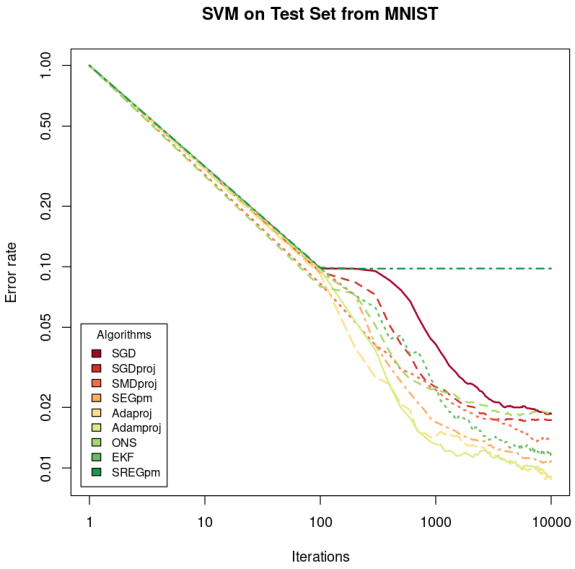

Figure 7: Taux d'erreur en fonction du nombre d'itérations. Attention les méthode GD et GDproj ne sont entraîné que sur 100 itérations quand les méthodes stochastique sont entraînés sur 10000 itérations.

3 Précision et temps de calcul

Comme cela a été expliqué précédemment, notre critère principal pour évaluer la performance d'un algorithme est la précision (ou accuracy) (voir Tab[4]) mais nous sommes également intéressés par la complexité algorithmique de ceux-ci et comme nous l'avons expliqué en section 1.1.1, le temps de calcul en est une mesure intéressante.
Nous estimons qu'il est pertinent de comparer les performances et le temps de calcul des algorithmes une fois que ceux-ci ont convergé. Afin d'avoir un critère objectif de la convergence des algorithmes nous fixons un seuil ϵ tel que l'algorithme s'arrête de tourner lorsque que la différence absolue entre la performance à l'itération t et celle à l'itération $t-s$ est inférieure à ϵ (d'autres critères d'arrêt sont biensur possibles). Par ailleurs nous avons pu observer Fig[7] que certains algorithmes (Descente de Gradient et Descente de Gradient Stochastique) montrent un plateaux dans leur performance avant de continuer à s'améliorer, nous nous intéressons donc uniquement aux performances une fois ce plateau passé.

Le tableau 4 résume l'ensemble des performances obtenues pour chacun des algorithmes étudiés avec les hyperparamètres trouvés dans 3.

On remarque tout d'abord que les méthodes stochastiques convergent bien plus vite que la descente de gradient classique ce qui est cohérent car la complexité d'une itération est bien moins importante pour les méthode stochastiques car on calcul un seul sous-gradient à la fois par itération. On voit que malgré cela les méthodes stochastique sont plus performantes (en particulier les algorithmes Adamproj et Adaproj) que la descente de gradient classique.

On notera également que les méthode de type Follow the Leader régularisé (SMDproj,

Optimiseur	temps avant convergence (s)	itérations	Performance
GDproj	581.995	400	0.9799
SGDproj	2.929	3600	0.9825
SMDproj	3.0408	3200	0.9824
SEGpm	4.190	4400	0.988
Adaproj	3.885	4400	0.9891
Adamproj	2.472	2400	0.9892
ONS	28.128	2200	0.9788
EKF	27.671	2800	0.9857

Table 4: Performances, temps de convergence et nombre d'itération avant convergence pour les différents algorithmes avec $\epsilon = 0.001$ et $s = 200$. [5]

SEGpm et Adaproj) sont plus performantes que la descente de gradient stochastique et ce principalement entre les itérations 100 à 1000 où la vitesse de convergence de SGD diminue significativement quand l'utilisation d'une régularisation (divergence de Bergmann pour SMDproj, entropie négative pour SEGpm ou une régularisation adaptative dans le cas de Adaproj) permet d'éviter cela (voir Fig[7]); ce qui accélère donc la convergence et permet d'obtenir de meilleurs performances globales.

Aussi, malgré qu'il ait de bonnes propriétés théoriques (en particulier une borne de regret en $O(\log(T))$), l'algorithme ONS montre de moins bonnes performances que les autres méthodes. Il est tout d'abord plus lent que les méthodes stochastiques du fait de l'inversion de matrice qui est effectué à chaque itération. Par ailleurs, il semble converger vers un score de précision moins intéressant que l'ensemble des autres méthodes stochastiques.

Estimation du regret

Pour ce qui est des bornes de regrets associées aux différents algorithmes traités, il va nous falloir les estimer. Pour cela, on va exécuter les algorithmes plusieurs fois avec un paramètre z commun.
En effet, n'ayant pas accès à $x^* := \underset{x\in\mathcal{K}}{min} f(x)$[6], on va l'estimer par le x qui atteint la meilleure performance parmis toutes les expériences de tous les algorithmes que l'on réalise. On obtient alors la figure [6].

Nous pouvons voir sur la figure [6] que les méthodes SMDproj et Adaproj montrent des regrets estimés qui semblent cohérents avec les bornes de regret étudiées dans le cours Wintenberger (2021). En effet ces deux méthodes présentent un regret de l'ordre de $O(\sqrt{T})$ ce qui est cohérent avec les estimations des regrets que nous avons pu calculer. En revanche notre estimation du regret des algorithmes SGDproj et ONS ne concorde pas avec les résultats du cours puisque cet algorithme a un regret de l'ordre de $O(\log(T))$ ce qui n'est pas le cas de notre estimation qui semble être de l'ordre de $\sqrt{T}$.

[6] où $\mathcal{K}$ est l'ensemble sur lequel a lieu l'optimisation donc ici une boule l_1 de rayon $z = 100$

4 Sparsité et forme des solutions

Le second critère de comparaison retenu pour notre étude est la sparsité. On aimerait que l'algorithme "détecte" de lui même le fait que beaucoup de pixels sont non significatifs pour la tâche de classification.
Biensûr, l'agorithme qui retourne toujours le vecteur nul a une sparsité maximale, il est donc important de garder en tête qu'on veut une sparsité grande mais des taux d'erreurs faibles.

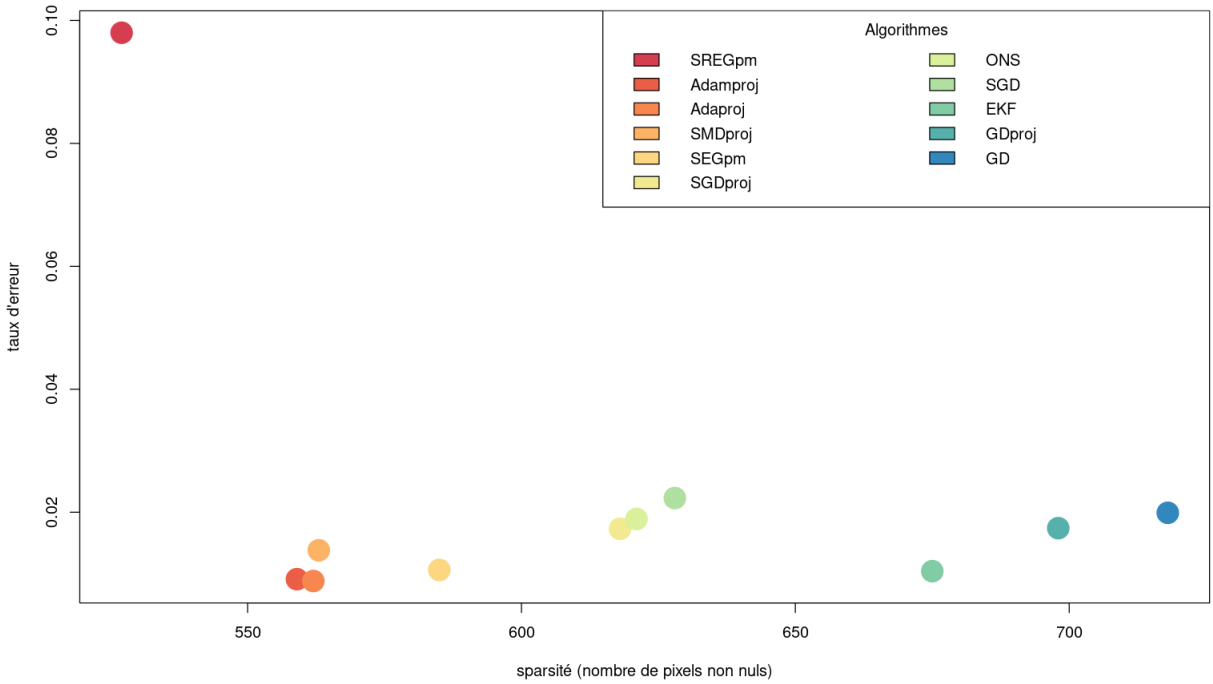

Figure 8: Taux d'erreur en fonction de la sparsité pour les méthodes étudiées (10000 itérations, sauf pour SREGpm 100000, avec les hyperparamètres de la section 2.1)

Le graphique [8] précédent nous permet de discriminer, à l'oeil, les algorithmes qui réalisent un bon compromis sparsité/taux d'erreur[7] :

1. En 'tête de file' on a Adamproj, Adaproj et SMDproj (un peu moins performant) qui réalisent un bon compris, suivis d'assez près par SEGpm.
2. Un deuxième regroupement que l'on peut faire est celui de SGDproj, ONS et SGD.
3. Puis en 'queue de peloton' on a EKF, GDproj et GD qui présentent des sparsités faibles (et des taux d'erreurs faibles à l'exception d'EKF).

- SREGpm présente quant à lui un problème car il ne dépasse pas vraiment la performance d'un classifieur aléatoire (~ 0.1 des données sont des '0').

