SCIENCES
SORBONNE
UNIVERSITÉ

Online Convex Opitmization: Project

Students:
Hakim Chekirou
Aymen Merrouche

June 2020

Contents

1 Introduction

This project aims at applying the online gradient methods in the context of linear classification on the MNIST dataset.

2 Formal Definition of the Problem

The classification problem aims at separating two (or more) classes of d-dimensionnal data points. We consider a Binary Linear classification problem, meaning that the learner must learn a hyperplane seperating two classes. Each data point denoted a_i is represented in R^d. The labels, b_i take their values in
± 1

We base our work on the MNIST data set witch an established benchmark in machine learning, consisting of 16*16 grayscale images of handwritten digits (0-9), we restrict the problem to classifying the 0 class versus all the others, the data preparation is detailed in the next section.

More formally, given $\{a_i \in R^d, i \in [n]\}$ and $\{b_i \in \{+1, -1\}, i \in [n]\}$, we aim at finding a vector $\mathbf{x} \in R^d$, that minimizes the classification error :

$$\min_{\mathbf{x} \in R^d} \sum_{i=1}^{n} \mathbb{1}_{sign(\mathbf{x}^T.a_i) \neq b_i} \tag{1}$$

were $\mathbb{1}$ is the indicator function.
To solve this problem, we will the Linear Support vector machine model. The (0-1) loss defined in 1 is replaced by a convex surrogate, the hinge loss defined by :

$$Hinge(x) = max(0, 1 - x) \tag{2}$$

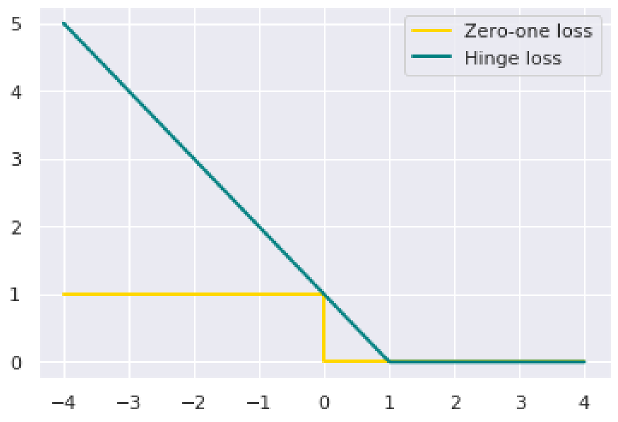


Figure 1: Hinge loss and (0-1) loss

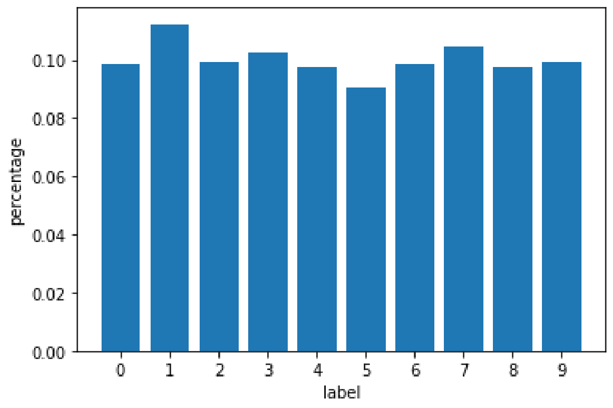
A quadratic regularization parameter is added to the objective function, the problem becomes:

$$\min_{\mathbf{x} \in R^d} \frac{\lambda}{2}||\mathbf{x}||^2 + \frac{1}{n}\sum_{i=1}^{n} hinge(b_i * \mathbf{x}^T a_i) \tag{3}$$

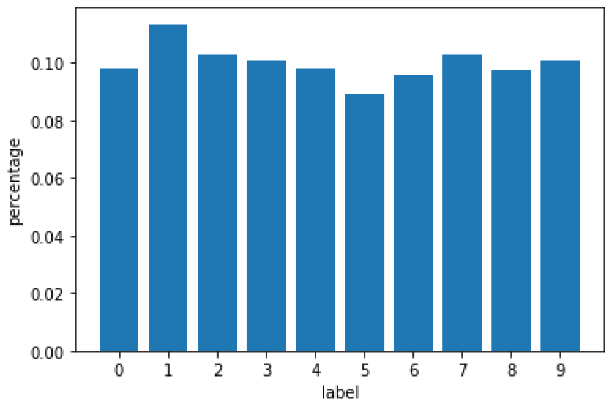
3 Data Preparation and exploration

As mentioned earlier, a digit recognition benchmark, MNIST is used. It consists of 16 * 16 gray scale (0-255) images of handwritten digits. The data set is separated into test and train sets, the first one contains 10000 images, the second 60000. The images are stored in the form of 784-vectors.

We first start by checking for missing values or duplicates which are not present. We the standardize the feature vectors in the [0,1] range by diving by 255.



