Analysis and comparison between State-of-the-art SFO optimization methods

Marco Tronu†

***Abstract*—In this report I study novel stochastic first order optimization methods, firstly highlighting their theoretical convergence capability and then exploring whether these theoretical results are reflected into the numerical experiments.**

I. Introduction

A. Definition of the problem of interest

The focus of this work is to compare the latest methods aimed to solve the problem of the form:

$$\min_{w\in\mathbb{R}^d}\left\{P(w) \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i\in[n]} f_i(w)\right\} \tag{1}$$

where each f_i, with $i \in 1,...,n$ is in general a non-convex function with a Lipschitz continuous gradient, which by definition satisfies the following:

$$||\nabla f_i(x) - \nabla f_i(y)|| \leq L \cdot ||x-y|| \ \forall x,y \in \mathbb{R}^d \tag{2}$$

Problem (1) is of high interest as many Machine Learning (ML) methods rely on the search of the minimum value of the function $P(w)$, which is also called Loss function, as to maximize the performance of a model (i.e., minimize the error on a dataset of interest). The ML algorithms which rely on this mechanism are of the Supervisioned Learning type and work in the following way. Given a training set $\{(x_i, y_i)\}$ with $x_i \in \mathbb{R}^d$, $y_i \in \mathbb{R}$, we define a function which quantifies the error of our model on a dataset of interest. Such function can take many forms depending on the problem, and therefore can be a convex or non-convex function, which requires different strategies to find its minima.

In the case of a binary classification problem, in which the samples of a dataset are compartmentalized into two classes, a logistic regression strategy can be applied, which is a parametric method having parameters $\vec{w} \in \mathbb{R}$, which are called from now on *weights*.

The form of the resulting function $P(w)$ can be deduced with the following reasoning.

The conditional probability of having y (with $y \in \{0,1\}$) is:

$$P(y|x) = \frac{P(x|y)\cdot P(y)}{P(x)} \tag{3}$$

Note that here we're referring with $P(y|x)$ the posterior probability of having y given x. As $P(x)$ is unknown, we can get rid of it by observing that $P(\overline{y}|x) = 1 - P(y|x)$, and therefore by taking the log of the ratio of the two probabilities we have the log-odds quantity:

$$\log\frac{\mathrm{P(y \mid x)}}{\mathrm{P(\overline{y} \mid x)}} = \log\left(\frac{\mathrm{P(x \mid y)}}{\mathrm{P(x \mid \overline{y})}}\right) + \log\left(\frac{\mathrm{P(y)}}{\mathrm{P(\overline{y})}}\right) = w^\top \mathrm{x} \tag{4}$$

The assumption is that the log-odds (logits) quantity in the leftmost term can be expressed by a linear function, a linear combination of the parameters w and the values x. By using the first and rightmost term in 4 we find:

$$P(y|x) = \frac{1}{1+e^{-w^T x}} \coloneqq \sigma(w^T X), \tag{5}$$

where σ is the sigmoid function.

The probability of having a particular value y is described by the Bernoulli distribution:

$$P(Y=y) = P(Y=1)^Y + P(Y=0)^{1-Y} = \sigma(w^T x)^Y \cdot \sigma(w^T x)^{1-Y} \tag{6}$$

By applying the Maximum Likelihood approach to 6, we have that:

$$L(\vec{w}) \coloneqq ln \prod_{i=1}^{n} \left[\sigma\left(w^\top x_i\right)\right]^{y_i} \left[1-\sigma\left(w^\top x_i\right)\right]^{1-y} \tag{7}$$

which takes the final form as:

$$L(\vec{w}) = \sum_{i=1}^{n} y_i \log\left[\sigma\left(x_i; w\right)\right] + (1-y_i)\log\left[1-\sigma\left(x_i; w\right)\right] \tag{8}$$

The function 8 is the logistic regression function for the binary classification problem.

It is possible of course to l_1 or l_2 regularize it, in which case we need to add respectively $\lambda|\vec{w}|$ or $\lambda/2||\vec{w}||^2$. This is done to force a bound on the weight, which also prevents over-fitting on the training set.

By substituting eq.5 in 7 and reordering, we have the more elegant and compact version of the l_2 regularized logistic regression for binary classification:

$$L(\vec{w}) = \sum_{i=1}^{n} f_i(w) + \frac{\lambda}{2}||\vec{w}||^2 = \sum_{i=1}^{n} log(1+e^{-y_i x_i^T w}) + \frac{\lambda}{2}||\vec{w}||^2 \tag{9}$$

with $y_i \in \{-1,+1\}$.

Of course it's possible to define other types of Loss functions. In the numerical analysis in fact, not only the Loss (9) will be used but also a logistic regression with non-convex

†Department of Data Science, University of Padova, email: {marco.tronu}@studenti.unipd.it

regularization:

$$L(\vec{w}) = \sum_{i=1}^{n} log(1 + e^{-y_i x_i^T w}) + \lambda \sum_{j}^{d} \frac{w_j^2}{1 + w_j^2} \tag{10}$$

and a robust non-convex non-regularized loss-function:

$$L(\vec{w}) = \sum_{i=1}^{n} log\left(1 + \frac{1}{2}\left(y_i - \vec{w}\vec{x_i}\right)^2\right) \tag{11}$$

The following part of the report, will be focused on exploring the best way to find the non-trivial minimum of this function (as the function admits a trivial one for $\vec{w} = 0$

B. The need for an efficient method

Different methods have been proposed to find an estimate of the solution of 1, one being the Newton-Raphson one, in which the second derivative (i.e. the Hessian) of the function needs to be calculated. Such method is however computationally unfeasible when the data of interest starts becoming important in size, and therefore first-order methods are needed. Such methods rely on the idea of updating the weights iteratively using a scaling factor η_t called learning rate, and some kind of function $Q_t(w)$ which involves the computation of the first order derivatives of the Loss function, such that:

$$w_{t+1} = w_t - \eta_t \cdot Q_t(w_t) \tag{12}$$

II. Previous Works

A. Gradient Descent (GD)

The simplest way of updating the weights is to choose $Q_t(w_t) = \nabla L(w)$, such that:

$$w_{t+1} = w_t - \eta_t \cdot \nabla L(w) \tag{13}$$

which means that for each iteration (epoch) n gradients are calculated, with n being the number of observations in the dataset of interest. The following theorem guarantees, under a suitable choice of η, the linear convergence for the method 13:

Theorem 2.1 (Linear convergence of Gradient Descend): Suppose the function $f : R^n \longrightarrow R$ is convex and differentiable, and that its gradient is Lipschitz continuous with constant $L > 0$, i.e. we have that $||\nabla f(x) - \nabla f(y)|| \leq L \cdot ||x - y|| \ \ \forall x, y \in R^d$. Then if we run the iteration (13) k times with a fixed step size $t \leq 1/L$, it will yield a solution $f(w_k)$ which satisfies:

$$||f(w^{(k)} - f(w^*)|| \leq \frac{||w^{(0)} - w^*||}{2tk} \tag{14}$$

where w^* if the non-trivial optimum of f and $w^{(0)}$ is the starting point.

Even though the linear convergence is guaranteed by theorem 2.1, there are two main problems:

1) we can only choose a $\eta \leq 1/L$, else we don't have guarantee of convergence;
2) the computation of n gradients every iteration is in many cases not feasible as it is too costly computationally.

B. Stochastic Gradient Descent (SGD)

To overcome the computational cost of gradient descent, instead of updating the weight by a full-gradient computation, at each iteration a random sample $i \in \{1, ..., n\}$ is chosen such that:

$$w_{t+1} = w_t - \eta_t \cdot \nabla L_i(w) \tag{15}$$

The convergence rate of SGD is slower than that of GD, more specifically, it's *sublinear in the strongly convex case*, meaning:

$$||f(w^{(k)}) - f(w^*)|| \leq \frac{c}{k^\beta} \tag{16}$$

with $c, \beta > 0$.

Of course, the trade-off is that, at the cost of a slower convergence compared to the GD, we have a vastly cheaper computational cost per iteration. This trade-off has been analyzed for example in (Buttou,1998).

In practice, SGD in the form of (15) is not really useful because the method is subject to a too high variance, meaning that the performance is too sensitive to the specific per-iteration value of the gradient $\nabla f_i(w^{(k)})$. To reduce the variance and improve the convergence rate, more modern methods use a mini-batches approach, in which a random sample S, with $s \in \{1, ..., n\} \ \forall s \in S$ is taken, and the corresponding iteration gradient is averaged over this sample. Furthermore, a diminishing learning rate $\{\eta_k\}$ is taken in order to control further the variance:

$$w^{(k+1)} = w^k - \eta_k \nabla F(w^k, S) \tag{17}$$

where:

$$\nabla F(w^k, S) = \frac{1}{|S|} \sum_{i}^{|S|} \nabla f_i(w^k)$$

With these improvements, under strong convexity, SGD is guaranteed to have linear convergence:

Theorem 2.2 (Convergence of SGD): Let $f : R^d \longrightarrow R$ be a σ-strongly convex function with continuous Lipschitz gradient, assume that there exists $M > 0$ such that:

$$\mathbb{E}\left[\|\nabla F(x, \xi)\|^2\right] \leq M^2, \forall x \in \mathbb{R}^n$$

Then the SGD method (17) with $\eta_k = \frac{\gamma}{k+\delta}$ and $\gamma > \frac{1}{2\sigma}$ satisfies:

$$\mathbb{E}\left[f\left(x_k\right) - f\left(x^*\right)\right] \leq \frac{LC(\gamma)}{2(k + \delta)}$$

where $C(\gamma)$ is:

$$C(\gamma) = \max\left\{\gamma^2 M^2 (2\sigma\gamma - 1)^{-1}, (1 + \delta)\left\|x_1 - x^*\right\|^2\right\}$$

A problem of the mini-batches version of SGD is that one still needs to hand picks the sequence, and the performance is very sensitive to this choice.