[7] Compromis dont il a déjà été question pour la méthode GD dans la section 2.2

Le récapitulatif du graphe [8] avec les valeurs exacte est le suivant :

	nombre de coordonnées non nulles	taux d'erreur
paramSREGpm	527	0.098
paramAdamproj	559	0.0091
paramAdaproj	562	0.0088
paramSMDproj	563	0.0138
paramSEGpm	585	0.0106
paramSGDproj	618	0.0173
paramONS	621	0.0189
paramSGD	628	0.0223
paramEKF	675	0.0104
paramGDproj	698	0.0174
paramGD	718	0.0199

Table 5: Récapitulatif du taux d'erreur et de la sparsité obtenue pour 10000 itérations, sauf pour SREGpm 100000 (sous les hypérparamètres trouvés en section 2.1)

Les sparsités atteintes ne sont pas encore totalement convaincantes car en calculant les moyennes et variances empiriques des pixels sur l'ensemble des données on trouve que 369 des pixels ont à la fois des variances et des moyennes empiriques en dessous de 0.2 . Cette valeur de 0.2 est, à nos yeux, la limite de ce que l'on peut personnellement discerner du noir profond sur les images [8]. On aimerait donc obtenir des sparsités plus proches de $784 - 369 = 458$ que ce que l'on a obtenu.
L'article Xiao (2009) dont il est question en section 7, essaie justement de répondre à cette attente (voir section 7.1).

En ce qui concerne la forme des solutions, elle ne permet pas de bien discriminer selon l'accomplissement du compromis sparsité/taux d'erreur de façon fiable. Mais, pour les curieux, la page suivante (voir figure [9]) présente les profils d'activations finaux pour différentes méthodes (après 10000 itérations) ainsi que les sparsités de ses profils.

[8] et pour ce qui est de la variance de 0.2 nous avons estimé que c'était suffisamment faible

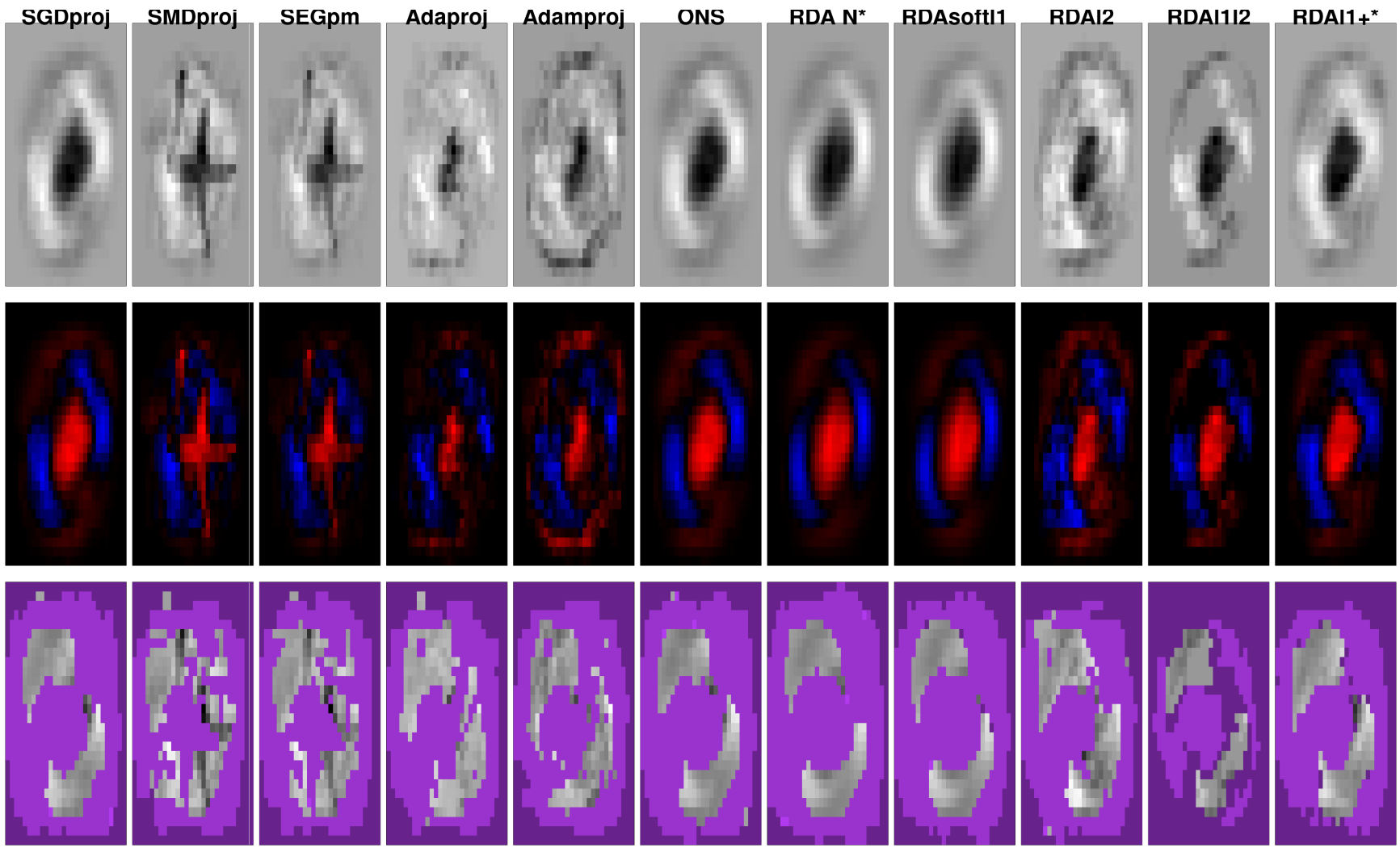

Figure 9: Formes des solutions pour différents algorithmes
(en niveau de gris, en rouge/bleu pour négatif/positif et enfin avec le code couleur de la figure [1])

5 Sensibilité à l'initialisation (aux hyperparamètres)

Comme expliqué à la section 1.1.3, nous allons chercher à comparer la sensibilité à l'initialisation des différents algorithmes (mis à part SEGpm, SREGpm, SBEGpm où l'initialisation est fixée, ils ne sont donc pas concernés).

Pour cela, la procédure que nous allons suivre est la suivante, pour chacun des algorithmes : On se donne un ensemble d'initialisations possibles $(x_0^{(i)})_{i\in I} \in \mathbb{R}^{785}$ et on observe dans un premier temps l'évolution du taux d'erreurs de classification de l'algorithme en fonction du nombre d'itérations.
On illustrera dans un second temps la capacité d'un algorithme à 'se défaire' de l'initialisation en un certain nombre d'itérations avec l'évolution des poids x pour quelques algorithmes.

Commençons alors par présenter les 13 initialisations possibles considérées[9] :

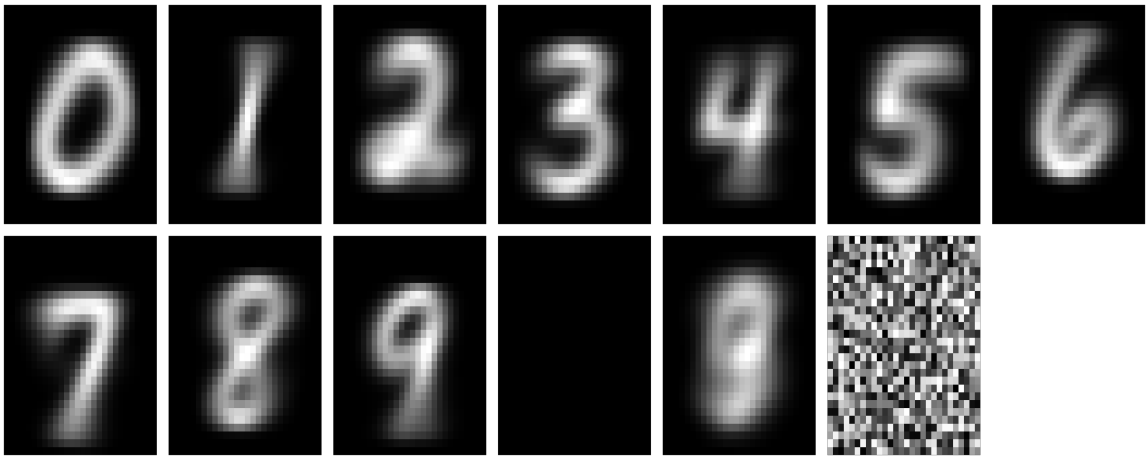

Figure 10: Les 13 initialisations utilisées : De 0 à 9 il s'agit des moyennes sur les données d'entraînement pour chaque chiffres, puis d'une initialisation au vecteur nul, d'une initialisation à la moyenne de tout les chiffres qui ne sont pas des 0 et enfin une initialisation aléatoire (uniforme sur chaque pixel)

5.1 Le taux d'erreur par rapport à l'initialisation

Lors de notre étude expérimentale nous avons décidé de regrouper les algorithmes en trois familles.

La première regroupe les algorithmes : SGDproj, SMDproj et Adaproj.
Ces algorithmes semblent assez vite (en moins de 100 itérations) ne plus trop dépendre de l'initialisation. On donne à titre d'exemple deux réalisations de trajectoires (Taux d'erreur par rapport au nombre d'itération) pour deux de ces méthodes dans la figure [11] suivante.