(a) Label distribution on the train set



(b) Label distribution on the train set

Afterwards, we restrict the problem to a 0 vs all problem by setting all labels equal to 0 at +1 and the others to -1. We plot the distribution of the labels in the data set after the restriction to the 0 vs all paradigm.

4 Gradient descent

The first algorithm we will examine is the unconstrained gradient descent.

Choice of λ. We first optimize the λ parameter, to do so, we run gradient descent up to 20 iterations for every value of lambda, the results are represented in figure 3. When the values of λ are too big, the regularization term overtakes the average cost term in the objective function and results is poor performances. The optimal value for λ is thus 0.3 according to our experiment.

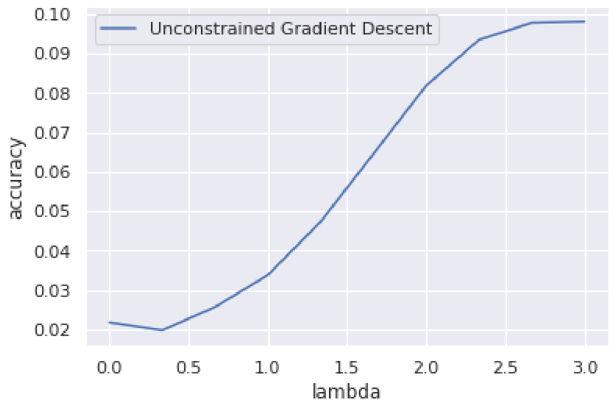


Figure 3: Classification error per value of λ for the unconstrained gradient descent algorithm.

Choice of z : radius of the ℓ^1 ball. For the constrained version of gradient descent, the choice of the radius of the ℓ^1 ball is important, as can be seen in figure 4. In this experiment, we vary the value of z and plot after 20 iterations of the constrained gradient descent algorithm the classification error on the test set. As ca be seen, the value are not as stable, we choose the value 100 as it gives the best performance.

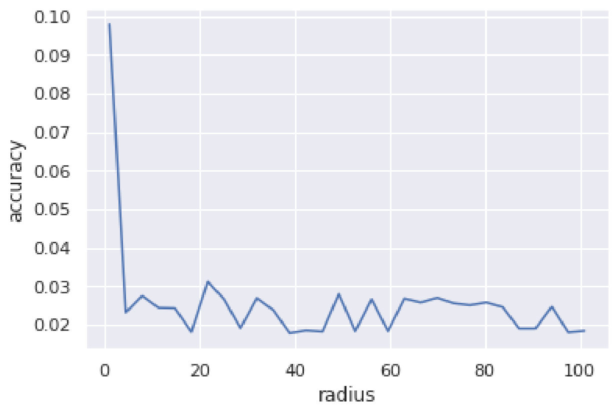
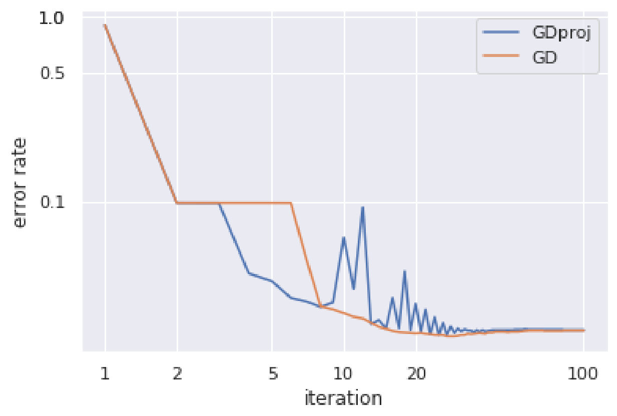


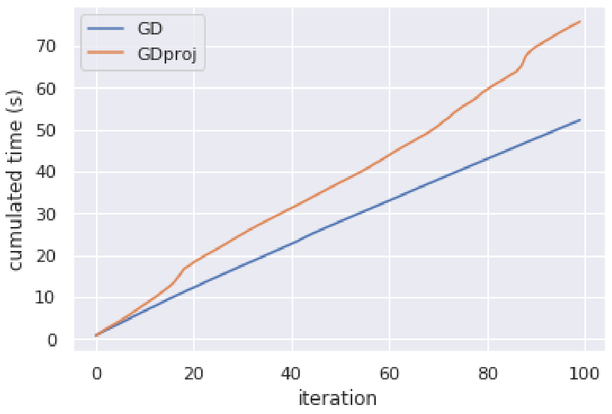
Figure 4: Classification error on the test set per value of z for the constrained gradient descent algorithm.

Choice of η. Given that the regularized f is a strongly convex function, we have from the course that by choosing $\eta_t = 1/(\lambda * t)$, $\tilde{O}(\frac{1}{\varepsilon})$ iterations suffice to attain an ϵ-approximate solution.