C. Other methods

More sophisticated algorithms have more recently emerged which combines some deterministic and stochastic aspects to reduce variance of the steps. Some of them are SAG/SAGA, SDCA, SVRG, DIAG, MISO and S2GD, each one enjoying a

faster convergence rate than SGD while using a fixed learning rate parameter η. In this paper, the focus will be put mainly on three relatively recent methods for problem (1): SARAH, SPIDER and SPIDER-Boost

III. SARAH: StochAstic Recursive gradient AlgorithM

SARAH was proposed in 2017 by Lam M. Nguyen et al., and aims to combine some of the good properties of existing algorithms, such as SAGA and SVRG, while providing some further improvements. More specifically:

- Similarly to SVRG, SARAH's iterations are divided into the outer loop where a full gradient is computed and the inner loop where only stochastic gradient is computed. Unlike the case of SVRG, the steps of the inner loop of SARAH are based on accumulated stochastic information.
- SARAH uses a constant learning rate, whose size is larger than that of SVRG. However, unlike SAG/SAGA but similar to SVRG, SARAH does not require a storage of n past stochastic gradients.
- SARAH possesses linear convergence rate (in the strongly convex case) for the inner loop of SARAH, the property that SVRG does not possess. Furthermore, the variance of the steps inside the inner loop goes to zero, thus SARAH is theoretically more stable and reliable than SVRG.

Algorithm 1 SARAH

Parameters: the learning rate $\eta > 0$ and the inner loop of size m.
Initialize: $\tilde{w}_0$
for $s = 1, 2, ..., epochs$ **do**
  $w_0 = \tilde{w}_{s-1}$
  $v_0 = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w_0)$
  $w_1 = w_0 - \eta v_0$
  for $t = 1, ..., m-1$ **do**
    Sample i_t uniformly at random from {1,...,n}
    $v_t = \nabla f_{i_t}(w_t) - \nabla f_{i_t}(w_{t-1}) + v_{t-1}$
    $w_{t+1} = w_t - \eta v_t$
  end for
  Set $\tilde{w}_s = w_t$ with t chosen uniformly at random from {0,1,...,m}
end for

Note that at each iteration (epoch), a combination of a full gradient update (like in GD) with the addition of m updates considering random samples from the data is considered. This means that for each epoch, the cost is $\mathcal{O}(n+m)$ in terms of the calls to the gradient function. Note also that if $m = 1$ is chosen, SARAH reduces to the GD algorithm. One important property of SARAH which hugely discriminates it from SVRG, is that the latter has a non-convergent inner loop. This means that if we run the inner loop of SVRG many times, the variance of the steps does not reduce. On the other side, it goes to zero in the case of SARAH.

1) Strong convex case: This is demonstrated under the assumptions of L-smoothness of the gradient and μ-strong convexity for each f_i in the Loss function, which yield to the following result:

Theorem 3.1: Consider v_t defined as in SARAH alg.III and choose $\eta \leq 2/(\mu + L)$. Then the following bound holds $\forall t \geq 1$:

$$\begin{aligned}\mathbb{E}\left[\|v_t\|^2\right] &\leq \left(1 - \frac{2\mu L\eta}{\mu + L}\right)\mathbb{E}\left[\|v_{t-1}\|^2\right] \\ &\leq \left(1 - \frac{2\mu L\eta}{\mu + L}\right)^t \mathbb{E}\left[\|\nabla P(w_0)\|^2\right]\end{aligned}$$

Therefore under the previous assumptions we're guaranteed a linear convergence rate of the inner loop.

For future reference, the following iteration complexity analysis aims to bound the number of outer iterations T (or total number of stochastic gradient evaluations) which is needed to guarantee that $\|\nabla P(w_\mathcal{T})\|^2 \leq \epsilon$. In this case we will say that $w_\mathcal{T}$ is a ϵ-accurate solution.

2) General convex case: Under not strong convex assumptions, it's possible to show that the inner loop still is guaranteed to converge, but in this case, contrary to theorem3.1, sublinearly.

Theorem 3.2: Consider the SARAH algorithm with $\eta \leq 1/L$. Then for any $s \leq 1$ we have:

$$\begin{aligned}\mathbb{E}\left[\|\nabla P(\tilde{w}_s)\|^2\right] \leq \frac{2}{\eta(m+1)}\mathbb{E}\left[P(\tilde{w}_{s-1}) - P(w^*)\right] \\ + \frac{\eta L}{2-\eta L}\mathbb{E}\left[\|\nabla P(\tilde{w}_{s-1})\|^2\right]\end{aligned} \tag{18}$$

The result in eq.(18) implies that, under the choice $\eta = \sqrt{\frac{2}{L(m+1)}}$, with m, the size of the inner loop, such that $\eta \leq 1/L$, we can derive the following **sublinear convergence result**:

$$\begin{aligned}&\mathbb{E}\left[\|\nabla P(\tilde{w}_s)\|^2\right] \\ &\leq \sqrt{\frac{2L}{m+1}}\mathbb{E}\left[P(\tilde{w}_{s-1}) - P(w^*) + \|\nabla P(\tilde{w}_{s-1})\|^2\right]\end{aligned} \tag{19}$$

Finally, we can estimate the convergence of SARAH with multiple outersteps, by simply using Theorem 18 many times (for the total number of epochs), so that we have the following:

$$\mathbb{E}\left[\|\nabla P(\tilde{w}_s)\|^2\right] - \Delta \leq \alpha^s\left(\|\nabla P(\tilde{w}_0)\|^2 - \Delta\right) \tag{20}$$

where:

$$\delta_k = \frac{2}{\eta(m+1)}\mathbb{E}\left[P(\tilde{w}_k) - P(w^*)\right], k = 0, 1, \ldots, s-1,$$

$\sigma = max_{0\leq k\leq s-1}\delta_k$, $\Delta = \delta\left(1 + \frac{\eta L}{2(1-\eta L)}\right)$, and $\alpha = \frac{\eta L}{2-\eta L}$

It directly follows from the Theorem 20 that by choosing $\Delta = \epsilon/4, \alpha = 1/2, \eta = 2/(3L)$, and $m = \mathcal{O}(1/\epsilon)$, then the total complexity to achieve an ϵ-accuracy solution is $\mathcal{O}((n + (1/\epsilon))log(1/\epsilon)$

A. Strongly convex case

Under stricter assumptions, compared to result (20), of strong convexity, it's possible to show that

$$\mathbb{E}\left[\left\|\nabla P\left(\tilde{w}_s\right)\right\|^2\right] \leq (\sigma_m)^s \left\|\nabla P\left(\tilde{w}_0\right)\right\|^2 \tag{21}$$

where:

$$\sigma_m \overset{\text{def}}{=} \frac{1}{\mu\eta(m+1)} + \frac{\eta L}{2-\eta L} < 1$$

is the convergence rate.

An important remark is that:

- any $\eta < 1/L$ will work for SARAH;
- $\sigma_m > \alpha_m$, where α_m is the convergence rate of SVRG

Finally, we conclude with the following:

Corollary of Theorem20 Fix $\epsilon \in (0,1)$, run SARAH with $\eta = 1/(2L)$ and $m = 4.5\kappa$ ($\kappa = L/\mu$ by definition) for $\mathcal{T}$ iterations where $\mathcal{T} = \left[log(||\nabla P(\tilde{w_0})||^2\epsilon)/log(9/7)\right]$, then we can obtain the total complexity of SARAH, to achieve the ϵ-accuracy solution, as $\mathcal{O}((n+\kappa)log(1/\epsilon)$

B. SARAH+: a practical variant of SARAH

It is known for SVRG that different sizes m of the inner loop impact sensibly the performance of the method. That's also the case with SARAH. While it's also known that $m \sim \mathcal{O}(\kappa)$ [1], it's still not clear which should be the optimal choice. That's why the need of a more applicable method led to the implementation of SARAH+:

Algorithm 2 SARAH+

Parameters: the learning rate $\eta > 0$, $0 < \gamma < 1$, and the maximum loop size m.
Initialize: $\tilde{w}_0$
for $s = 1, 2, ..., epochs$ **do**
  $w_0 = \tilde{w}_{s-1}$
  $v_0 = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w_0)$
  $w_1 = w_0 - \eta v_0$
  $t = 1$
  while $\left\|v_{t-1}\right\|^2 > \gamma \left\|v_0\right\|^2$ **and** $t < m$ **do**
    Sample i_t uniformly at random from {1,...,n}
    $v_t = \nabla f_{i_t}(w_t) - \nabla f_{i_t}(w_{t-1}) + v_{t-1}$
    $w_{t+1} = w_t - \eta v_t$
    $t = t + 1$
  end while
  Set $\tilde{w}_s = w_t$
end for

The parameter γ acts as a threshold: the closer it is to 1, the sooner the inner loop gets interrupted. For a value of 1, the algorithm is reduced again to a standard Gradient Descent. A value of 0 instead always passes the first condition in the while, so in that case the upper bound will be always given by m. The main idea is that γ quantifies a trade-off between the fast (computationally speaking) sublinear convergence of the inner loop and the much costlier linear convergence of the outer loop. How γ impacts the performance will be explored in the numerical analysis.