[9]Des initialisations en des éléments même de la base de donnée ont aussi été considérées mais nous en donnerons seulement quelques exemples dans la section 5.3

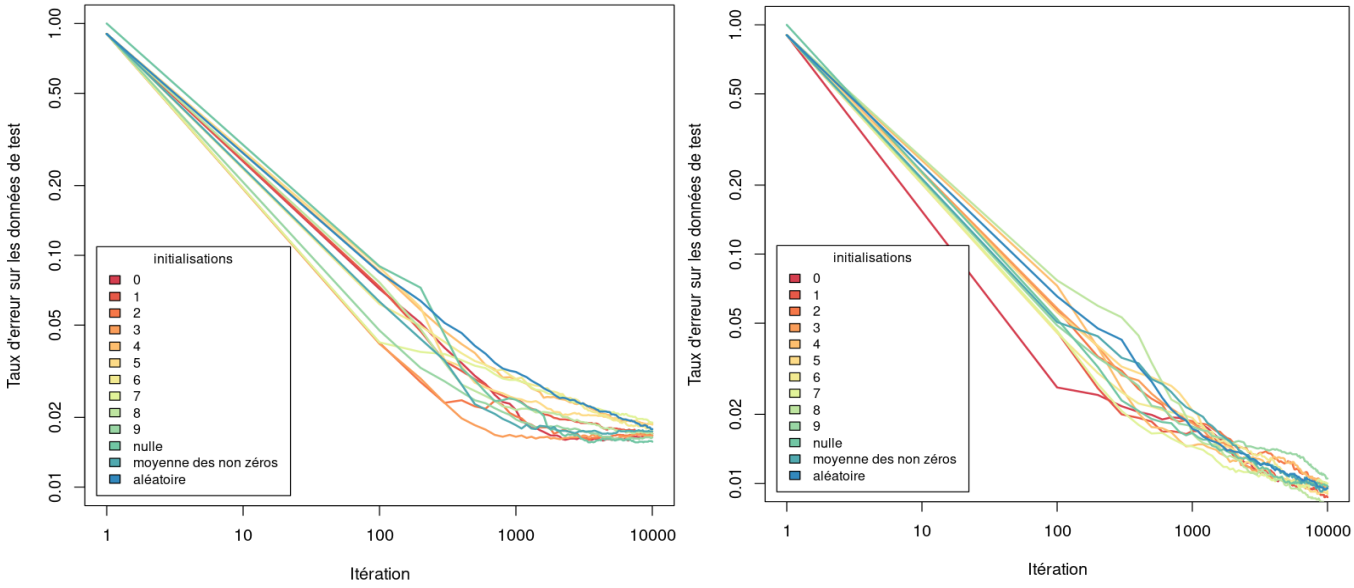

Figure 11: Taux d'erreur sur les données de test en fonction du nombre d'itérations pour les différentes initialisations (à gauche SGDproj à droite Adaproj)

La seconde famille regroupe les algorithmes : Adamproj et EKF (voir figure [12]).
Ces algorithmes semblent légèrement plus sensibles à l'initialisation.
On peut par exemple voir que pour Adamproj les premières itérations (< 100) semblent être très différenciées, mais au final les trajectoires se rejoignent avec le nombre d'itération.
Pour EKF par ailleurs, la situation est légèrement différente car les trajectoires semblent moins différenciées que pour Adamproj au départ, mais elles tardent à se rejoindre (on voit que l'initialisation nulle est toujours largement meilleure au court des 10000 itérations).

Enfin, une dernière catégorie est destinée à un algorithme en particulier : ONS.
Ce dernier semble extrêmement sensible à l'initialisation. Il est si sensible à l'initialisation que même l'optimisation des hyperparamètres que nous avons effectué en section 2.1 est en fait propre à l'initialisation, en le vecteur nul, choisie.
En effet, comme on peut le voir dans la figure [13] suivante, pour deux groupes d'hyperparamètres différents les trajectoires sont très différentes.

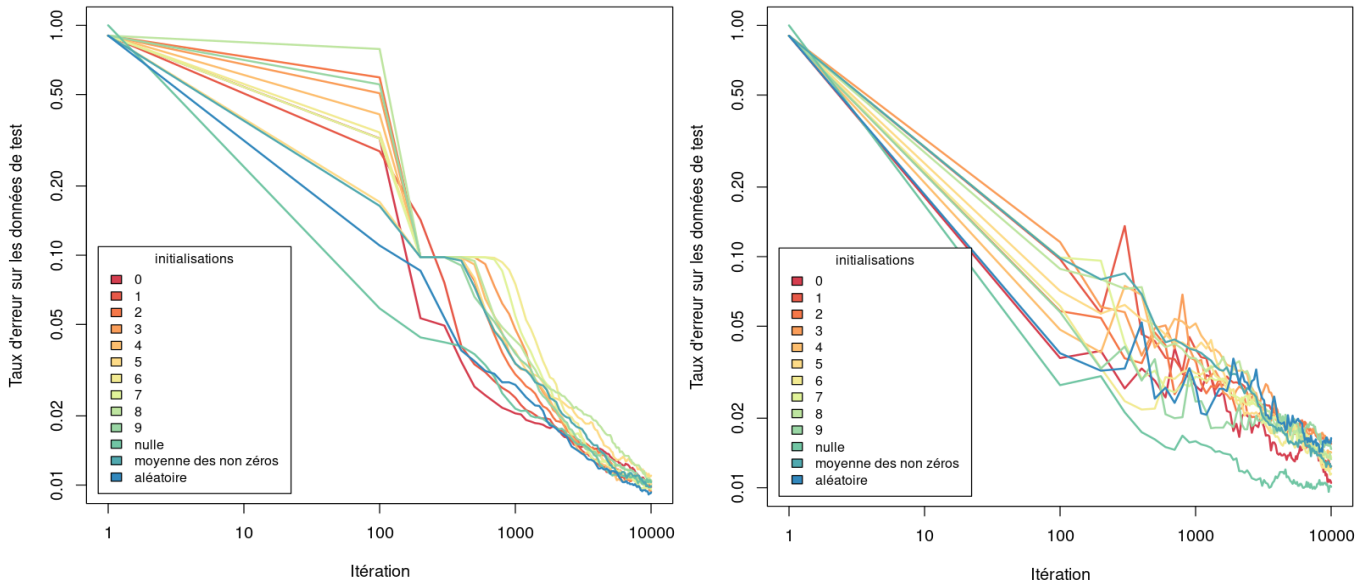

Figure 12: Taux d'erreur sur les données de test en fonction du nombre d'itérations pour les différentes initialisations (à gauche Adamproj à droite EKF)

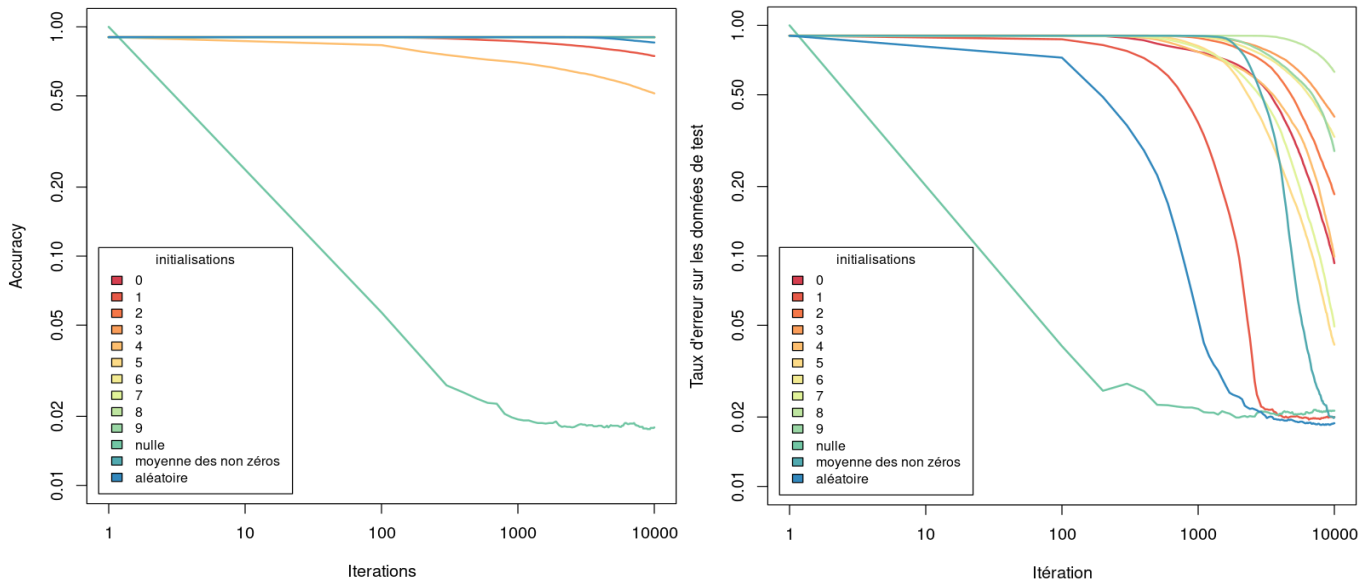

Figure 13: Taux d'erreur sur les données de test en fonction du nombre d'itérations pour les différentes initialisations (à gauche pour nos hyperparamètres, à droite pour $\lambda = \frac{1}{3}$ et $\gamma = \frac{1}{8}$, $z = 100$)

5.2 sensibilité au pas de descente : Comparaison de SGDproj et SEGpm

Le pas d'un algorithme d'optimisation en ligne peut s'avérer particulièrement important. Bien que nous choisissons de manière générale des pas de descente d'un ordre qui nous permet d'atteindre des bornes de regret optimale, il peut être intéressant de voir si un

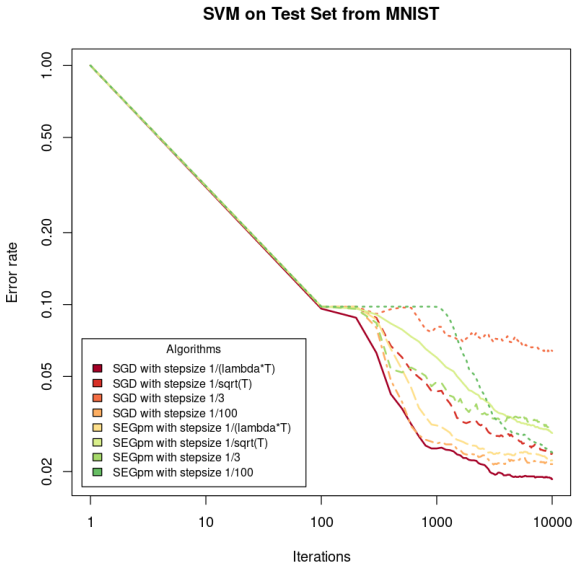

Figure 14: Sensibilité au pas de descente pour SEGpm et SGDproj

algorithme est robuste à la modification du pas de descente.