Results for GD vs GDproj. We end this section by plotting the results of running the two algorithms, the unconstrained gradient descent and the projected version. As can be seen in plot 5a, the vanilla gradient descent remains at a plateau for quite a while before converging to the best solution, for the projected version, it doesn't remain as long on the plateau, but fluctuates due to over fitting. For the time taken to train, figure 5b shows that the projected version of the algorithm is slower by a linear factor, due to the projection step.



(a) Error rate on for the two version of gradient descent using $z = 100$ and $\lambda = 1/3$.



(b) Time taken by both algorithm per iteration.

Figure 5: Tests on the two version of gradient descent.

5 Stochastic Gradient Descent

5.1 Unconstrained SGD vs Projected SGD

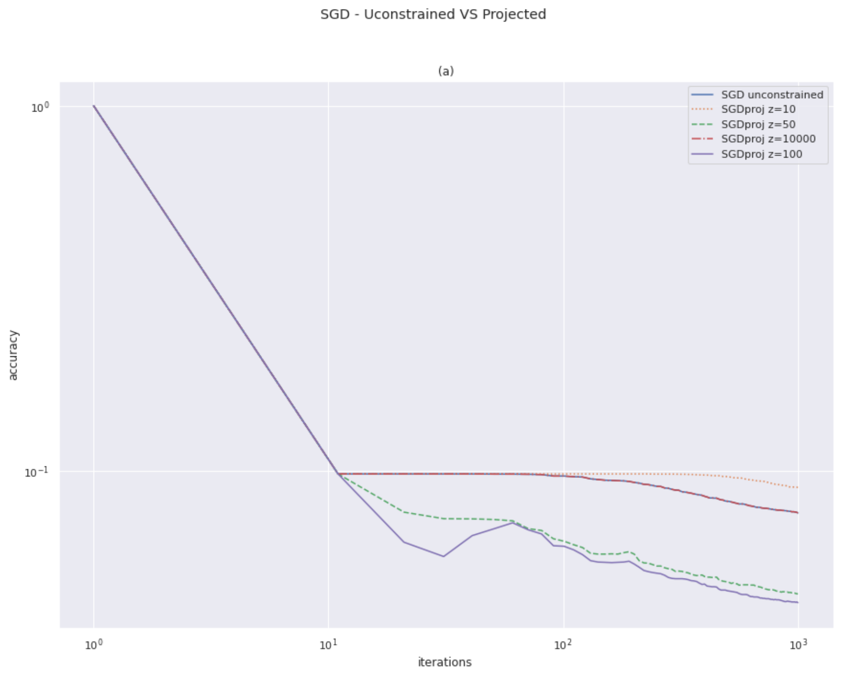


Figure 6: The plots above represent the classification accuracy of SGD on a test set across different learning iterations. Both the unconstrained version and the projected version are tested. For the projected version, we vary the radius z of the ℓ_1 ball. We use a **log log scale**.

Hyper-parameters.

- Number of epochs $T = 1000$.
- Regularization factor $\lambda = \frac{1}{3}$.
- Initialization at 0.
- Learning rate $\eta_t = \frac{1}{\lambda t}$.
- Same sampling of the gradients across iterations (same seeding).

Unconstrained SGD vs Projected. Projection on the ℓ_1 ball for radius $z \in \{50, 100\}$ allows a faster convergence and better performance. For the latter values of z, the algorithm continues to learn across all iterations (no plateau). This is due to the sparsity introduced by the projection on the $\ell_1$ball which helps with learning by detecting important pixels with respet to the objective function.

For $z = 10$, SGDproj is too constrained and projection on $\ell_1$ball introduces too much sparsity hence impeding learning.

For $z = 10000$, the resulting $\ell_1$ball is too large and the algorithm behaves exactly as if it was unconstrained (plots coincide).

Concerning running time, they all have approximately the same $4.08s \sim 4.60s$.