[1] In practice, when n is large, $P(w)$ is often considered as a regularized Empirical Loss Minimization problem with regularization parameter $\lambda = \frac{1}{n}$, then $\kappa \sim \mathcal{O}(n)$

IV. SPIDER

A. Stochastic Path-Integrated differential Estimator (SPIDER)

The algorithm was proposed by Cong Fang et. al. in 2017 as the first algorithm which achieved $\mathcal{O}\left(\min\left(n^{1/2}\epsilon^{-2}, \epsilon^{-3}\right)\right)$ in both upper and lower (finite-sum only) bound for finding first-order stationary points for the optimization problem 1.

One characteristic aspect of SPIDER, is the normalized gradient update as in NGD:

$$w^{k+1} = w^k - \eta \frac{\nabla f\left(w^k\right)}{\|\nabla f\left(w^k\right)\|} \tag{22}$$

B. The algorithm

Algorithm 3 SPIDER-SFO

Parameters: $w^0, q, S_1, S_2, n_0, \epsilon, \tilde{\epsilon}$
for $k = 0, 1, ..., K$ **do**
  if $mod(k, q) = 0$ **then**
    Draw S_1 sample (or compute the full gradient for the finite-sum case)
    $v_k = \nabla f_{S_1}\left(w^k\right)$
  else
    Draw S_2 samples
    $v_k = \nabla f_{S_2}\left(w^k\right) - \nabla f_{S_2}\left(w^{k-1}\right) + v^{k-1}$
  end if
  OPTION I
  if $\|v^k\| \leq 2\tilde{\epsilon}$ **then**
    return w^k
  else
    $\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \cdot \left(\mathbf{v}^k / \left\|\mathbf{v}^k\right\|\right)$ where $\eta = \frac{\epsilon}{Ln_0}$
  end if
  OPTION II
  $\mathbf{w}^{k+1} = \mathbf{w}^k - \eta^k \mathbf{v}^k$ where $\eta^k = \min\left(\frac{\epsilon}{Ln_0\|\mathbf{v}^k\|}, \frac{1}{2Ln_0}\right)$
end for
OPTION I: Return w^K
OPTION II: Return $\tilde{w}$ chosen unifromly at random from $\{w^k\}_{k=0}^{K-1}$

V. SpiderBoost

A. The needs for a SPIDER improvement

While SARAH achieves an overall $\mathcal{O}(\epsilon^{-4})$ SFO complexity, SPIDER achieves an overall $\mathcal{O}(\min\{n^{1/2}\epsilon^{-2}, \epsilon^{-3}\})$ SFO with a normalized gradient update using a stepsize $\eta = \mathcal{O}(\eta/L)$

The main problem of SPIDER however, is that it requires a very restrictive η which prevents it to making more progress even when it's possible, and it's not easy to generalize it to a proximal algorithm as it requires a very small per-iteration increment $||x_{k+1} - x_k|| = \mathcal{O}(\epsilon/L)$. The main advantages of SpiderBoost over SPIDER are the following:

- it allows for a much larger stepsize $\eta = \mathcal{O}(1/L)$ but it retains the same state-of-the-art complexity as SPIDER;
- it can be naturally generalized to a proximal algorithm;

B. SpiderBoost algorithm

Algorithm 4 Prox-SpiderBoost

Parameters: the learning rate $\eta = \frac{1}{2L}$, $q > 0$, K, $|S|$.
Initialize: $\tilde{w}_0$
for $s = 1, 2, ..., epochs$ **do**
  $w_0 = \tilde{w}_{s-1}$
  for $k = 0, 1, ..., K-1$ **do**
    if $mod(k, q) = 0$ **then**
      $v_k = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w_k)$
    else
      Draw $|S|$ samples with replacement from the dataset.
      $v_k = \frac{1}{|S|}\sum_{i \in S}\left[\nabla f_i(w_k) - \nabla f_i(w_{k-1}) + v_{k-1}\right]$
    end if
    $w_{k+1} = prox_{\eta h}(w_k - \eta v_k)$
  end for
  Set $\tilde{w}_s = w_\xi$ where $\xi \overset{Unif}{\sim} \{0, ..., K-1\}$
end for

Some important notes are the following:

1) The shape of the loss function is

$$P(w) = \frac{1}{n}\sum_{i}^{n} f_i(w) + h(w) \tag{23}$$

where h is a simple convex but possibly nonsmooth regularizer.

2)

$$prox_{\eta h}(w) \coloneqq arg\min_{u \in \mathcal{X}}\{h(u) + \frac{1}{2\eta}||u - w||^2\} \tag{24}$$

Now we notice then Prox-SpiderBoost is a generalization of SpiderBoost, i.e. if $h(w) = 0 \forall w$, then we have $prox_{\eta h}(w_k - \eta v_k) = w_k - \eta v_k$. So from now on referring to SpiderBoost equals referring to Prox-SpiderBoost with $h(w) = 0 \forall w$

Theorem 5.1: Under the assumption of the Loss function being bounded below, i.e. $P(w^*) = P^* \coloneqq inf_{x \in \mathcal{R}^d} P(w) > -\infty$, apply Prox-SpiderBoost to solve problem 1 with parameters $q = |S| = \sqrt{n}$ and stepsize $\eta = \frac{1}{2L}$. Then, the corresponding output w_ξ satisfies $\mathbb{E}\,||\nabla f(x_\xi)|| \leq \epsilon$ provided that the total number K of iterations satisfies

$$K \geq \mathcal{O}\left(\frac{L \cdot (P(w_0) - P^*)}{\epsilon^2}\right)$$

Moreover, the overall SFO complexity is $\mathcal{O}(\sqrt{n}\epsilon^2 + n)$

C. Accelerated Prox-SpiderBoost

To further improve the alg.5

One important observation on Prox-SpiderBoost-M is that the momentum scheme design applies the same proximal gradient term $prox_{\lambda_k h}(w_k - \lambda_k v_k)$ to update both variables w_{k+1} and y_{k+1} and therefore requires less computation compared

Algorithm 5 Prox-SpiderBoost-M

Input: $q, K \in \mathcal{N}, \{\lambda_k\}_{k=1}^{K-1}, \{\beta_k\}_{k=1}^{K-1} > 0, y_0 = w_0 \in \mathcal{R}^d$
Initialize: $\tilde{w}_0$
for $s = 1, 2, ..., epochs$ **do**
  $w_0 = \tilde{w}_{s-1}$
  for $k = 0, 1, ..., K-1$ **do**
    $z_k = (1 - \alpha_{k+1})y_k + \alpha_{k+1} w_k$
    if $mod(k, q) = 0$ **then**
      $v_k = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w_k)$
    else
      Draw $|S|$ samples with replacement from the dataset.
      $v_k = \frac{1}{|S|}\sum_{i \in S}\left[\nabla f_i(w_k) - \nabla f_i(w_{k-1}) + v_{k-1}\right]$
    end if
    $w_{k+1} = prox_{\lambda_k h}(w_k - \lambda_k v_k)$
    $y_{k+1} = z_k - \frac{\beta_k}{\lambda_k} w_k + \frac{\beta_k}{\lambda_k}\,\mathrm{prox}_{\lambda_k h}\left(w_k - \lambda_k v_k\right)$
  end for
  Output: z_ζ where $\zeta \overset{Unif}{\sim} \{0, ..., K-1\}$
end for

to the momenthum scheme of proximal SGD and proximal SVRG:

$$w_{k+1} = \mathrm{prox}_{\lambda_k h}\left(w_k - \lambda_k v_k\right), \quad y_{k+1} = \mathrm{prox}_{\beta_k h}\left(z_k - \beta_k v_k\right)$$

Let's define:

$$G_\eta(x) \coloneqq \frac{1}{\eta}\left(x - \mathrm{prox}_{\eta h}(x - \eta \nabla f(x))\right)$$

Prox-SpiderBoost-M can be proven to yield the following result:

Theorem 5.2: Under the assumption of the Loss function being bounded below, i.e. $P(w^*) = P^* \coloneqq \inf_{x \in \mathcal{R}^d} P(w) > -\infty$, apply Prox-SpiderBoost to solve problem (1) with parameters $q = |S| = \sqrt{n}$ and stepsize $\eta = \frac{1}{8L}$ and $\lambda_k \in [\beta_k, (1 + \alpha_k)\beta_k]$. Then, the corresponding output w_ξ satisfies $\mathbb{E}\left\|G_{\lambda_\zeta}\left(z_\zeta, \nabla f\left(z_\zeta\right)\right)\right\| < \epsilon$ for any $\epsilon > 0$ provided that the total number K of iterations satisfies

$$K \geq \mathcal{O}\left(\frac{L \cdot (P(w_0) - P^*)}{\epsilon^2}\right)$$

Moreover, the overall SFO complexity is $\mathcal{O}(\sqrt{n}\epsilon^2 + n)$

VI. Numerical Analysis

A. Tools used

Python3.9.6 was the tool through which I did all the numerical experiments. Python however, it's not a fast programming language when it comes to heavy computation, and therefore needs proper libraries to fully exploit its simplicity while keeping the performance comparable to much faster languages like C and C++. Of course, NumPy vectorization allows the user to speed computation a lot, however I found unfeasible to use it in most algorithms I had to implement for the project, in which there are nested loops which repeats thousands of times, where a variable gets iteratively updated.