Comme nous pouvons le voir sur la figure [14] le choix du pas de descente est d'une grande importance en ce qui concerne l'algorithme SGDproj. On remarque en effet qu'un pas constant trop grand ralentit significativement la convergence de l'algorithme. En revanche il semble que l'algorithme SEGpm soit plus robuste à la modification du pas de descente. Même dans le cas d'un pas constant relativement grand l'algorithme converge vite après avoir passé un plateau.

5.3 Illustrations de la dépendance à l'initialisation

Comme nous avons pu le voir dans ce qui précède, un des algorithmes les plus sensible à l'initialisation est ONS. C'est donc sur cet algorithme que nous allons focaliser l'illustration de la dépendance à l'initialisation.
Pour rendre compte du fait qu'ONS peut avoir du mal à se défaire de certaines initialisations, on comparera des initialisations[10] aux profils d'activation finaux (en niveau de gris cette fois-ci) correspondants. À titre comparatif, on fera de même pour la méthode SGDproj, qui comme on l'a vue semble assez "indépendante" de l'initialisation.

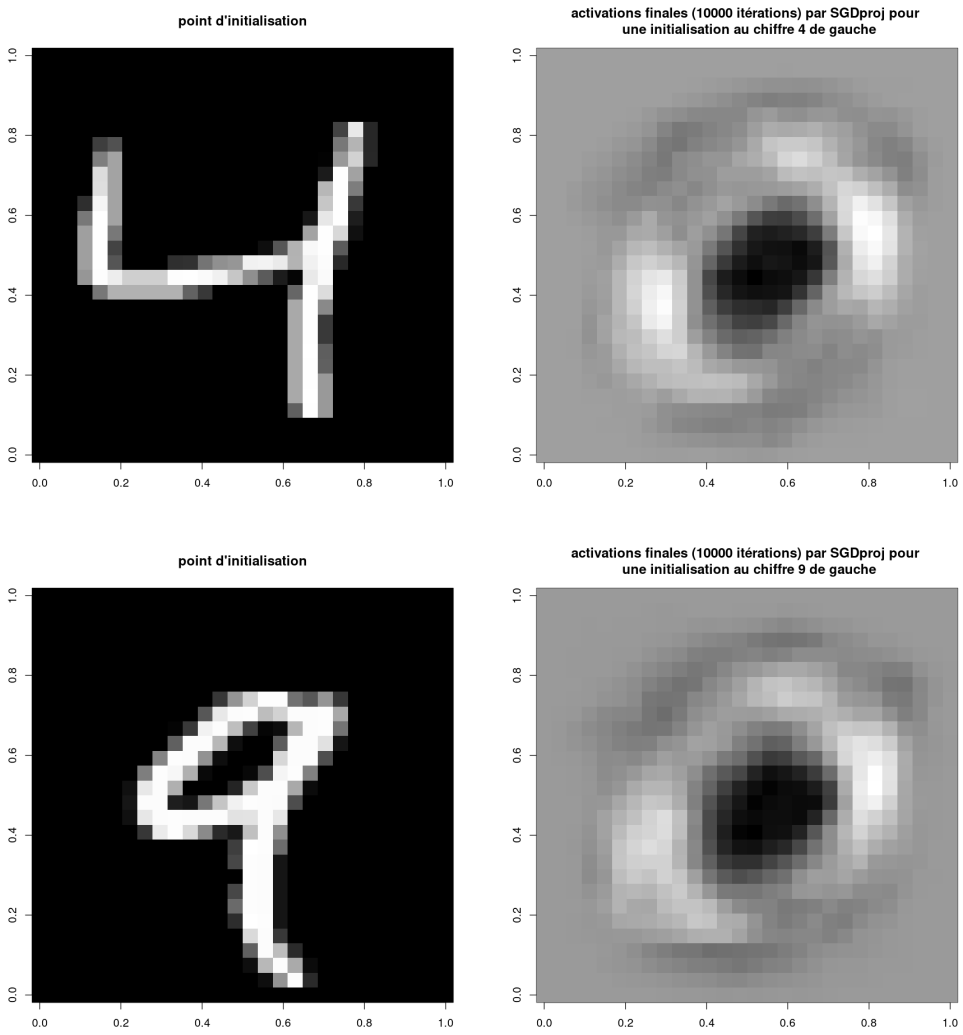

Figure 15: Exemples d'initialisations et de résultats finaux pour les activations pour la méthode SGDproj

On voit donc bien qu'avec SGDproj on obtient un résultat visuellement semblable pour les deux initialisations différentes.

[10]Notons qu'ici nous n'utiliserons pas les mêmes initialisations que dans la figure 10, pour avoir des résultats plus parlants

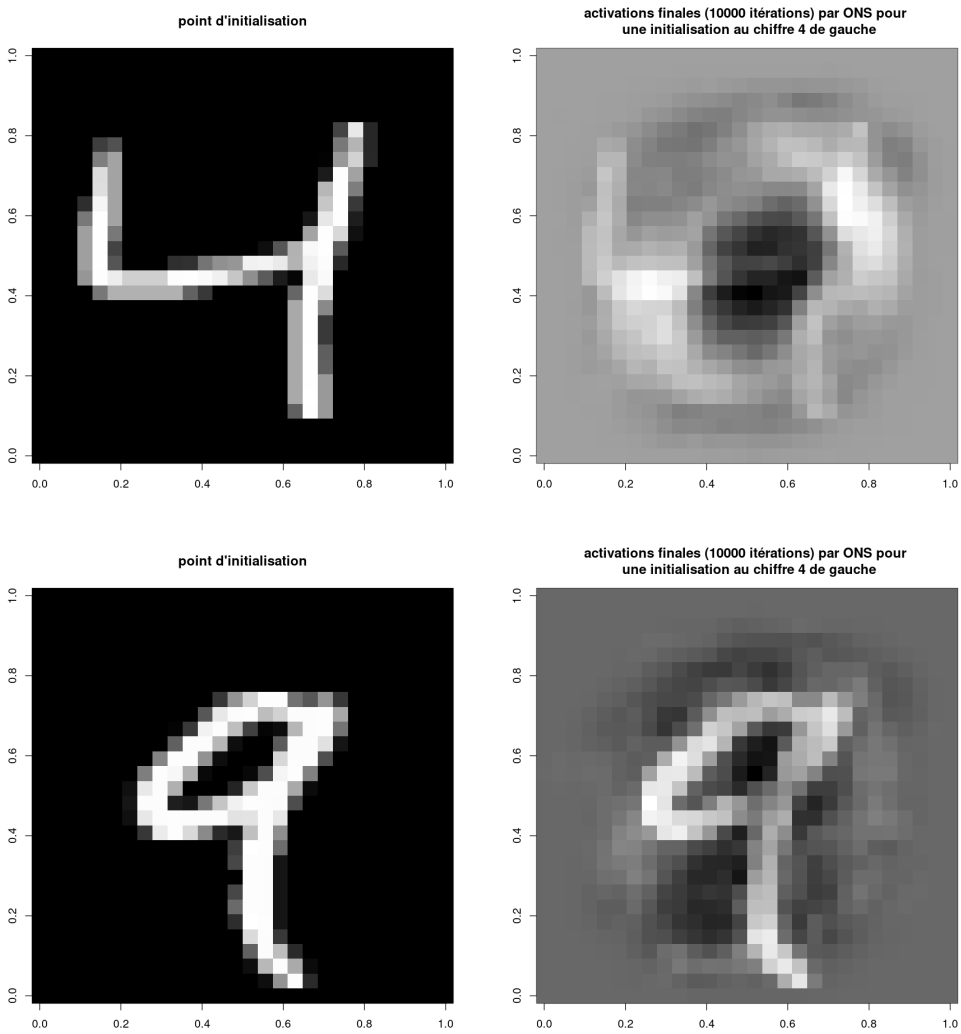

Figure 16: Exemples d'initialisations et de résultats finaux pour les activations pour la méthode ONS

ONS quant à lui semble mis à mal par ces initialisations (bon... elles sont volontairement trompeuses...).
Pour l'initialisation à '4' on voit qu'il se rapproche un petit peu d'une solution qui semble correcte, malgré le fait qu'on observe toujours la trace de l'initialisation (même si on a fait 10000 itérations).
Pour l'initialisation à '9' par ailleurs, le résultat est très décevant... On voit qu'ONS a "compris" que les pixels des bords sont peu significatifs (d'où la couleur grise neutre unie car proche de 0), mais pour le reste on peut encore clairement distinguer le '9' de l'initialisation...

6 Robustesse aux attaques

Dans cette section nous allons nous attaquer à la thématique introduite en section 1.1.4 qui sera celle de la résistance des algorithmes à l'empoisonnement des données.
Même si l'on ne traitera pas de ce sujet de manière formelle, la procédure que l'on a tenté de mettre en place cherche à imiter le cadre adversariel tel qu'il en est question dans le cours de Wintenberger (2021).

Notre démarche a été la suivante:

- On construit l'image d'un chiffre dont on se dit qu'il va induire le classifieur en erreur (voir figure [17]).
- On injecte cette image un certain nombre de fois (à des emplacements aléatoires) dans la base de donnée.
- On compare la performance des algorithmes sur les données d'entraînement standard et sur les données d'entraînement corrompues.
  (Et ce sur plusieurs expériences étant donné que les algorithmes enjeux sont aléatoires, puis on fait la moyenne des résultats.)