5.2 Projected SGD vs Projected GD

5.2.1 Accuracy

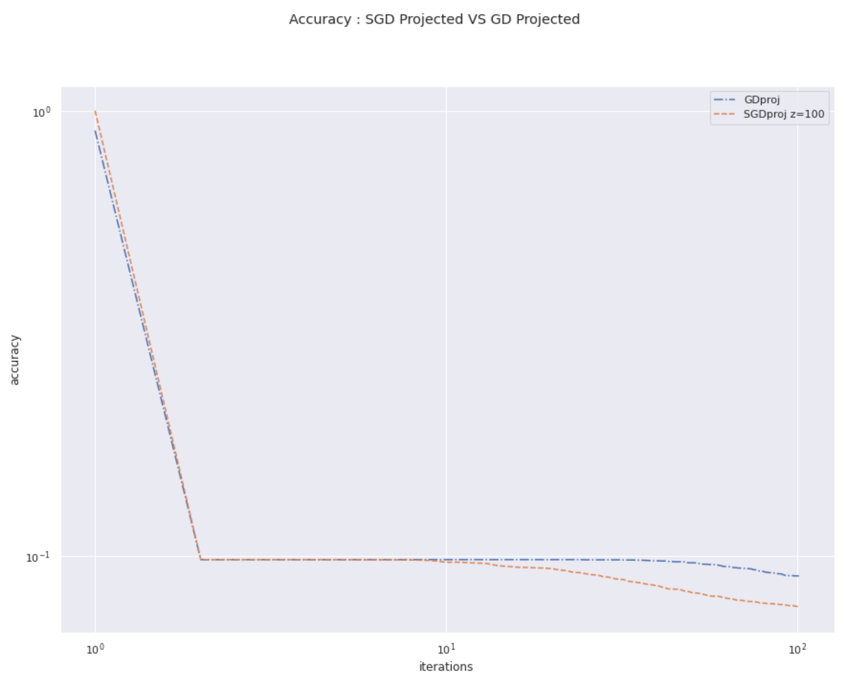


Figure 7: The plots above represent the classification accuracy of SGD and GD on a test set across different learning iterations. Both SGD and GD include a step of projection on the ℓ_1 ball of radius $z = 100$. We use a **log log scale**

Hyper-parameters.

- Same radius $z = 100$.
- Same number of epochs $T = 100$.
- Same regularization factor $\lambda = \frac{1}{3}$.
- Both initialized at 0.
- Same learning rate $\eta_t = \frac{1}{\lambda t}$.

Accuracy. SGDproj performs better and converges faster than GDproj and manages to escape faster from the plateau in which they both seem to be stuck in the beginning. This is due to the noise introduced by the sampled gradients, which allows to escape from local minima. The averaging at each step allows SGDproj to have smooth accuracy graphs and to accumulate information captured by previously sampled gradients.

5.2.2 Ruining Time

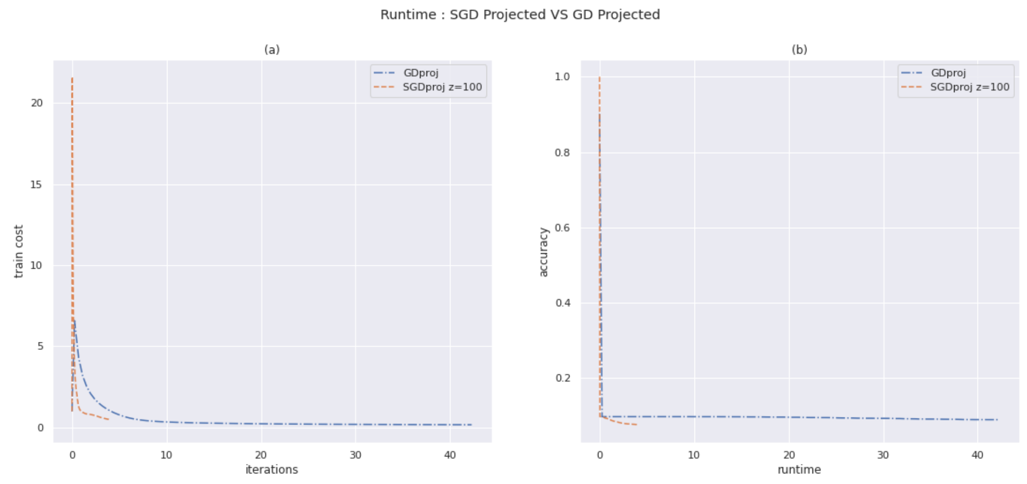


Figure 8: The plots above represent the classification accuracy (a) and the train cost (b) of both GD and SGD along with the time elapsed from the beginning of each algorithm's execution (end of plot means end of execution).

Hyper-parameters.

- Same radius $z = 100$.
- Same number of epochs $T = 100$.
- Same regularization factor $\lambda = \frac{1}{3}$.
- Both initialized at 0.
- Same learning rate $\eta_t = \frac{1}{\lambda t}$.

GDproj (blue plot) takes much longer to train : $42.28s$ vs $4.08s$ for SGDproj. Moreover it doesn't perform as good as SGDproj on the test set although it achieved a smaller training cost.

This behaviour is expected. The cost of an iteration of SGD is $O(d)$ (d being the dimension of the problem which equals $784 + 1$- +1 for the intercept-) because only one gradient is computed. Whereas for GD, we need to compute the gradient across all training points, an then use the average of these gradients in the gradient step. This results in an iteration of cost $O(nd)$ where n denotes the size of the training set.

Moreover, to achieve an accuracy less than some $\epsilon > 0$, both SGD and GD need $T = O(\epsilon^{-1})$. This results in a total cost of $O((d + P)\epsilon^{-1})$ for SGDproj and $O((nd + P)\epsilon^{-1})$ for GD. Where P is the cost of the projection on the $\ell_1$ball which equals $O(d \log(d))$.