The two "third-parthy" libraries I'm mostly aware of, are Numba and Cython.

Cython is a programming language that is part Python and part C/C++, it can be compiled into a python extension and/or an executable. If one is familiar with Python it is reasonably easy to understand Cython code, it largely just has a few "boiler-plate" code blocks along with a few static type declarations (familiar to those who know C/C++/related languages). Since there are no dynamic types (in well written Cython code) and it is compiled typically the resulting code is orders of magnitude faster than Python. Compared to vectorised NumPy there may not be a significant improvement but this depends on the exact implementation.

Numba uses the concept of a "just in time" compiler (JIT). Essentially this means that code is compiled "on the fly" during runtime instead of requiring compilation prior to execution. Numba compiles the python code using a LLVM compiler. The syntax is very simple and most of the time just requires a simple decorator on a Python function. It also allows for parallelisation and GPU computation very simply (typically just a "target = 'cuda'" type statement in a decorator). Much fewer boilerplate code, if not any, is required to switch between pure Python and Numba.

From my understanding, there's no real performance difference between those two libraries, however it's possible that Cython provides more flexibility than Numba, at the cost of more boilerplate code.

I did not look much into Cython, because I found Numba to work already almost always flawlessly, providing ***orders of magnitude improvement*** in speed. As an example, I wrote a quite optimized Python implementation of the Gradient Descent algorithm (13), using only NumPy builtin vectorized functions. This is of course the best-case as for arguably all the other more complex algorithms this is not possible. However even in the best-case, for a dataset of shape $(40000, 22)$, with 1000 iterations of GD, Numba was still more than twice faster with $\sim 8s$ versus $\sim 18s$ of the pure Python implementation. In the other algorithms where pure Numpy vectorized function can not be directly implemented, the time difference easily reaches two orders of magnitude. Therefore, I conducted all work in Python implementing Numba for all the algorithms. As a side note, Numba is still incompatible with Numpy 1.21.2, meaning that if Numba is imported when Numpy 1.21.2 is present in the environment, it will throw an error and won't allow the import. However, at the cost of the risk of some sketchy stuff happening, it's possible to modify the allowed version of Numpy by modifying a line in the $\_\_init\_\_.py$ file where Numba is located.

B. Datasets and task

All the datasets used for all the experiments are available here[2], and are:

- ijcnn1;
- a9a;
- w8a;
- rcv1.

[2]https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/

The problem I focused on is the optimization problem 1 with $f(w)$ taking the form of eq.(9), eq.(10), eq.(11).

C. SARAH and SARAH+

Firstly, I wanted to see how sensible is SARAH to a particular choice of the inner loop size m.

Fig.1 shows that indeed an optimal choice of m is required for the best convergence rate. Note that in both plots, the number of effective passes displayed on the x-axis is $n * m$. Notice that $\eta = 0.1$ seems to be a good choice for all three datasets. For bigger values, most of the times what happens is that the loss diverges for the first iterations and then finally decreases, as one can see in the left plot, for the dot-dashed lines (corresponding to a $\eta = 0.5$. This was observed also for the other types of loss. Once a proper value of η is choosen, the difference in performance between different choices of m is also quite apparent: observe for example the green lines corresponding to $m/n = 2$. By far, this choice of m leads to a much slower convergence. This is because the computationally cheap inner loop advantage is out-weighted by its sublinear convergence. What I also observed, is that a higher learning rate η requires a smaller inner loop size m. This is reasonable, because with a higher η the jumps towards the optimum are bigger and therefore we need more "precision" by doing more full gradient computations along the way.

Since SARAH+ was created with the goal of having all the benefits of SARAH but with less effort required in tweaking the parameter m, one interesting thing to explore is how the performance changes depending on the parameter γ.

Fig.2 corroborate the intuition behind the role of γ in SARAH+. We notice two things:

- reducing the value of γ doesn't equate to improving the convergence rate;
- a fixed value of γ yields consistently much better results compared to keeping the value of m fixed

Perhaps a bit differently to what stated in the SARAH paper, a value of γ of $1/32$ usually yields a fast enough convergence among different data-sets. However the value $1/8$ suggested in the paper yields anyway usually a good result.

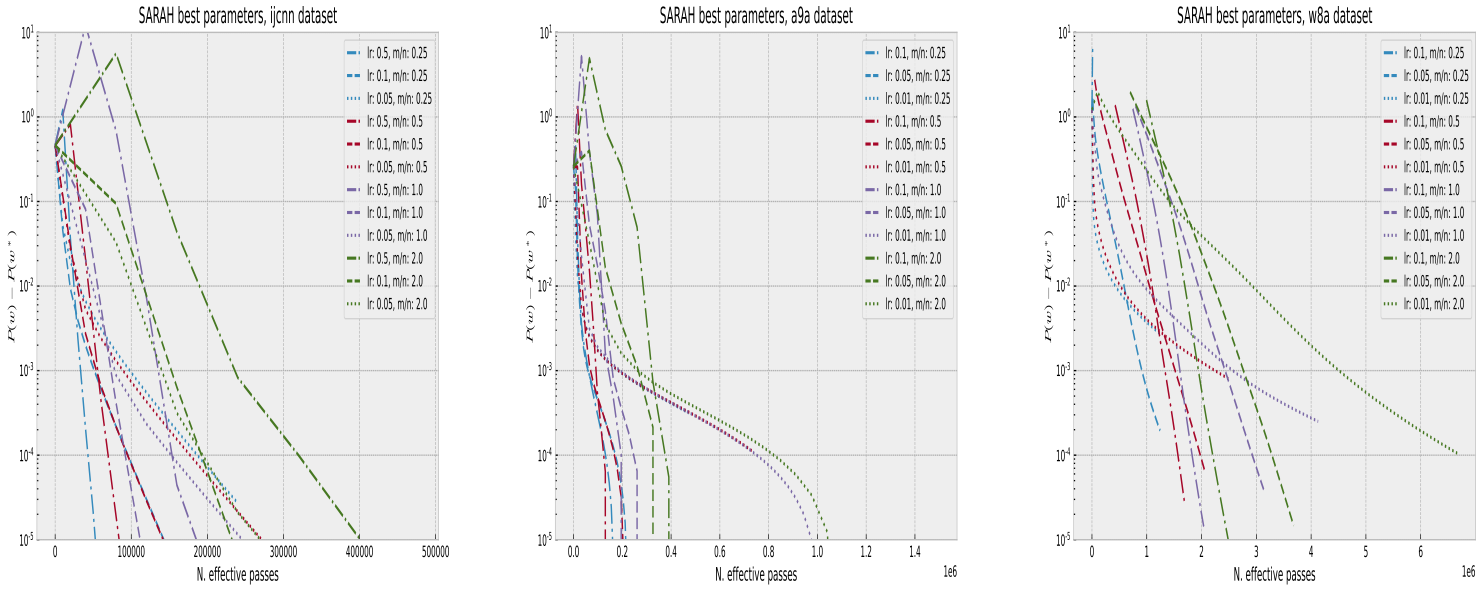
To show that a good value of γ naturally selects, epoch by epoch, the optimal size of the inner loop, thus providing a faster convergence, a comparison between SARAH and SARAH+ is shown in fig.3,4 and 5

On all the datasets, it's noticeable that indeed SARAH+ seems to have an edge over standard SARAH.

D. Spider, SpiderBoost and SpiderBoost-M

Since SpiderBoost and SpiderBoost-M are proposed as an improvement over Spider, their effectiveness is explored.

The plots in fig.6,7,8 highlight that, first of all, SpiderBoost is indeed an improvement over Spider in most cases, and doesn't exhibit the convergence problematics Spider shows. Furthermore, they also prove that the accelerated version of SpiderBoost converges faster (apart for the w8a dataset), thus legitimating the addition of the momentum scheme.



(a) ijcnn dataset (b) a9a dataset (c) w8a dataset

Fig. 1: Loss residuals for different combination of m and η, $l2$-regularized loss.

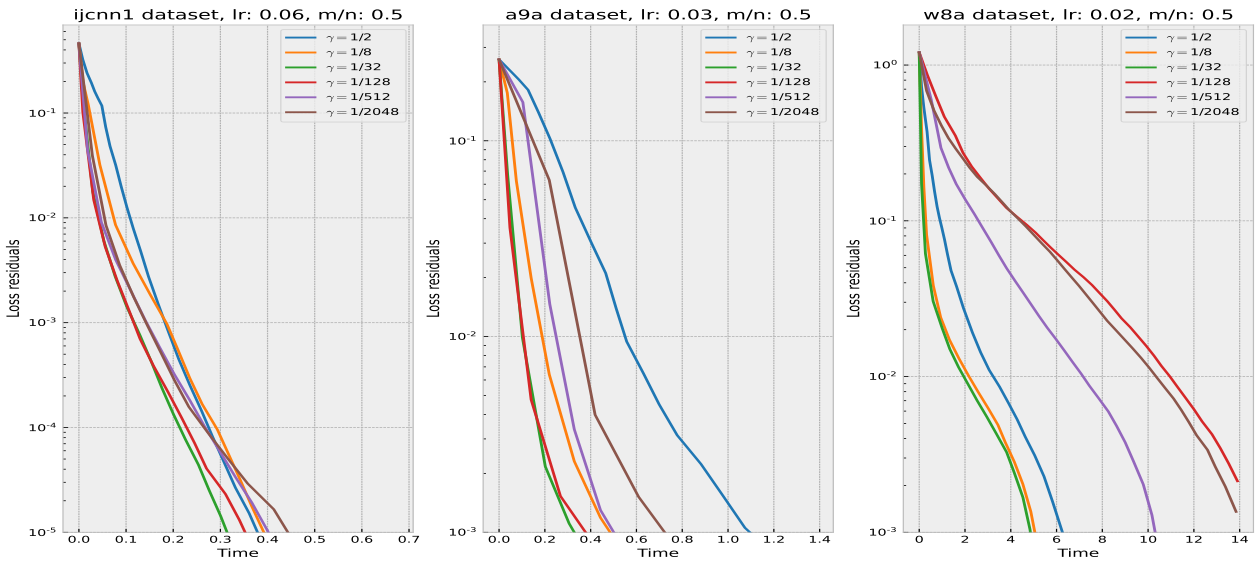


Fig. 2: Loss residuals for several γ values,$l2$-regularized logistic regression

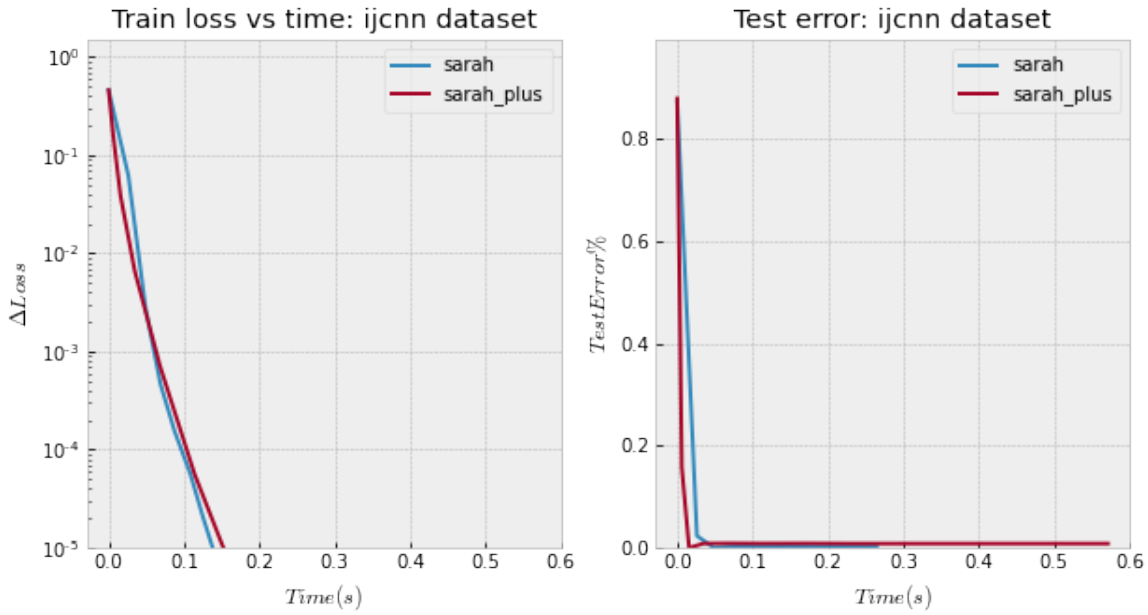


Fig. 3: Sarah vs Sarah+, $\gamma = 1/16$, $m = 0.5n$, $lr = 0.1$

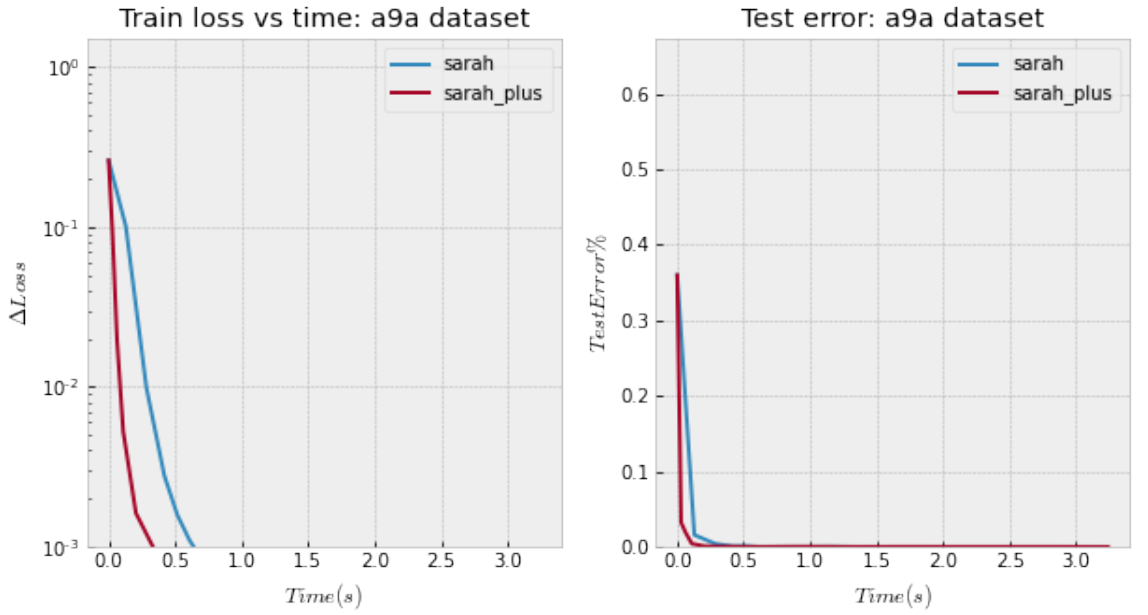


Fig. 4: Sarah vs Sarah+, $\gamma = 1/16$, $m = 0.5n$, $lr = 0.05$

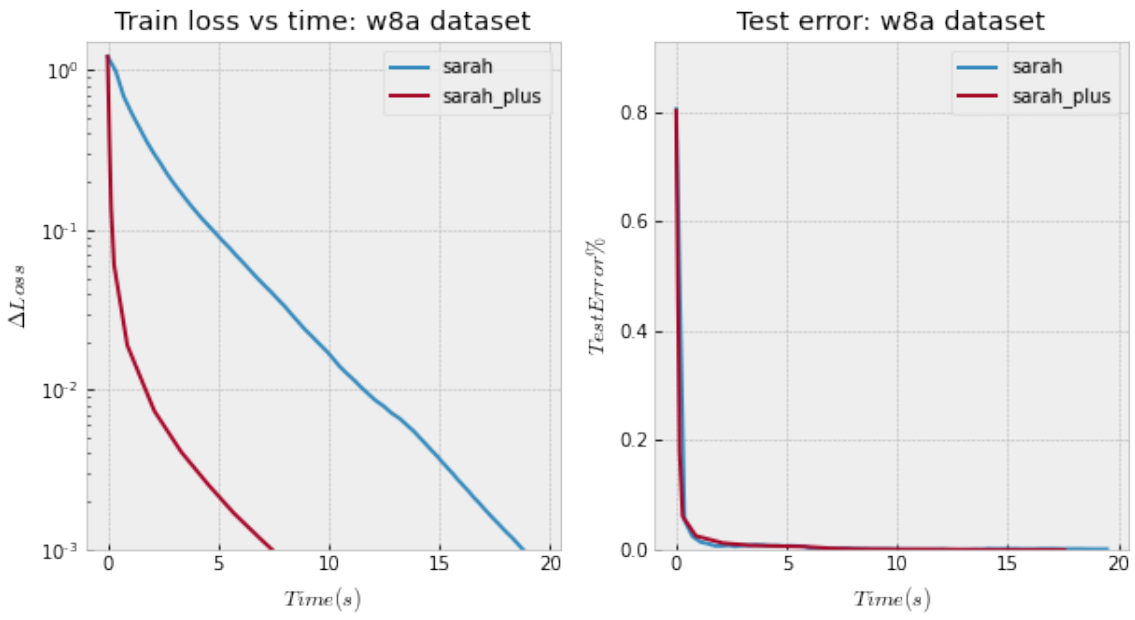


Fig. 5: Sarah vs Sarah+, $\gamma = 1/16$, $m = 0.5n$, $lr = 0.05$

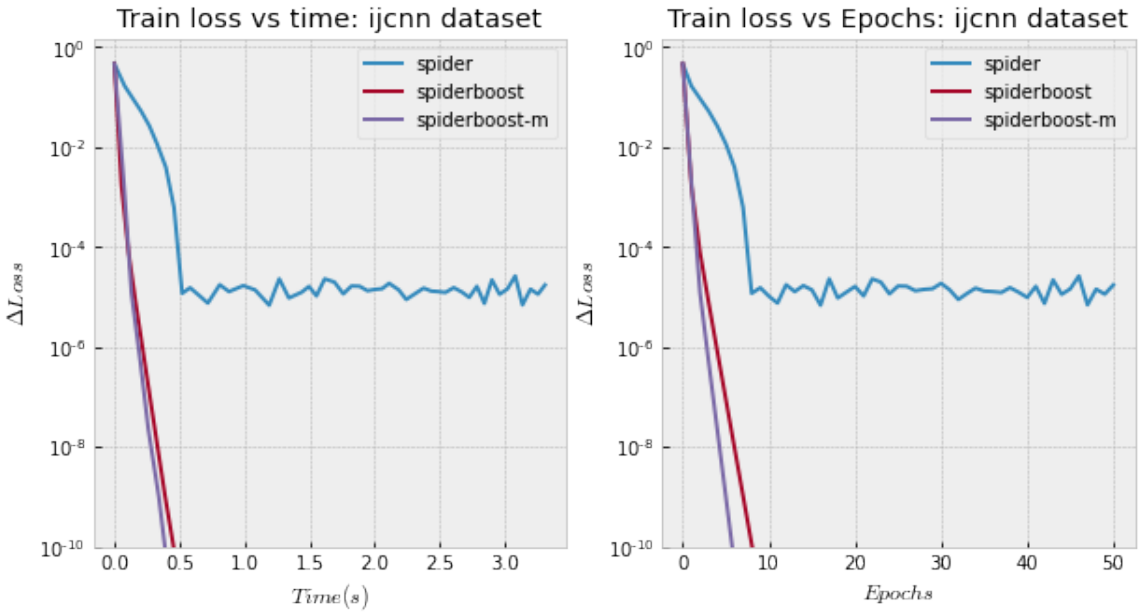


Fig. 6: Spider vs SpiderBoost vs SpiderBoost-M, ijcnn dataset

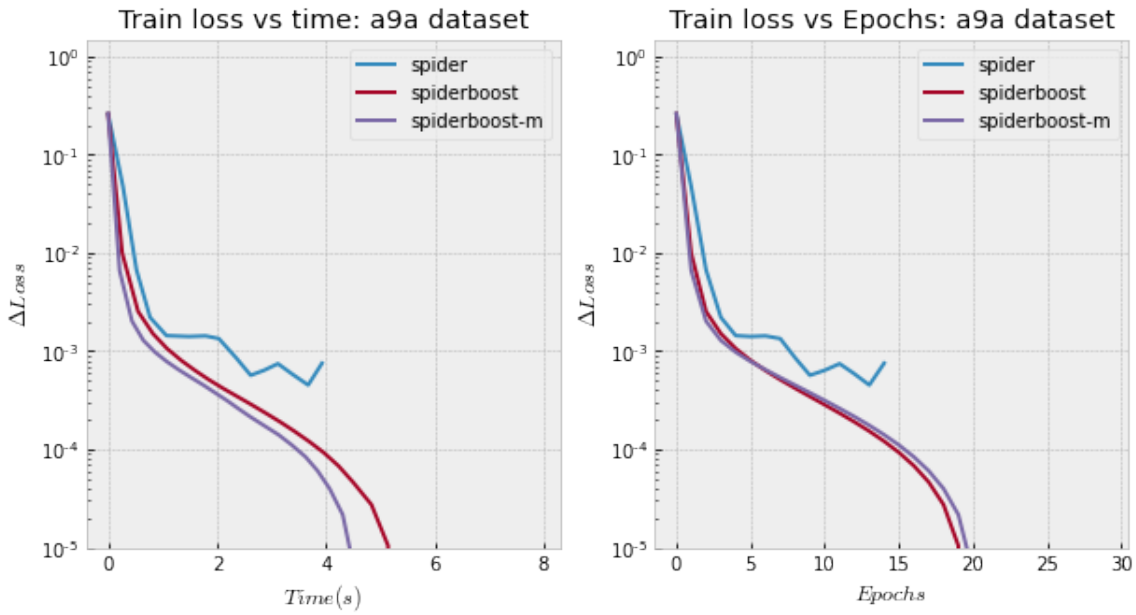


Fig. 7: Spider vs SpiderBoost vs SpiderBoost-M, a9a dataset

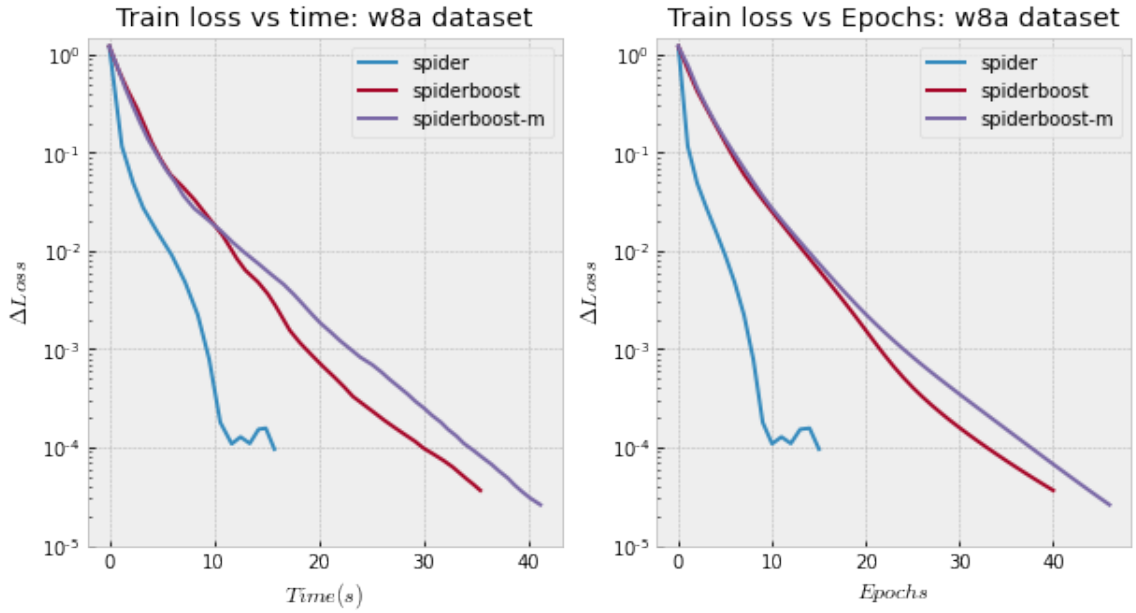


Fig. 8: Spider vs SpiderBoost vs SpiderBoost-M, w8a dataset

E. Comparison among all methods

Finally, the performance of several algorithms is compared.

In all the simulations, I choose a mini-batch size of 256, a value of q (for the spider algorithms) of n/256, and a value of K of 2q. The learning rate was chosen to be 0.05. After running the algorithms a certain amount of times on different datasets and different loss functions, from my understanding **SARAH+ performed consistently very good, with very little, if not any, tweaking**. On the contrary, **SpiderBoost-M came several times on top, but it's much harder to find the optimal parameters**. In any way I think it's quite fair to say that indeed **SpiderBoost-M and SARAH+ most times performed better than the competition**, which is in line with the theoretical findings. Most of the times, SpiderBoost, SpiderBoost-M and SARAH+ are the top 3 most better performing algorithms, however there are times in which these are outperformed by SPIDER or even SVRG. I surely can not exclude that in those cases it was my mistake in choosing unoptimal parameters.

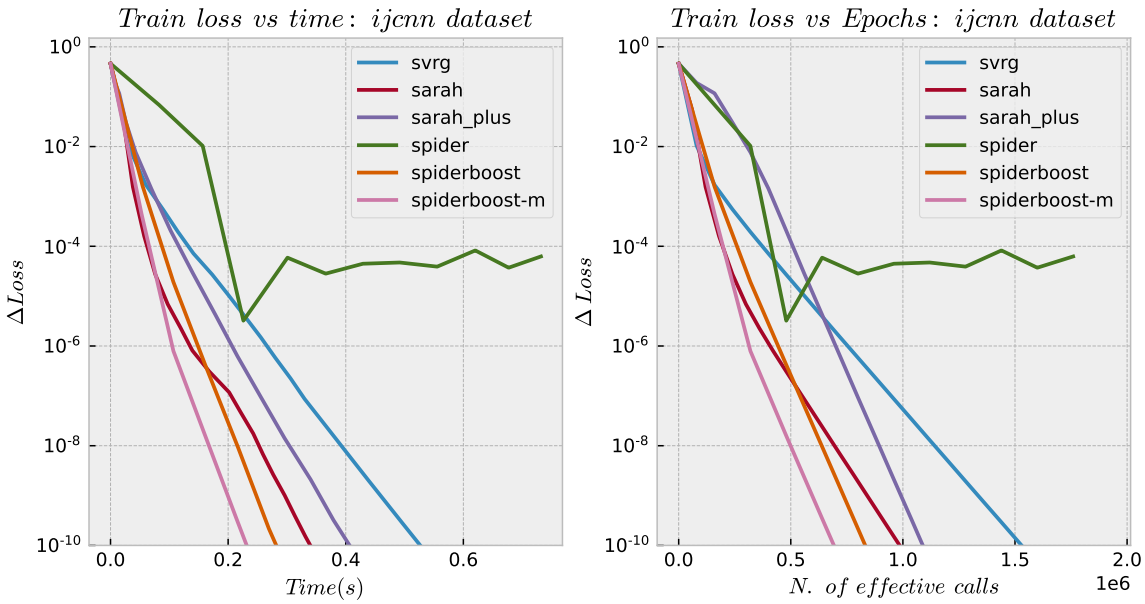


Fig. 9: ijcnn dataset, $l2$-regularized logistic regression

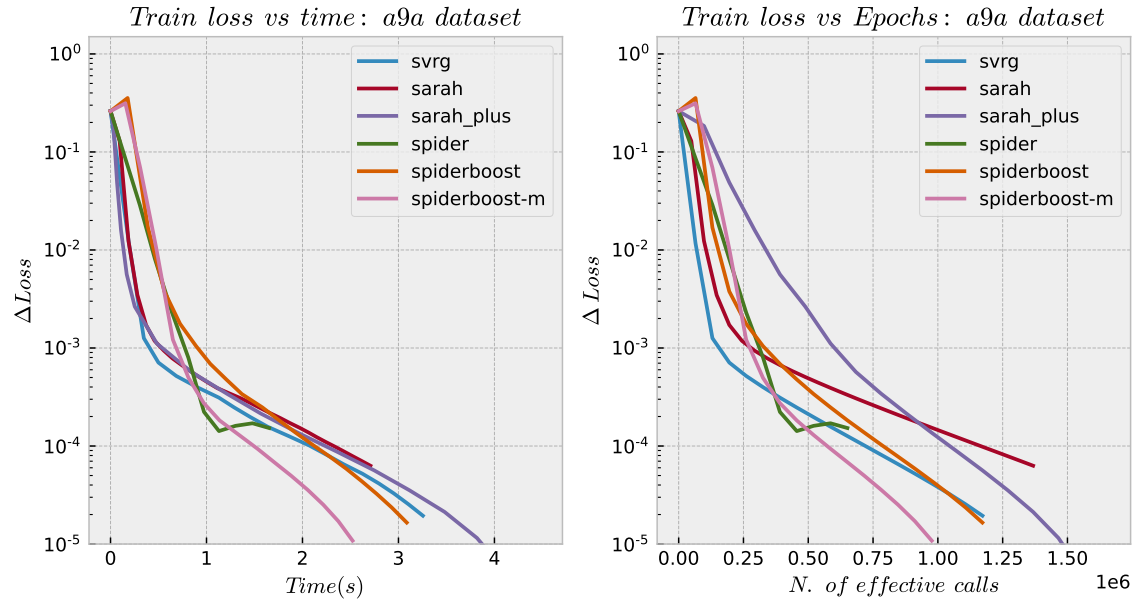


Fig. 10: a9a dataset, $l2$-regularized logistic regression

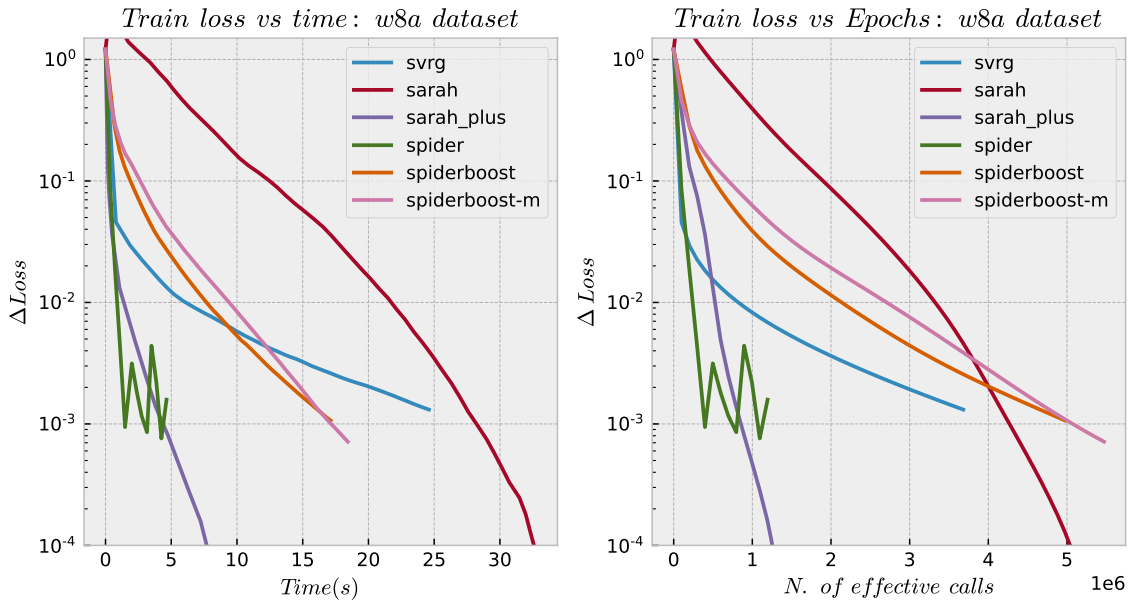


Fig. 11: w8a dataset, $l2$-regularized logistic regression

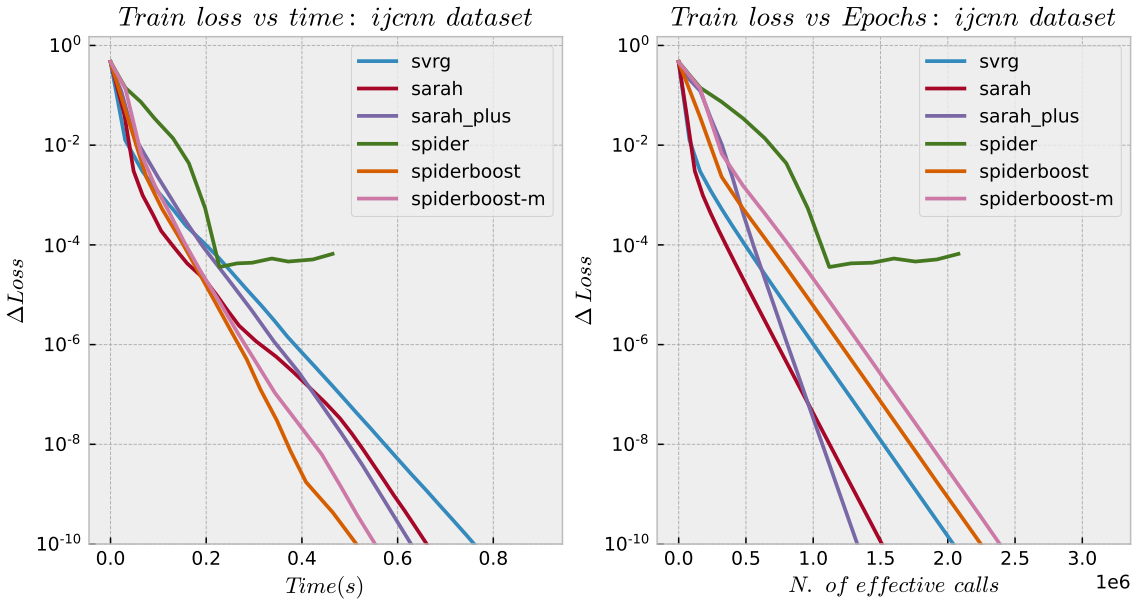


Fig. 12: ijcnn, non-convex regularized logistic regression

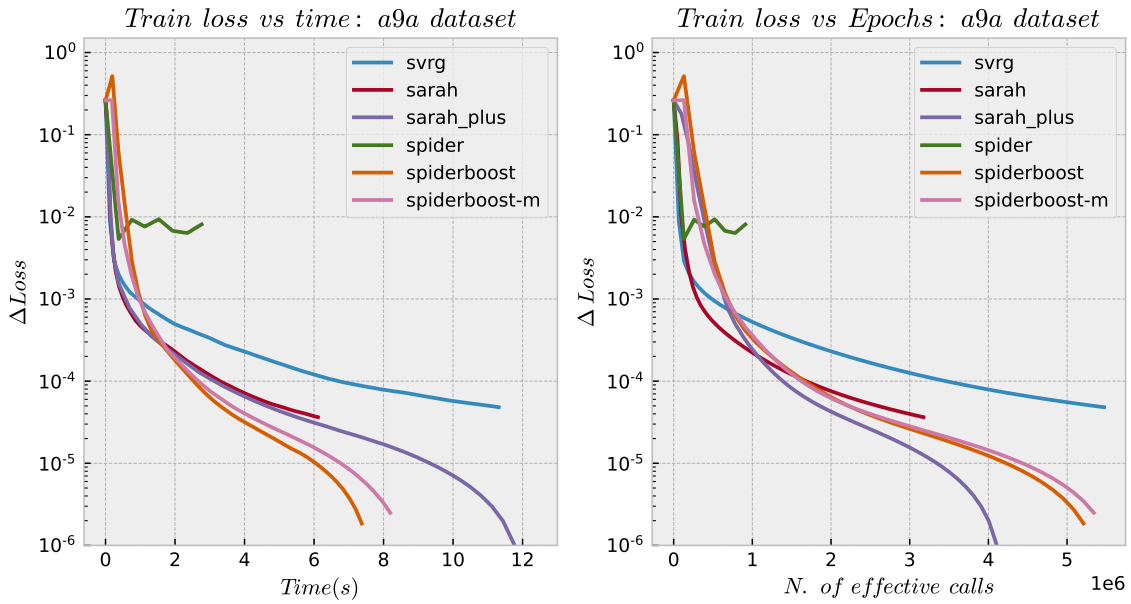


Fig. 13: a9a, non-convex regularized logistic regression

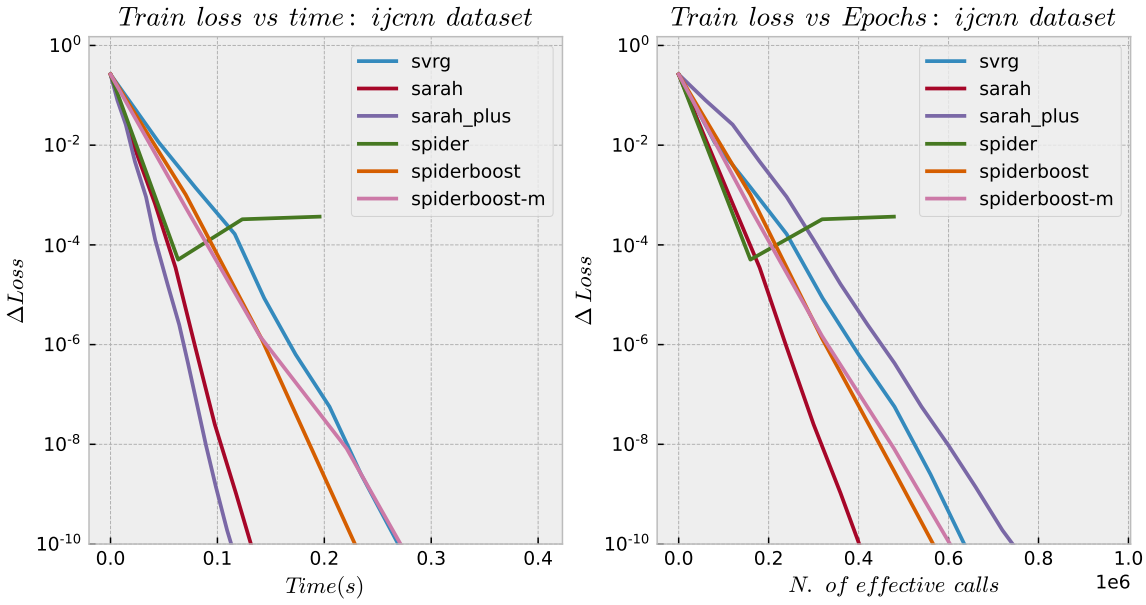


Fig. 14: ijcnn, robust non regularized loss function (type 3)

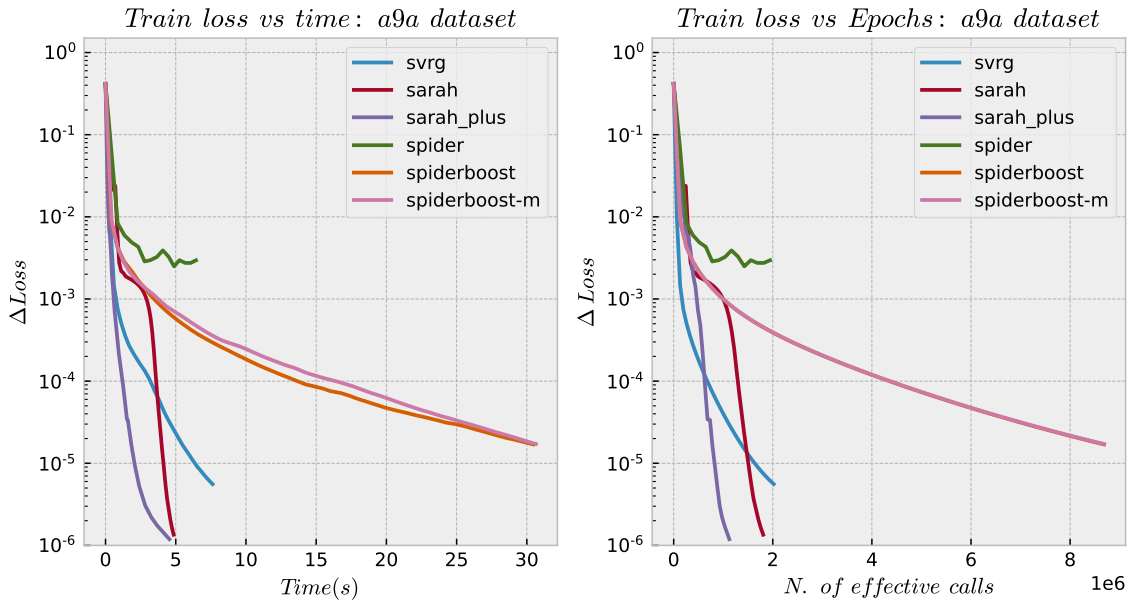


Fig. 15: a9a dataset, robust non regularized loss function (type 3)

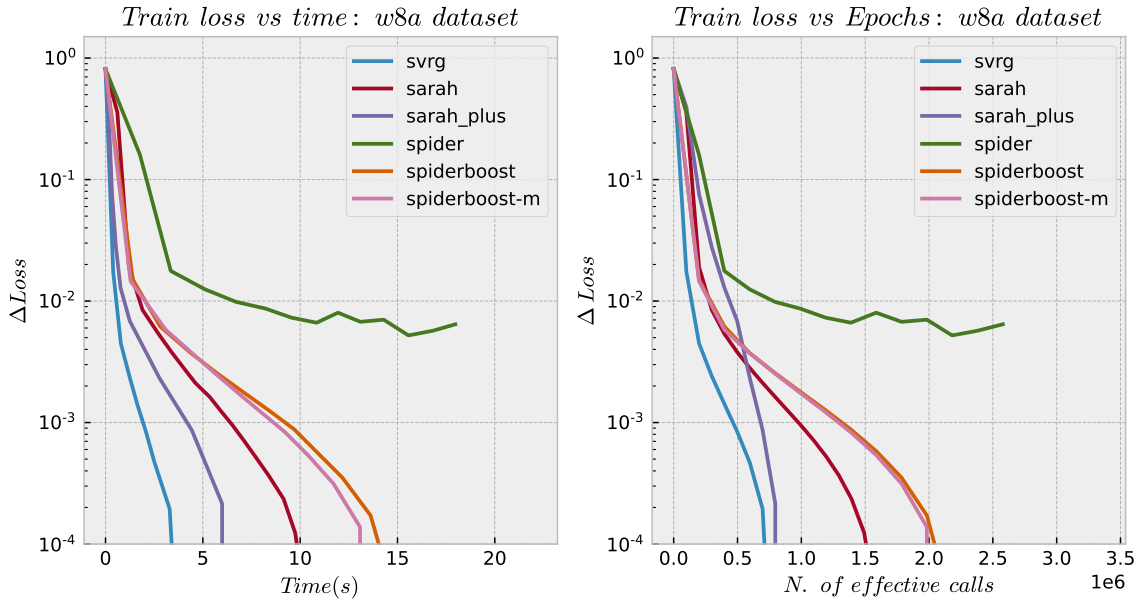


Fig. 16: w8a dataset, robust non regularized loss function (type 3)