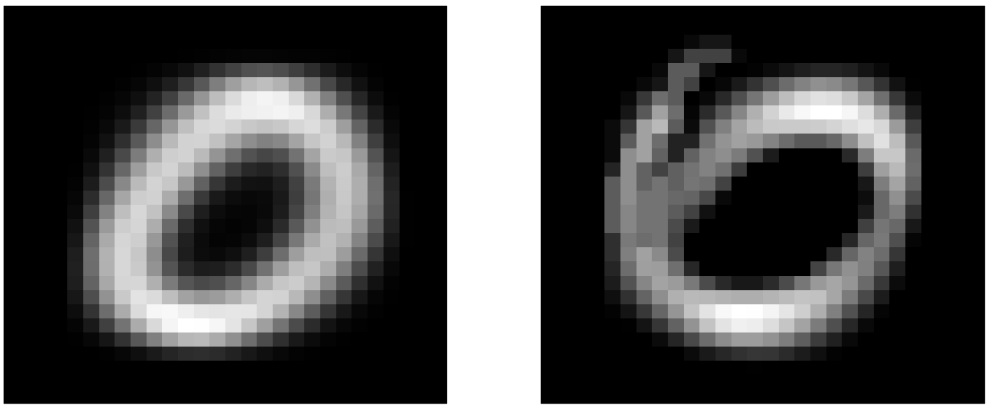

Figure 17: À gauche l'image moyenne des '0' des données MNIST, à droite un '6' fabriqué pour empoisonner les données

L'idée est de se dire que si beaucoup de 'petits malins' s'amusent à écrire des '6' qui ressemblent ("comme par hasard") à la moyenne des '0' cela peut compliquer la tâche d'apprentissage à nos algorithmes. Dans notre cas il n'y aura malheureusement pas plusieurs types de '6' (faute de temps) mais un seul.

Les résultats suite à la mise en oeuvre de la procédure d'empoisonnement progressif des données sont alors les suivants :

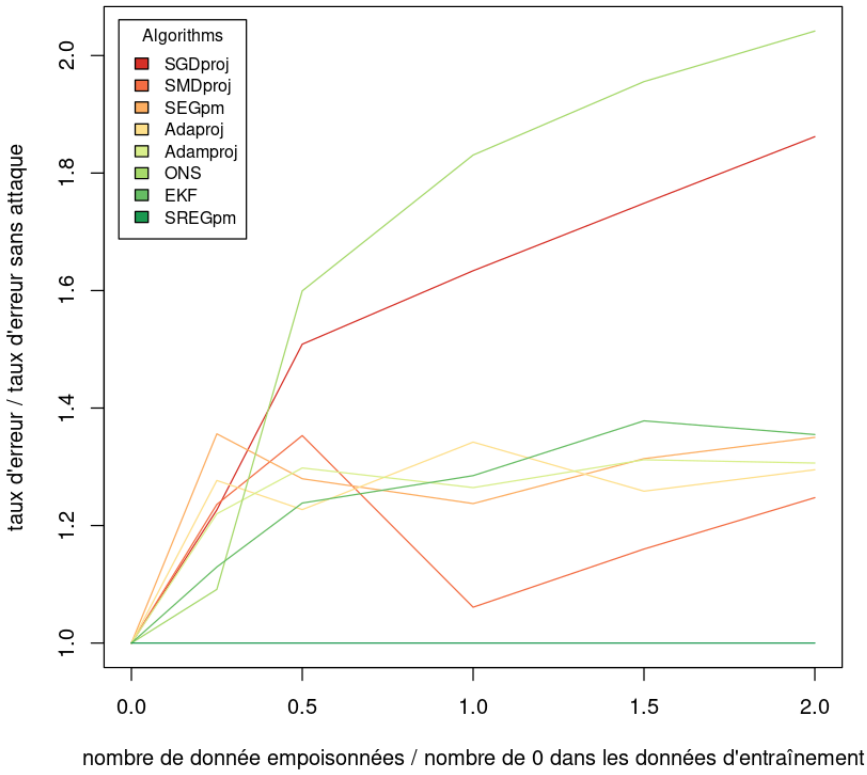

Figure 18: Détérioration du taux d'erreur en fonction du ratio du nombre de données empoisonnées par rapport au nombre de 0 (moyenne de 10 expériences de 10000 itérations pour chaque algorithme)

On a alors l'impression de pouvoir observer deux régimes principaux suite à cette expérience :

1. Des algorithmes dont la performance décroît de manière importante (une erreur 1.8 à 2 fois plus grande lorsque les données sont très corrompues) : ONS et SGDproj.

2. Des algorithmes pour lesquels on observe une détérioration mais qui semblent se stabiliser à partir d'un certain seuil [11] (facteur de détérioration en dessous de 1.4) : SMDproj, SEGpm, Adaproj, Adamproj, EKF.

- Pour ce qui est de SREGpm, il n'y a aucune détérioration mais nous n'avons pas réussi à identifier pourquoi. Le taux d'erreur est constant égal à 0.902 qui est à peu près le taux de non '0' dans les données...

Notons que ces graphes représentent des moyennes des résultats sur plusieurs expériences (les variances n'étant pas représentées car très faibles, de l'ordre de 10^{-6}).

Tentative non aboutie

Un autre point que nous avons voulu traiter concerne "la capacité d'exploration" des algorithmes. La discussion autour de la sensibilité à l'initialisation en section 5.1 traite

[11] Ce qui est probablement accentué par le fait qu'on empoisonne toujours avec la même donnée

déjà un peu de cette thématique mais l'idée que nous avons voulu mettre en oeuvre est la suivante :

- Modifier la base de données pour créer 'un minimum local' qui n'est pas global pour la perte considérée.
- Initialiser les algorithmes en ce minimum local et voir lesquels arrivent à en sortir.

Pour cela nous avons tenté de d'entraîner les algorithmes sur une nouvelle base de donnée où l'on a remplacé $\frac{1}{3}$ des '0' par le chiffre se trouvant dans la figure [19]. On initialise alors les algorithmes en le point illustré dans la figure [19]. On espère observer (au moins pour certains algorithmes) un taux d'erreur plus grands sur les premières itérations que pour le cas des données non modifiées[12].
Malheureusement les résultats observées n'ont pas été convaincants, c'est dommage car on pensait mettre à l'honneur les algorithmes dit "exploratoire" tels que SREGpm...

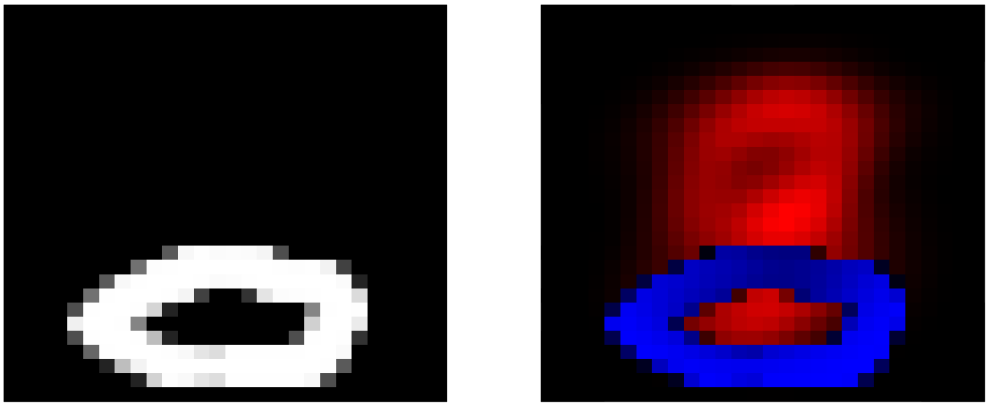

Figure 19: L'image utilisée pour modifier les données d'entraînement (à gauche), le point d'initialisation à (à droite)

[12]On pensait que certains algorithmes allaient stagner (au moins un moment) dans le minimum local que l'on a essayé de fabriquer.

7 Méthode de moyennisation duale régularisée

L'article Xiao (2009) propose une nouvelle classe d'algorithme en ligne, à savoir la moyennisation duale régularisé (regularized dual averaging ou RDA). L'intérêt de cette méthode est de mieux exploiter les méthodes de régularisation dans le cadre de l'optimisation en ligne.

7.1 Motivations

La méthode de descente de gradient stochastique est une des méthodes les plus populaire que ce soit dans la communauté de l'optimisation en ligne (ou même du machine learning de manière plus générale) du fait de ses bonnes performances pratiques mais également de la compréhension théorique qu'on en a. Malgré cela, l'algorithme SGD exploite peu la structure du problème d'optimisation et ce plus particulièrement pour le cas de problèmes régularisés qui sont de grande importance pour la résolution de problèmes de machine learning en grande dimension. Plus précisément, il est particulièrement difficile pour l'algorithme SGD (SGDproj) de renvoyer des solutions sparses comme nous avons pu le voir dans la section 4.
Enfin, bien que l'algorithme converge vers une solution optimale lorsque t tend vers l'infini, en pratique il est bien souvent stoppé avant cela ce qui entraîne une forme de variabilité dans le résultats obtenus (qui sont d'ailleurs accentué par l'aspect stochastique de cet algorithme).

La méthode de moyennisation duale régularisée se propose d'atténuer ces inconvénients avec un algorithme qui exploite la structure du problème de manière plus efficace en permettant l'utilisation de différentes fonctions de régularisation.