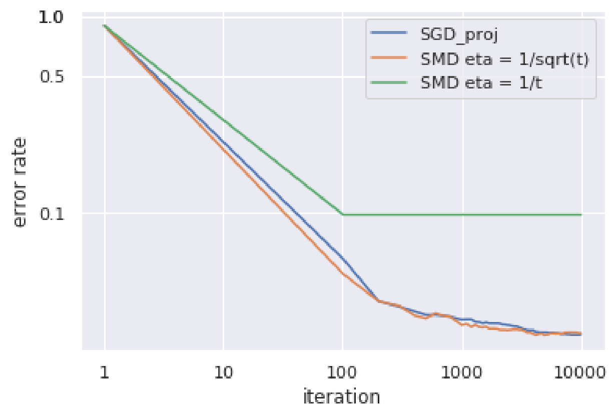
In some cases, where the sampled gradients are large resulting in large variance and hence introducing a form of instability, SGD may be slow to converge. Generally, we have a trade-off of fast computation per iteration and slow convergence for SGD and slow computation per iteration and fast convergence for gradient descent.

6 Regularized follow the leader

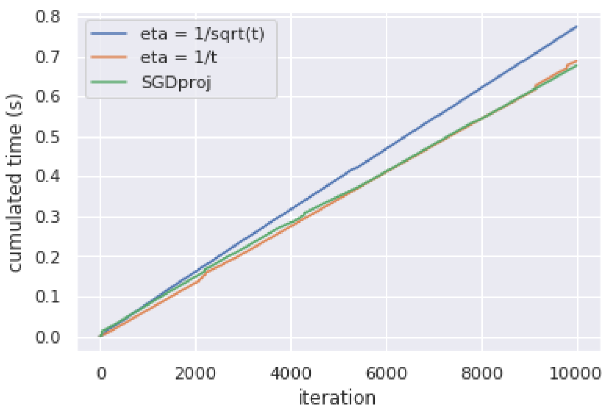
6.1 Stochastic Mirror Decent

We first examine the stochastic mirror decent algorithm, and compare it to the projected stochastic gradient descent.
As advised, we tried multiple setting for the step size η_t.

SMD vs SGDproj. We plot both the error rate on the test set and the training time of the two algorithms in figure 9. As can be seen in the plot 9a, by setting $\eta_t = \frac{1}{\sqrt{t}}$ we achieve very similar results to SGDproj, SMD is faster in the beginning but fluctuates more. By setting $\eta_t = \frac{1}{t}$, SMD doesn't converge. For the computation time, we can see in plot 9b, that the training times are linear in terms of the number of total iterations T, both algorithms are very similar thus leading to nearly identical training times.



(a) Error rate on the test set for SGDproj and SMD with two settings for η, we keep $z = 100$ as previously and $\lambda = 1/3$.



(b) Time taken by both algorithms per iteration.

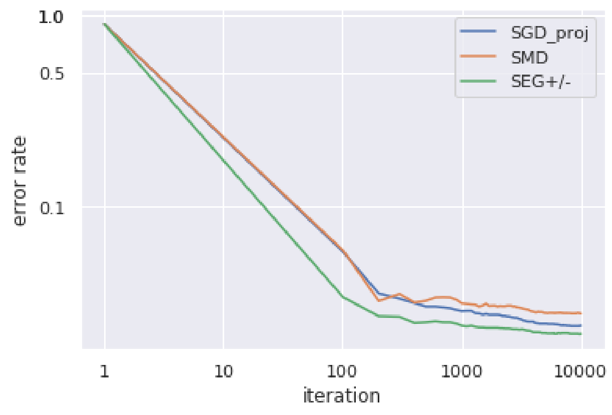
Figure 9: Tests on the two versions of SMD compared to SGDproj.

6.2 Stochastic Exponentiated Gradient +/-

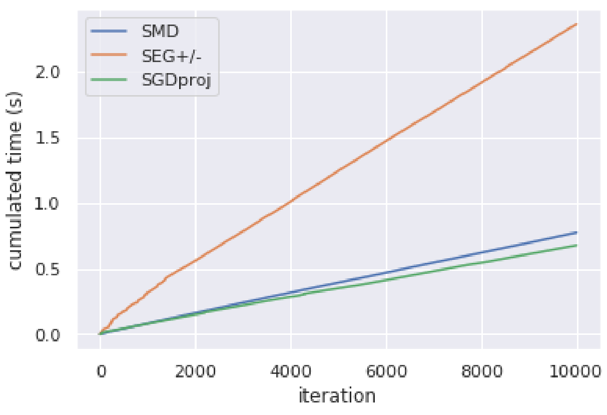
We implement the stochastic exponentiated Gradient +/- and compare it to both projected SGD and to the previous implementation of SMD.