Plus particulièrement, à chaque étape de la méthode RDA est résolue

$$w_{t+1} = \text{argmin}_w \left\{ \frac{1}{t} \sum_{\tau=1}^{t} < g_\tau, w > +\psi(w) + \frac{\beta_t}{t} h(w) \right\} \tag{2}$$

où $g_t \in \partial f(w_t, z_t)$ est un sous-gradient de la fonction de perte f, ψ est une fonction de régularisation convexe et h est une fonction auxiliaire fortement convexe (par exemple $h(w) = \frac{1}{2}\|w\|_2^2$). Une version complète de l'algorithme est disponible (voir Algorithm[7.1]).

Cette méthode est en fait une extension de la méthode par moyennisation duale de Nesterov (voir Nesterov (2009)), qui est équivalente à choisir $\psi(w)$ comme la fonction indicatrice sur un ensemble fermé convexe. La forme de la mise à jour des poids est dans ce cas donné par l'équation (4).

7.2 Application de la RDA aux régularisations usuelles

Comme cela est décrit dans l'algorithme [7.1], la méthode RDA se découpe en 3 étapes dont la première est celle d'un algorithme d'optimisation en ligne stochastique classique (tel que SGD), à savoir le calcul d'un sous gradient g_t de $f(w_t)$.
La seconde, s'apparente en une sorte de 'momentum' en utilisant un meilleur estimateur

Algorithm 1 Méthode de moyennisation duale régularisée (RDA)

Require:

- Une fonction auxiliaire $h(w)$ fortement convexe sur $\text{dom}(\psi)$ satisfaisant

$$\text{argmin}_w\, h(w) \in \text{argmin}_w\, \psi(w) \tag{3}$$

- Une suite non-négative et non-décroissante $\{\beta_t\}_{t\geq 1}$

Ensure: $w_1 = \text{argmin}_w\, h(w)$ et $\bar{g}_0 = 0$

for $t = 1, 2, 3, \ldots$ **do**

1. Etant donné la fonction f_t, calculer le sous gradient $g_t \in \partial f_t(w_t)$
2. Mettre à jour le sous gradient moyen :

$$\bar{g}_t = \frac{t-1}{t}\bar{g}_{t-1} + \frac{1}{t}g_t$$

3. Calculer le prochain vecteur de poids

$$w_{t+1} = \text{argmin}_w \left\{ < \bar{g}_t, w > + \psi(w) + \frac{\beta_t}{t}h(w) \right\}$$

end for

du gradient que de juste considérer un 'gradient instantané' mais en considérant une moyennisation $\bar{g}_t$[13].

La troisième étape est celle de la mise à jour des poids. La calculabilité de la mise à jour des poids déterminera l'intérêt de l'algorithme. C'est pourquoi il est intéressant de considérer des fonctions ψ et h suffisamment simples pour que le problème de minimisation 3 puisse être résolue explicitement et/ou rapidement.

Nous proposons par la suite différentes formes que peut prendre la mise à jour des poids en fonction de la structure du problème et donc plus particulièrement de ψ et de h.

Pour une fonction de régularisation $\psi(w)$ simplement convexe Xiao (2009) montre qu'on peut choisir toute suite $\{\beta_t\}_{t\geq 1}$ d'ordre $\sqrt{t}$, et on obtient alors une borne de regret en $O(\sqrt{t})$. Dans ce qui suit on choisira $\gamma > 0$ et on prendra la suite $\beta_t = \gamma\sqrt{t}$.
Voici quelques exemples de l'application de la RDA pour des fonctions de régularisation simplement convexes classiques :

- *Méthode de moyennisation duale de Nesterov.* Soit $\psi(w)$ la fonction indicatrice d'un ensemble fermé convexe $\mathcal{C}$. En choisissant $h(w) = \frac{1}{2}\|w\|_2^2$ alors l'équation (3) donne :

$$w_{t+1} = \Pi_{\mathcal{C}}\left(-\frac{\sqrt{t}}{\gamma}\bar{g}_t\right). \tag{4}$$

[13] C'est cette étape qui a donné son nom à la méthode car la moyennisation se passe sur l'espace dual (celui des gradients).

- *RDA avec régularisation "soft l_1".* Soit $\psi(w) = \delta\|w\|_1$, pour $\delta > 0$ et $h(w) = \frac{1}{2}\|w\|_2^2$. On a alors pour tout $i = 1, ..., n$:

$$w_{t+1}^{(i)} = \begin{cases} 0 & \text{si } |\bar{g}_t^{(i)}| \leq \delta \\ -\frac{\sqrt{t}}{\gamma}(\bar{g}_t^{(i)} - \delta \text{sign}(\bar{g}_t^{(i)})) \end{cases} . \tag{5}$$

- *RDA exponentielle.* Soit $\psi(w)$ la fonction indicatrice du simplexe S_n et $h(w) = \sum_{i=1}^n w^{(i)} \ln w^{(i)} + \ln n$, la fonction d'entropie négative. On alors pour $i = 1, ..., n$:

$$w_{t+1}^{(i)} = \frac{1}{Z_{t+1}} \exp\left(-\frac{\sqrt{t}}{\gamma}\bar{g}_t^{(i)}\right). \tag{6}$$

Si la fonction de régularisation $\psi(w)$ est fortement convexe, on peut alors utiliser toute suite $\{\beta_t\}_{t\geq 1}$ non-négative et non-décroissante qui ne croit pas plus vite que $O(\ln(t))$. On obtient alors une borne de regret en $O(ln(t))$. Par simplicité, on prendra pour la suite pour tout $t \geq 1, \beta_t = 0$.
Voici quelque exemple de l'application de régularisation fortement convexe à la méthode RDA :

- *RDA avec régularisation l_2.* Soit $\psi(w) = \frac{\lambda}{2}\|w\|_2^2$ pour $\lambda > 0$. On a alors

$$w_{t+1} = -\frac{1}{\lambda}\bar{g}_t. \tag{7}$$

- *RDA avec régularisation l_1/l_2^2.* Soit $\psi(w) = \delta\|w\|_1 + \frac{\lambda}{2}\|w\|_2^2$ avec $\delta > 0$ et $\lambda > 0$. On a pour $i = 1, ..., n$:

$$w_{t+1}^{(i)} = \begin{cases} 0 & \text{si } |\bar{g}_t^{(i)}| \leq \delta \\ -\frac{\sqrt{t}}{\lambda}(\bar{g}_t^{(i)} - \delta \text{sign}(\bar{g}_t^{(i)})) \end{cases} . \tag{8}$$

- *RDA avec régularisation l_1 améliorée.* Soit $\psi(w) = \delta\|w\|_1$ avec $h_\rho(w) = \frac{1}{2}\|w\|_2^2 + \rho\|w\|_1$ où $\rho \geq 0$ est un terme de renforcement de la sparsité. On a alors pour $i = 1, ..., n$:

$$w_{t+1}^{(i)} = \begin{cases} 0 & \text{si } |\bar{g}_t^{(i)}| \leq \delta_t^{\text{RDA}} \\ -\frac{\sqrt{t}}{\gamma}(\bar{g}_t^{(i)} - \delta_t^{\text{RDA}} \text{sign}(\bar{g}_t^{(i)})) \end{cases} . \tag{9}$$

avec $\delta_t^{\text{RDA}} = \delta + \frac{\gamma\rho}{\sqrt{t}}$.

Le tableau [7.2] donne un récapitulatif des méthodes que nous avis réussies à mettre en oeuvre numériquement et sur lesquels portera la section 7.4

7.3 Propriétés théoriques

Nous rapellons dans cette section les principales propriétés théoriques de l'algorithme d'optimisation RDA, c'est à dire les bornes de regret dans le cas où la fonction $\psi(w)$ est simplement convexe et dans le cas où elle est fortement convexe.
Les résultats théoriques de Xiao (2009) reposent sur les hypothèses suivantes :

Algorithme	Acronyme	mise à jour
Randomized Dual Averaging Nesterov	RDANesterov	4
Randomized Dual Averaging soft l_1	RDAl1	5
Randomized Dual Averaging $\frac{l_1}{l_2^2}$	RDAl1l2	8
Randomized Dual Averaging l_2	RDAl2	7
Randomized Dual Averaging enhanced l_1	RDAenhancedl1	9

Table 6: Récapitulatif des méthodes mis en oeuvre numériquement

- La fonction de régularisation $\psi(w)$ est une fonction δ-convexe, son domaine de définition est fermé et on a $\min_w \psi(w) = 0$.
- La fonction $f_t(w)$ est sous-différentiable et convexe sur le domaine de ψ.
- La fonction $h(w)$ est 1-fortement convexe sur le domaine de ψ et on a $\min_w h(w) = 0$.

7.3.1 Bornes de regret

Dans le cadre de l'optimisation séquentielle régularisée, on ajoute une fonction de régularisation $\psi(w)$ à la fonction de coût $f_t(w)$. Le regret pour tout $w \in dom(\psi)$ est alors donné par :

$$R_t(w) = \sum_{t=1}^{T}(f_t(w_t) + \psi(w_t)) - \sum_{t=1}^{T}(f_t(w) + \psi(w)) \tag{10}$$

Pour une fonction de régularisation $\psi(w)$ simplement convexe on a alors le résultat suivant :

Corollaire 7.1. *Soit $\{w_t\}_{t\geq 1}$ et $\{g_t\}_{t\geq 1}$ généré par l'algorithme 7.1 avec $\beta_t = \gamma\sqrt{t}$ et supposons que $\|g_t\|_* \leq G, \forall t \geq 1$. Alors pour tout $t \geq 1$ et $w \in \{w \in dom(\psi) | h(w) \leq D^2\}$, la borne de regret R_t est telle que :*

$$R_t(w) \leq \left(\gamma D^2 + \frac{G^2}{\gamma}\right)\sqrt{t} \leq O(\sqrt{t}) \tag{11}$$

Dans le cas d'une fonction de régularisation fortement convexe on a un meilleur contrôle et on obtient un regret en $O(\log(t))$ si β_t est tel que $\beta_t \leq O(\log(t))$. Le terme de constante dépendra alors du choix spécifique de β_t.

7.4 Performances

7.4.1 Sélection des hyperparamètres

De la même manière que pour les algorithmes étudiés en 2.1, nous sélectionnons les hyperparamètres de la RDA par recherche par grille avec validation croisée. Ici, on prend $K = 5$ et on optimise les algorithmes pour $T = 10000$ itérations. Le critère d'optimisation est encore une fois le taux d'erreur. Pour les grilles Tab[7] on obtient les hyperparamètres[14] donnés par Tab[8].

[14]On notera que nous n'avons pas réussi à trouver des hyperparamètres permettant de faire converger l'algorithme EDA au delà du taux d'erreur d'une loi uniforme (à savoir un taux d'erreur de 0.1 environ).

Optimiseur	Paramètres
NDA	$z = \{5, 10, 20, 50, 100, 200, 500, \infty\}$
soft l_1 regularization	$\delta = \{0, 0.0001, 0.001, 0.005, 0.01, 0.02, 0.025\}$
EDA	–
l_2 regularization	$\lambda = \{0.001, 0.01, 0.05, 0.1, 0.5, 1, 1.3, 1.8\}$
l_1/l_2^2 regularization	$\lambda = \{0.0001, 0.001, 0.005, 0.01, 0.02\}, \delta = \{0.0001, 0.001, 0.005, 0.01, 0.02\}$
Enhanced l_1	$\gamma = \{10, 50, 100\}, \delta = \{0.0001, 0.001, 0.01\}, \rho = \{0.0001, 0.001, 0.01\}$

Table 7: Grille de recherche pour méthodes RDA

Optimiseur	γ	λ	δ	ρ	z
NDA	–	–	–	–	200
soft l_1 regularization	–	–	0.0001	–	–
l_2 regularization	–	0.01	–	–	–
l_1/l_2^2 regularization	–	0.02	0.001	–	–
Enhanced l_1	10	–	0.0001	0.001	–

Table 8: Hyperparamètres sélectionné par recherche par grille avec validation croisée

7.4.2 Comparaison avec les algorithmes du cours

Temps de calcul, performance et sparsité

Les algorithmes RDA présentés sont tous dérivés de SGD classique et présentent une mise à jour des poids explicite, ils héritent donc d'un temps de calcul (complexité) similaire à celle de SGD : autour de 3 secondes pour 10000 itérations.

Pour ce qui est des précisions atteintes et de la sparsité des solutions la comparaison est bien capturée par le graphe [20].
La méthode RDAl1l2 excelle avec une sparsité de 296 pour un taux d'erreur de 0.0108. Elle est suivi d'une autre méthode issue de l'article de Xiao (2009) (toujours devant les meilleurs méthodes du cours pour la sparsité) RDAenhancedl1 avec une sparsité de 524 et un taux d'erreur de 0.0118.
Viennent en suite les "méthodes phares" du cours Adamproj, Adaproj, SMDproj auxquelles s'est ajoutée RDAl2 pour des sparsités entre 550 et 600 et des taux d'erreurs en dessous de 0.2.
Enfin la méthode RDAsoftl1 arrive bien à avoir une meilleure sparsité que ses homologues SGD et SGDproj mais on observe une détérioration de la performance. Enfin RDANesterov semble un peu moins convaincant au niveau de la sparsité mais arrive à atteindre une très bonne performance.

Sensibilité à l'initialisation

Les algorithmes issus de la RDA héritent aussi, de SGD, d'une faible sensibilité à l'initialisation (de façon relative). À titre indicatif la figure [21] présente des trajectoires des taux d'erreurs en fonction du nombre d'itération pour les différentes initialisations présentées en 5.1 pour deux des méthodes de RDA (les résultats sont semblables pour le restant des méthodes RDA présentées).

Robustesse aux attaques

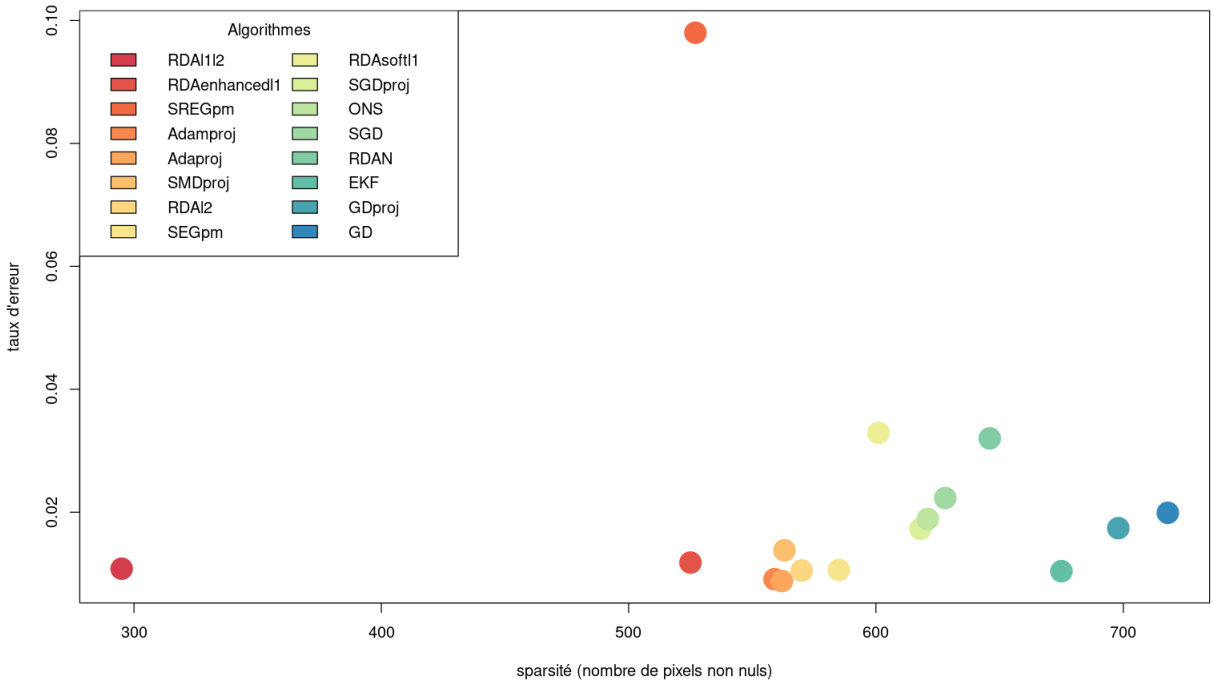

Figure 20: Taux d'erreur en fonction de la sparsité pour les méthodes étudiées (10000 itérations, sauf pour SREGpm 100000)

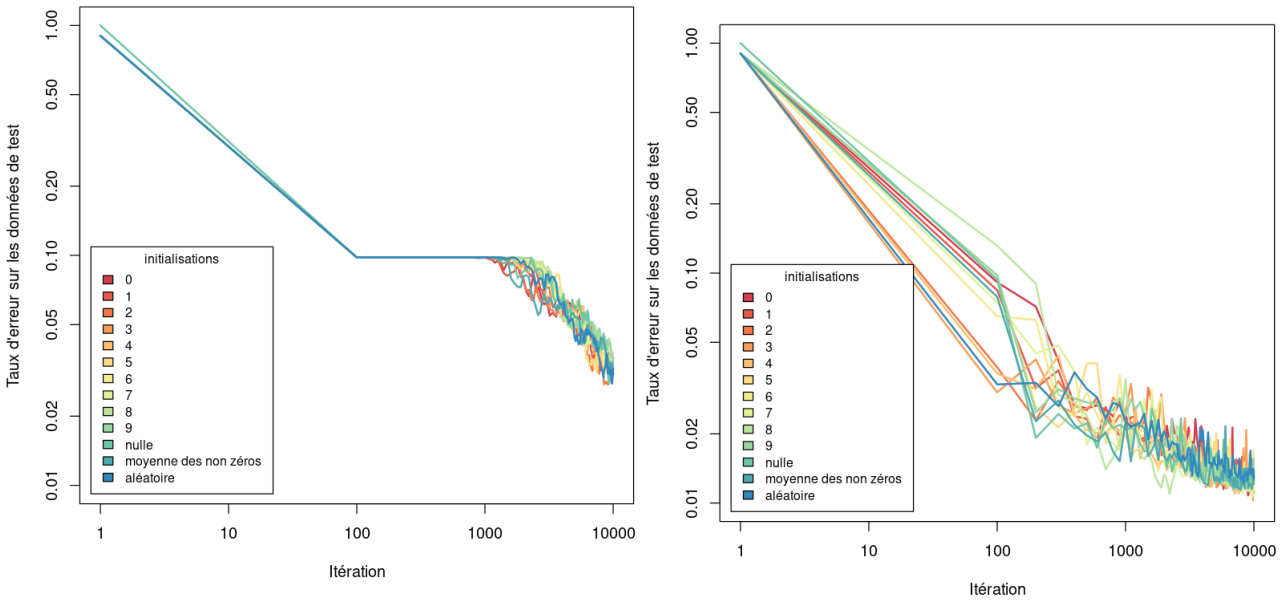

Figure 21: Taux d'erreur sur les données de test en fonction du nombre d'itérations pour les différentes initialisations (à gauche RDANesterov, à droite RDAenhancedl1)