Parameters. We keep the same value for $\lambda = 1/3$ and the radius $z = 100$. We run in total for $T = 10k$ iterations.

Results. In figure 10a, we plot the test error rates achieved by SGproj, SMD (the best verison from earlier) and SEG+/-, it is clear that SEG+/- outperforms both algorithms from the start and converges to a better error rate on the test set. For the training time, we plot it in figure 10b, as expected, SEG+/- is far slower than bothe SGDproj and SMD, the times are linear in total number of iteration T. We also note that SEG+/- has a greater memory complexity as well.



(a) Error rate on the test set for SGDproj, SMD and SEG+/-, we keep $z = 100$ as previously and $\lambda = 1/3$.

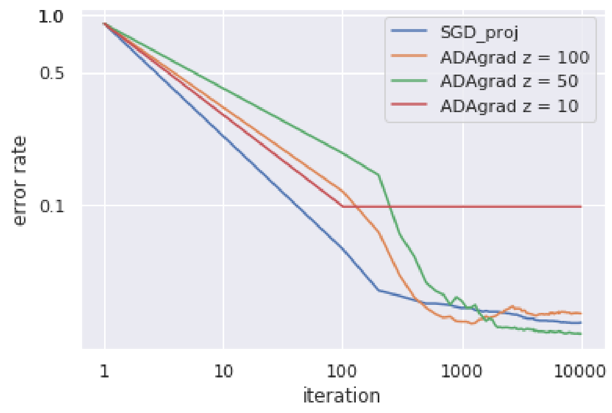


(b) Cumulative time taken by SEG+/-, SMD and SGDproj algorithms per iteration.

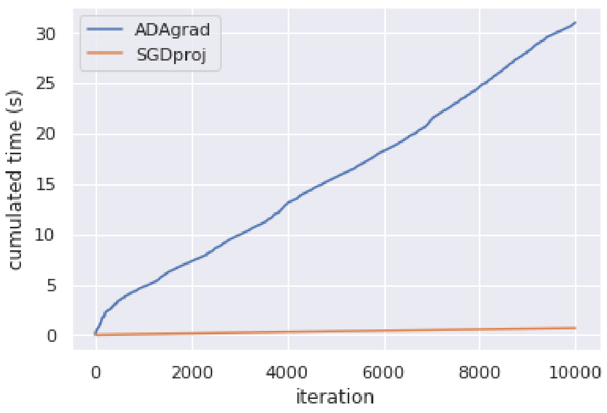
Figure 10: Comparison of SEG+/- in contrast to SGDproj and SMD.

6.3 Stochastic AdaGrad

We look now at the stochastic AdaGrad algorithm.
We vary the value of the radius z and observe both the test error rate and the time needed to train the algorithm.



(a) Error rate on the test set for AdaGrad for multiple values of z compared to SGDproj, we keep $\lambda = 1/3$.



(b) Cumulative time taken by ADAGrad and SGDproj algorithms per iteration.

Figure 11: Comparison of SEG+/- in contrast to SGDproj and SMD.

Accuracy. In figure 11a, for $z = 100$, AdaGrad initially outperforms SGDproj but then shows evidence of over-fitting as the error rate increases and converges to a higher value than SGDproj. For $z = 50$, we do not observe this phenomenon, AdaGrad converges to a smaller value, but is initially slower. For $z = 10$, the algorithm doesn't converge.

Train time. As expected and as observed in figure 11b, AdaGrad is much slower than SGD, as the projection step and the computation of the matrix D adds a constant time per iteration. It is still linear in T but slower than SGDproj.

7 Incomplete information methods

In this section we are interested the the Stochastic Randomized Exponentiated Gradient +/- where we face incomplete information (gradient only on one pixel at each iteration).

7.1 SREG± VS SEG±

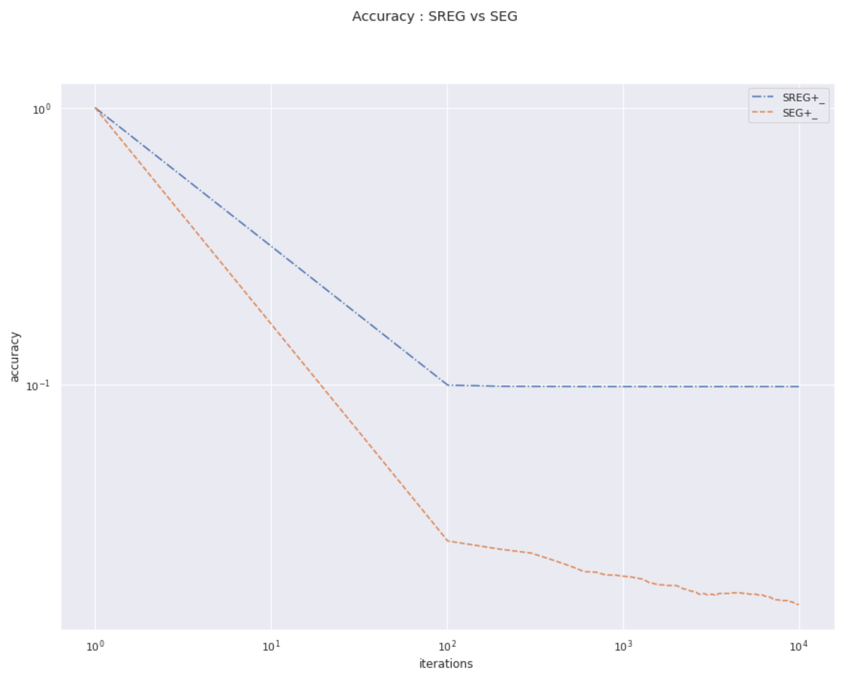


Figure 12: The plots above represent the classification accuracy on a test set of both SEG± and SREG± across different learning iterations.

The SREG± algorithm gets stuck at the accuracy 0.1. On the contrary, the accuracy of SEG± decreases smoothly to a minimum.

7.2 SREG±'s convergence

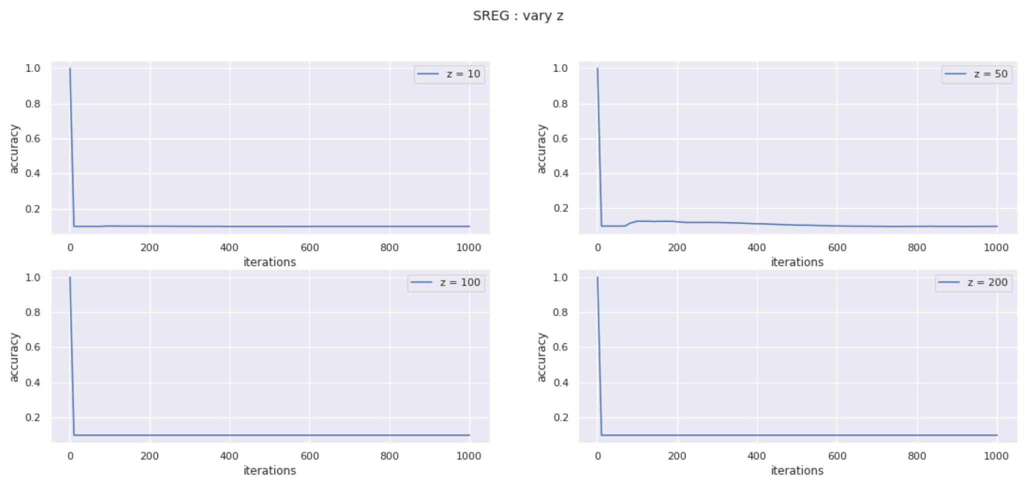


Figure 13: The plots above represent the classification accuracy on a test set of SREG± across different learning iterations for different values of the radius z of the $\ell_1$ball.

Different values of the ℓ_1-1 balls radius as well as other configurations of the learning rate were tested, the performance of SREG± is the same. This is due to the uncontrollable variance introduced by gradient sampling (gradient of one pixel at each iteration).

8 Accelerating Stochastic Gradient Descent using Predictive Variance Reduction

In this section we are interested in the work done by [1] on variance reduction for SGD. We will implement the proposed method, namely SVRG, for our problem of interest and compare it's asymptotic behaviour as well as it's accuracy with SGD.

Motivations. This paper came to deal with the caveat when using SGD. As mentioned is sub-subsection 5.2.2, the randomness introduced by the sampled gradients causes variance which may slow down the algorithm's convergence. With explicit variance reduction, this work remedies to this problem. Moreover, the proposed algorithm, unlike other works which have tackled this problem, doesn't require storage of all gradients which may pose a problem for more complex applications where storing all gradients is not conceivable.

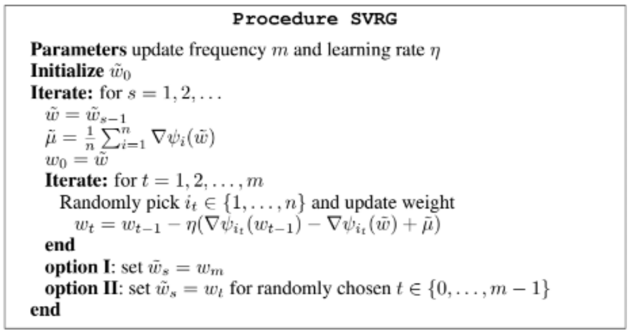
8.1 Algorithm

Let $\psi_n, ..., \psi_n$ be a sequence of vector functions from R^d to R, the goal is to optimize the following problem:

$$\min P(w), P(w) = \sum_{i=1}^{n} \phi_i(w) \tag{4}$$

Given that that $\forall i$, ϕ_i is convex and that P is strongly convex, [1] showed that for SVRG in figure 15, we have geometric convergence.

We implemented SVRG in the context of our problem, the following section presents our results.



```
Procedure SVRG
Parameters update frequency $m$ and learning rate $\eta$
Initialize $\tilde{w}_0$
Iterate: for $s = 1, 2, \ldots$
  $\tilde{w} = \tilde{w}_{s-1}$
  $\tilde{\mu} = \frac{1}{n}\sum_{i=1}^{n} \nabla\psi_i(\tilde{w})$
  $w_0 = \tilde{w}$
  Iterate: for $t = 1, 2, \ldots, m$
    Randomly pick $i_t \in \{1, \ldots, n\}$ and update weight
      $w_t = w_{t-1} - \eta(\nabla\psi_{i_t}(w_{t-1}) - \nabla\psi_{i_t}(\tilde{w}) + \tilde{\mu})$
  end
  option I: set $\tilde{w}_s = w_m$
  option II: set $\tilde{w}_s = w_t$ for randomly chosen $t \in \{0, \ldots, m-1\}$
end
```

Figure 14: The SVRG Algorithm

8.2 SVRG vs SGD :

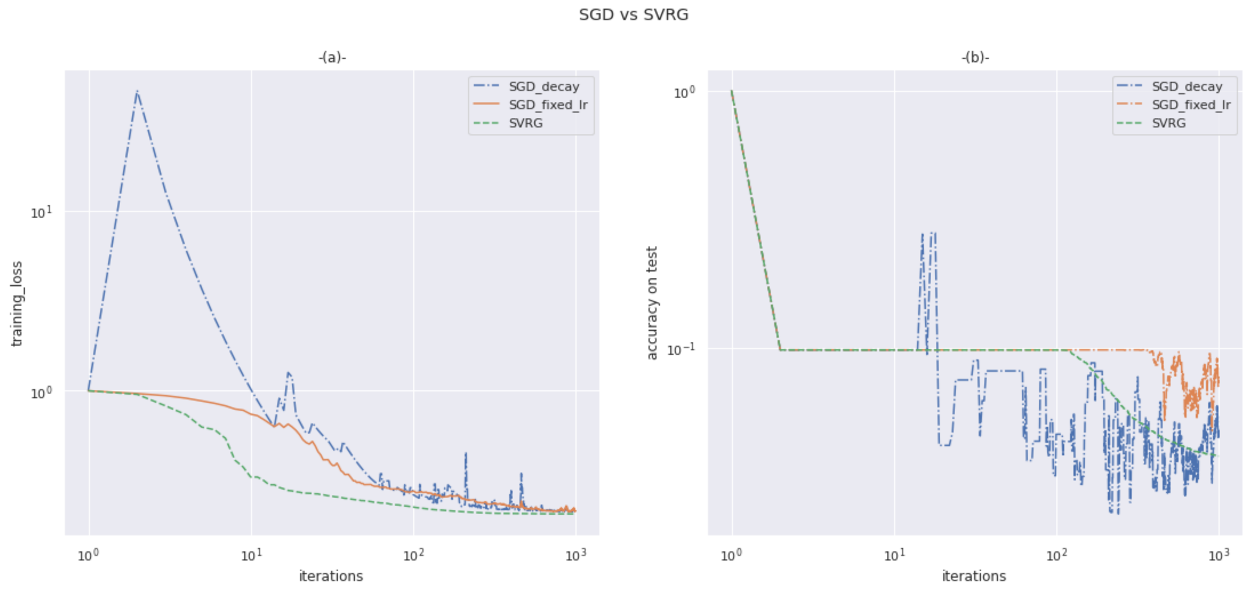


Figure 15: The plots above represent the classification accuracy on a test set as well as the training loss of SVRG and SGD (with fixed learning rate and with decreasing learning rate) across different learning iterations.

Training Loss. As expected, SGD behaves erratically. For a decreasing learning rate $\eta_t = 1/\lambda t$, the training loss drops, but fluctuates above the minimum attained by SVRG. For a fixed learning rate $\eta = 1/T$ (relatively small), a similar but less fluctuating behaviour is observed. On the contrary, for SVRG the training loss smoothly drops and converges faster and settles on a minimum.

Accuracy on test set. All three algorithms get stuck in a plateau in the beginning. SGD with decreasing learning rate's erratic behaviour is also visible here, it fluctuates around the value attained by SVRG but never settles. This is also visible with SGD with a fixed learning rate, however the latter achieves poor performances compared to the other two algorithms. SVRG, takes longer to escape from the plateau. But once it does, it's accuracy smoothly decreases and settles.

Our results coincide with the experiences illustrated in the article. Confirming that explicit reduction of the variance with SVRG deals with SGD's defect.