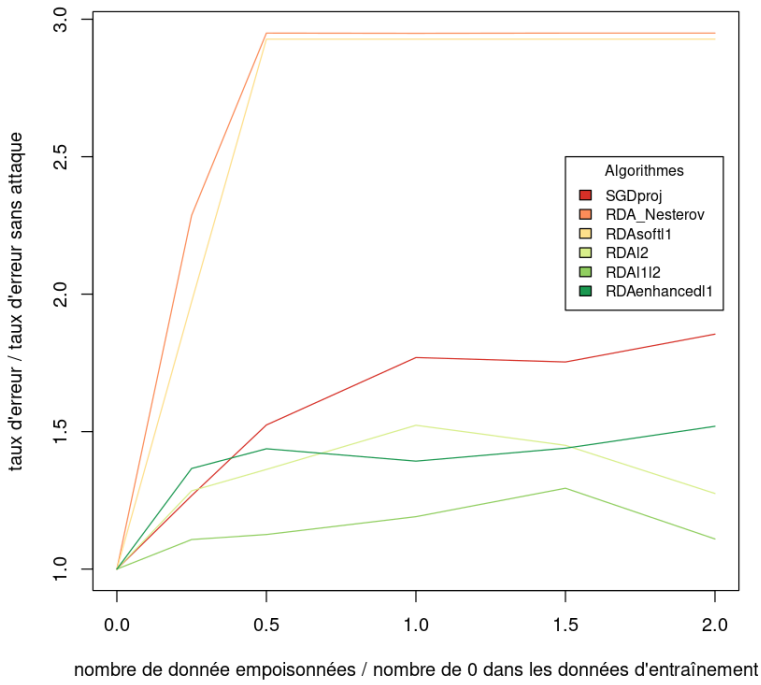

Figure 22: Détérioration du taux d'erreur en fonction du ratio du nombre de données empoisonnées par rapport au nombre de 0 (moyenne de 10 expériences de 10000 itérations pour chaque algorithme)

On a aussi soumis les méthodes RDA à la procédure d'empoisonnement des données progressives décrites en 6. La figure [22] présente les résultats. On y voit que malheureusement, RDANesterov et RDAsoftl1 semblent être plus sensible à l'empoisonnement des données dans un premier temps, avant d'atteindre un certain seuil[15]. Par contre, RDAl2 RDAl1l2 et RDAenhancedl1 semblent (en plus d'améliorer la sparsité des résultats de SGD classique) améliorer la robustesse à l'attaque d'empoisonnement.
On a vu (notamment dans la figure [20]) que ces trois méthodes améliorent grandement la sparsité de SGD. L'amélioration de la robustesse à l'empoisonnement est peut-être juste dûe au fait qu'on a contaminé les données avec [17] et que les pixels correspondants à "la queue" du '6' sont éliminé par le seuillage effectué par RDA.
Pour mieux vérifier si ces algorithmes engendrent un renforcement de la robustesse de SGD face aux attaques par empoisonnement, il faudrait contaminer les données avec plusieurs images différentes.

Estimation du regret

En ce qui concerne le regret des méthodes RDA. On va procéder comme pour la section 3 en estimant le regret de chacune des méthodes.
Tout d'abord la méthode RDANesterov s'interprète comme une minimisation sur un convexe $\mathcal{K}$ (qui dans notre cas est la boule $l1$ de rayon $z = 200$); on va donc estimer $x^* := \underset{x\in\mathcal{K}}{min} f(x)$ par le x (le plus petit lors des expériences i.i.d.) obtenu avec Adamproj

[15] probablement dû au fait qu'on empoisonne toujours avec la même image plutôt qu'avec plusieurs variations...

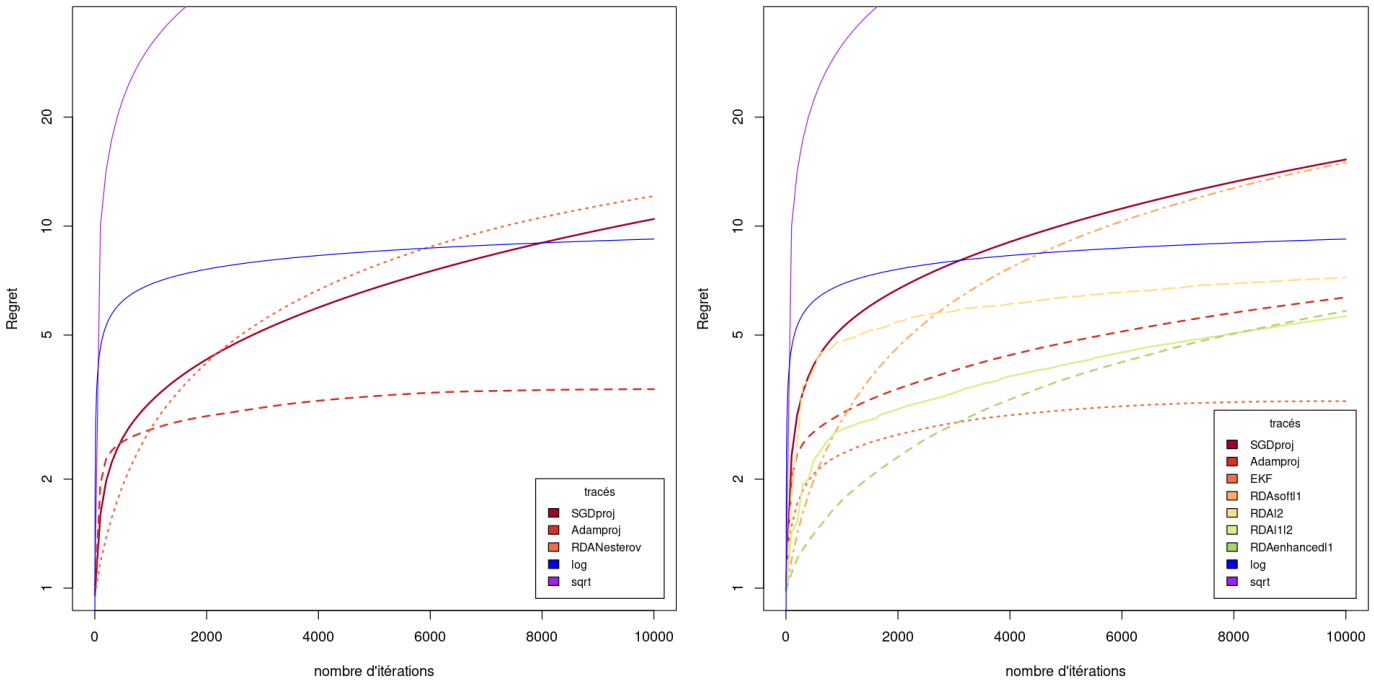

Figure 23: Estimation des regrets pour les algorithmes RDA

avec une projection sur la boule l_1 de rayon $z = 200$.
Pour ce qui est des autres algorithmes RDA mis en oeuvre, ceux-ci s'interprétant comme des minimisations sans contraintes on va estimer x^* par le plus petit x obtenu (lors de toutes les expériences) pour toutes les méthodes sans la projection ($z = \infty$).
Les résultats de l'estimation des regrets sont regroupés dans la figure [23]

Nous pouvons voir sur la figure [23] que les regrets des méthodes ayant une fonction de régularisation $\psi(w)$ simplement convexe (RDAsoftl1, RDAenhancedl1) sont de l'ordre de $\sqrt{T}$ quand les méthodes ayant une fonction de régularisation $\psi(w)$ fortement convexe (RDAl2, RDAl1l2) ont un regret estimé logarithmique. On notera que ces résultats sont biens cohérents (voir section 7.3) avec les résultats théoriques présentés dans Xiao (2009).

7.5 Sensibilité aux hyperparamètres : Le cas l_1/l_2^2

Comme nous avons pu le voir (7.4.2) l'algorithme RDA avec régularisation l_1/l_2^2 est particulièrement performant dans la résolution du problème qui nous intéresse que ce soit pour les performances ou la sparsité des solutions qu'il propose. Il nous semble alors intéressant de voir l'impact du choix des hyperparamètres sur la sparsité des solutions pour cet algorithme en particulier.

On peut voir Fig[24] que la sparsité des solutions augmente bien en fonction du paramètre de régularisation l_1, λ, et du paramètre de régularisation l_2, δ, ce qui est ce qui nous intéresse particulièrement dans l'utilisation de cet algorithme. On voit ainsi que la méthode RDA permet d'obtenir des solutions performantes du point de vue de la précision et sur lesquelles on a un bon contrôle de la sparsité dans le cas de l'utilisation de la régularisation l_1/l_2^2.

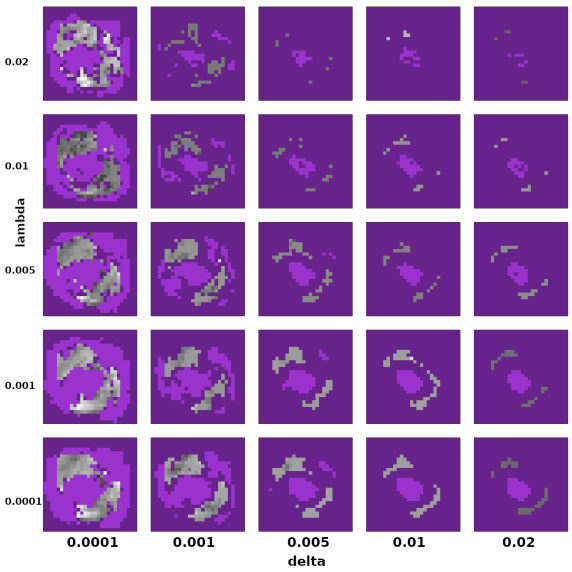

Figure 24: Sparsité de RDA avec régularisation l_1/l_2^2 en fonction de λ et δ

References

Biggio, B., Nelson, B., and Laskov, P. (2013). Poisoning attacks against support vector machines.

Nesterov, Y. (2009). Primal-dual subgradient methods for convex problems. *Mathematical Programming*, 120:221–259.

Wintenberger, O. (2021). Online Convex Optimisation . http://wintenberger.fr/cours/OCO/OCO2021.pdf. [M2 Course, 2021].

Xiao, L. (2009). Dual averaging method for regularized stochastic learning and online optimization. In Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C., and Culotta, A., editors, *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc.