Adaptive scaling of the learning rate by second order automatic differentiation

Frédéric de Gournay, Alban Gossard

Adaptive scaling of the learning rate by second order automatic differentiation

Frédéric de Gournay[1,2] Alban Gossard[1,3]
degourna@insa-toulouse.fr alban.paul.gossard@gmail.com

[1] Institut de Mathématiques de Toulouse; UMR5219; Université de Toulouse; CNRS
[2] INSA, F-31077 Toulouse, France [3] UPS, F-31062 Toulouse Cedex 9, France

October 25, 2022

Abstract

In the context of the optimization of Deep Neural Networks, we propose to rescale the learning rate using a new technique of automatic differentiation. This technique relies on the computation of the *curvature*, a second order information whose computational complexity is in between the computation of the gradient and the one of the Hessian-vector product. If $(1C, 1M)$ represents respectively the computational time and memory footprint of the gradient method, the new technique increase the overall cost to either $(1.5C, 2M)$ or $(2C, 1M)$. This rescaling has the appealing characteristic of having a natural interpretation, it allows the practitioner to choose between exploration of the parameters set and convergence of the algorithm. The rescaling is adaptive, it depends on the data and on the direction of descent. The numerical experiments highlight the different exploration/convergence regimes.

1 Introduction

The optimization of Deep Neural Networks (DNNs) has received tremendous attention over the past years. Training DNNs amounts to minimize the expectation of non-convex random functions in a high dimensional space $\mathbb{R}^d$. If $\mathcal{J} : \mathbb{R}^d \to \mathbb{R}$ denotes this expectation, the problem reads

$$\min_{\Theta \in \mathbb{R}^d} \mathcal{J}(\Theta), \tag{1}$$

with Θ the parameters. Optimization algorithms compute iteratively Θ_k, an approximation of a minimizer of (1) at iteration k, by the update rule

$$\Theta_{k+1} = \Theta_k - \tau_k \dot{\Theta}_k, \tag{2}$$

where τ_k is the learning-rate and $\dot{\Theta}_k$ is the update direction. The choice of $\dot{\Theta}_k$ encodes the type of algorithm used. This work focuses on the choice of the learning rate τ_k.

There is a trade-off in the choice of this learning rate. Indeed high values of τ_k allows *exploration* of the parameters space and slowly decaying step size ensures *convergence* in accordance to the famous Robbins-Monro algorithm [36]. This decaying condition may be met by defining the step as $\tau_k = \tau_0 k^{-\alpha}$ with τ_0 being the initial step size and $\frac{1}{2} < \alpha < 1$ a constant. The choice of the initial learning rate and its decay are left to practitioners and these hyperparameters have to be tuned manually in order to obtain the best rate of convergence. For instance, they can be optimized using a grid-search or by using more intricated strategies [40], but in all generality tuning the learning rate and its decay factor is difficult and time consuming. The main issue is that the learning rate has no natural scaling. The goal of this work is to propose an algorithm that, given a direction $\dot{\Theta}_k$ finds automatically a scaling of the learning rate. This rescaling has the following advantages:

- The scaling is adaptive, it depends on the data and of the choice of direction $\dot{\Theta}_k$.
- The scaling expresses the convergence vs. exploration trade-off. Multiplying the rescaled learning rate by 1/2 enforces convergence whereas multiplying it by 1 allows for exploration of the space of parameters.

This rescaling comes at a cost and it has the following disadvantages:

- The computational costs and memory footprint of the algorithm goes from $(1C, 1M)$ to $(1.5C, 2M)$ or $(2C, 1M)$.
- The rescaling method is only available to algorithms that yield directions of descent, it excludes momentum method and notably Adam-flavored algorithm.
- Rescaling is theoritically limited to functions whose second order derivative exists and does not vanish. This non-vanishing condition can be compensated by L^2-regularization.

1.1 Foreword

First recall that second order methods for the minimization of a deterministic $\mathcal{C}^2$ function $\Theta \mapsto \mathcal{J}(\Theta)$, with a Hessian that we denote $\nabla^2\mathcal{J}$, are based on the second order Taylor expansion at iteration k:

$$\mathcal{J}(\Theta_k - \tau_k\dot{\Theta}_k) \simeq \mathcal{J}(\Theta_k) - \tau_k\langle\dot{\Theta}_k, \nabla\mathcal{J}(\Theta_k)\rangle + \frac{\tau_k^2}{2}\langle\nabla^2\mathcal{J}(\Theta_k)\dot{\Theta}_k, \dot{\Theta}_k\rangle. \tag{3}$$

If the Hessian of $\mathcal{J}$ is positive definite, the minimization of the right-hand side leads to the choice

$$\dot{\Theta}_k = P_k^{-1}\nabla\mathcal{J}(\Theta_k) \text{ with } P_k \simeq \nabla^2\mathcal{J}(\Theta_k). \tag{4}$$

Once a direction $\dot{\Theta}_k$ is chosen, another minimization in τ_k gives

$$\tau_k = \frac{\langle\dot{\Theta}_k, \nabla\mathcal{J}(\Theta_k)\rangle}{\|\dot{\Theta}_k\|^2 c(\Theta_k, \dot{\Theta}_k)}, \tag{5}$$

where c is the curvature of the function, and is defined as

$$c(\Theta_k, \dot{\Theta}_k) \overset{\text{def}}{=} \frac{\langle\nabla^2\mathcal{J}(\Theta_k)\dot{\Theta}_k, \dot{\Theta}_k\rangle}{\|\dot{\Theta}_k\|^2}. \tag{6}$$

A second-order driven algorithm can be decomposed in two steps: i) the choice of P_k in (4), and if this choice leads to an update which is a direction of ascent, that is $\langle\dot{\Theta}_k, \nabla\mathcal{J}(\Theta_k)\rangle > 0$, ii) a choice of τ_k by an heuristic inspired from (6) and (5).

In the stochastic setting, we denote as $s \mapsto \mathcal{J}_s$ the mapping of the random function. At iteration k, only information on $(\mathcal{J}_s)_{s\in\mathcal{B}_k}$ can be computed where $(\mathcal{B}_k)_k$ is a sequence of mini-batches which are indepently drawn. If $\mathbb{E}_{s\in\mathcal{B}_k}$ is the empirical average over the mini-batch, we define $\mathcal{J}_{\mathcal{B}_k} = \mathbb{E}_{s\in\mathcal{B}_k}[\mathcal{J}_s]$. Given Θ, the quantity $\mathcal{J}(\Theta)$ is deterministic, and $\mathcal{J}$ is the expectation of $\mathcal{J}_s$ w.r.t. s.

1.2 Related works

Choice of P_k: The choice $P_k = \nabla^2\mathcal{J}_{\mathcal{B}_k}(\Theta_k)$ in (4), leads to a choice $\tau_k = 1$ and to the so-called Newton method. It is possible in theory to compute the Hessian by automatic differentiation if it is sparse [45], but to our knowledge it has not been implemented yet. In [28], the authors solve $\dot{\Theta}_k = \left[\nabla^2\mathcal{J}_{\mathcal{B}_k}(\Theta_k)\right]^{-1}\nabla\mathcal{J}_{\mathcal{B}_k}(\Theta_k)$ by a conjugate gradient method which requires only matrix-vector product which is affordable by automatic differentiation [9, 31]. This point of view, as well as some variants [44, 20], suffer from high computational cost per batch and go through less data in a comparable amount of time, leading to slower convergence at the beginning of the optimization.

Another choice is to set $P_k \simeq \nabla^2\mathcal{J}_{\mathcal{B}_k}(\Theta_k)$ in (4) which is coined as the "Quasi-Newton" approach. These methods directly invert a diagonal, block-diagonal or low rank approximation of the Hessian [5, 38, 37, 30, 29, 49]. In most of these works, the Hessian is approximated by $\mathbb{E}[\nabla\mathcal{J}_s(\theta_k)\nabla\mathcal{J}_s(\theta_k)^T]$, the so-called Fisher-Information matrix, which leads to the natural gradient method [3]. Note also the use of a low-rank approximation of the true Hessian for variance reduction in [14].

Finally, there is an interpretation of adaptive methods as Quasi-Newton methods. Amongst the adaptive method, let us cite RMSProp [43], Adam [19], Adagrad [13] and Adadelta [50]. For all these methods, P_k is as a diagonal preconditioner that reduces the variability of the step size across the different layers. This class of methods can be written

$$\dot{\Theta}_k = P_k^{-1}m_k, \quad m_k \simeq \nabla\mathcal{J}(\Theta_k) \quad \text{and} \quad P_k \simeq \nabla^2\mathcal{J}(\Theta_k). \tag{7}$$

For instance, RMSProp and Adagrad use $m_k = \nabla \mathcal{J}_{\mathcal{B}_k}(\Theta_k)$ whereas Adam maintains in m_k an exponential moving averaging from the past evaluations of the gradient. The RMSProp, Adam and Adagrad optimizers build P_k such that P_k^2 is a diagonal matrix whoses elements are exponential moving average of the square of the past gradients (see [35] for example). It is an estimator of the diagonal part of the Fisher-Information matrix.

All these methods can be incorporated in our framework as we consider the choice of P_k as a preconditioning technique whose step is yet to be found. In a nutshell, if P_k approximates the Hessian up to an unknown multiplicative factor, our method is able to find this multiplicative factor.

Barzilai-Borwein: The Barzilai-Borwein (BB) class of methods [4, 34, 11, 47, 6, 23] may be interpreted as methods which aim at estimating the curvature in (6) by numerical differences using past gradient computations. In the stochastic convex setting, the BB method is introduced in [42] for the choice $\dot{\Theta}_k = \nabla \mathcal{J}(\Theta_k)$ and also for variance-reducing methods [18]. It has been extended in [27] to non-convex problems and in [24] to DNNs. Due to the variance of the gradient and possibly to a poor estimation of the curvature by numerical differences, these methods allow prescribing a new step at each epoch only. In [48, 8], the step is prescribed at each iteration at the cost of computing two mini-batch gradients per iteration. Moreover, in [48] the gradient over all the data needs to be computed at the beginning of each epoch whereas [8] maintains an exponential moving average to avoid this extra computation. The downside of [8] is that they still need to tune the learning rate and its decay factor and that their method has not been tried on other choices than $P_k = \mathrm{Id}$.

Our belief is that approximating by numerical differences in a stochastic setting suffers too much from variance from the data and from the approximation error. Hence we advocate in this study for exact computations of the curvature (6).

Automatic differentiation: The theory that allows to compute the matrix-vector product of the Hessian with a certain direction is well-studied [45, 9, 15, 31] and costs 4 passes (2 forward and backward passes) and 3 memory footprint, when the computation of the gradient costs 2 passes (1 forward and backward pass) and 1 memory footprint. We study the cost of computing the curvature defined in (6), which to the best of our knowledge, has never been studied. Our method has a numerical cost that is always lower than the best BB method [8].

1.3 Our contributions

We propose a change of point of view. While most of the methods presented above use first order information to develop second order algorithms, we use second order information to tune a first order method. The curvature (6) is computed using automatic differentiation in order to estimate the local Lipschitz constant of the gradient and to choose a step accordingly. Our contribution is threefold:

- We propose a method that automatically rescales the learning rate using curvature information in Section 2.1 and we discuss the heuristics of this method in Section 2.2. The rescaling allows the practitioner to choose between three different physical regimes coined as : hyperexploration, exploration/convergence trade-off and hyperconvergence.
- We study the automatic differentiation of the curvature in Section 3 and its computational costs.
- Numerical tests are provided in Section 4 with a discussion on the three different physical regimes introduced in Section 2.1.

2 Rescaling the learning rate

2.1 Presentation and guideline for rescaling

The second order analysis of Section 1 relies on the Taylor expansion

$$\mathcal{J}(\Theta_k - \tau_k \dot{\Theta}_k) \simeq \mathcal{J}(\Theta_k) - \tau_k \langle \dot{\Theta}_k, \nabla \mathcal{J}(\Theta_k) \rangle + \frac{\tau_k^2}{2} c(\Theta_k, \dot{\Theta}_k) \|\dot{\Theta}_k\|^2,$$

with $c(\Theta_k, \dot{\Theta}_k)$ given by (6). This Taylor expansion yields the following algorithm: given Θ_k and an update direction $\dot{\Theta}_k$, compute $c(\Theta_k, \dot{\Theta}_k)$ by (6), the step τ_k by (5) and finally update the parameters

Θ_k by (2). The first order analysis is slightly different. Starting with the second order exact Taylor expansion in integral form:

$$\mathcal{J}(\Theta_k - \tau_k\dot{\Theta}_k) = \mathcal{J}(\Theta_k) + \tau_k\langle\dot{\Theta}_k, \nabla\mathcal{J}(\Theta_k)\rangle + \int_{t=0}^{\tau_k}(\tau_k - t)c(\Theta_k - t\dot{\Theta}_k, \dot{\Theta}_k)\|\dot{\Theta}_k\|^2 dt,$$

we introduce the local directional Lipschitz constant of the gradient

$$L_k = \max_{t\in[0,\tau_k]} |c(\Theta_k - t\dot{\Theta}_k, \dot{\Theta}_k)|, \tag{8}$$

in order to bound the right-hand side of the Taylor expansion. One obtains

$$\mathcal{J}(\Theta_k - \tau_k\dot{\Theta}_k) \leq \mathcal{J}(\Theta_k) - \tau_k\langle\dot{\Theta}_k, \nabla\mathcal{J}(\Theta_k)\rangle + \frac{\tau_k^2}{2}L_k\|\dot{\Theta}_k\|^2. \tag{9}$$

Introducing the rescaling r_k

$$r_k = \frac{\langle\dot{\Theta}_k, \nabla\mathcal{J}(\Theta_k)\rangle}{\|\dot{\Theta}_k\|^2 L_k} \tag{10}$$

and writing $\tau_k = r_k\ell$, Equation (9) turns into

$$\mathcal{J}(\Theta_k - \ell r_k\dot{\Theta}_k) \leq \mathcal{J}(\Theta_k) + \left(\ell^2 - \ell\right)\frac{L_k}{2}\|r_k\dot{\Theta}_k\|^2 \quad \forall\ell. \tag{11}$$

Any choice of ℓ in $]0,1[$ leads to a decrease of $\mathcal{J}$ in (11). The choice $\ell = \frac{1}{2}$ allows faster decrease of the right-hand side of (11). We coin the choice $\ell = 1$ in (11) as the *exploration choice* and the choice $\ell = \frac{1}{2}$ as the *convergence choice*. The only difficulty in computing (10) lies in the computation of L_k. Indeed, L_k is a maximum over an unknown interval and, in the stochastic setting, we only estimate the function $\mathcal{J}$ and its derivative on a batch $\mathcal{B}_k$.

We propose to build $\tilde{L}_k$ an estimator of L_k by the following rules.

- Replace the maximum over the unknown interval $[0,\tau_k]$ in (8) by the value at $t = 0$. This is reminiscent of the Newton's method.
- Perform an exponential moving average on the past computations of r_k in order to average over the data previously seen.
- Use the maximum of this latter exponential moving average and the current estimate in order to stablize $\tilde{L}_k$.

The algorithm reads as follows:

Algorithm 1 Rescaling of the learning rate

```
1: Hyperparameters $\beta_3 = 0.9$ (exponential moving average).
2: Initialization $\hat{c}_0 = 0$
3: Input (at each iteration $k$): a batch $\mathcal{B}_k$, $g_k = \mathbb{E}_{s\in\mathcal{B}_k}[\nabla\mathcal{J}_s(\Theta_k)]$ and $\dot{\Theta}_k$ a direction that
   verifies $\langle g_k, \dot{\Theta}_k\rangle > 0$.
4: $c_k = \mathbb{E}_{s\in\mathcal{B}_k}\left[\left|\langle\nabla^2\mathcal{J}_s(\Theta_k)\dot{\Theta}_k, \dot{\Theta}_k\rangle\right|\right]/\|\dot{\Theta}_k\|^2$        ▷ local curvature
5: $\hat{c}_k = \beta_3\hat{c}_{k-1} + (1-\beta_3)c_k$ and $\tilde{c}_k = \hat{c}_k/(1-\beta_3^k)$        ▷ moving average
6: $\tilde{L}_k = \max(\tilde{c}_k, c_k)$        ▷ stabilization
7: $r_k = \langle\dot{\Theta}_k, g_k\rangle / \left(2\|\dot{\Theta}_k\|^2\tilde{L}_k\right)$        ▷ rescaling factor
8: Output (at each iteration $k$): $r_k$ a rescaling of the direction $\dot{\Theta}_k$.
9: Usage of rescaling: The practitioner should use the update rule $\Theta_{k+1} = \Theta_k - \ell r_k\dot{\Theta}_k$, where $\ell$
   follows the Rescaling guidelines (see below).
```

Note that the curvature c_k is computed with the same batch that the one used to compute g_k and $\dot{\Theta}_k$.

Rescaling guidelines Given a descent direction $\dot{\Theta}_k$, the update rule is given by

$$\Theta_{k+1} = \Theta_k - \ell r_k \dot{\Theta}_k,$$

where ℓ is the learning-rate that has to be chosen by the practitioner and r_k is the rescaling computed by Algorithm 1. In the deterministic case, ℓ has a physical interpretation:

- $1 \geq \ell \geq \frac{1}{2}$ (*Convergence/exploration trade-off*). The choice $\ell = 1$ (exploration) is the largest step that keeps the loss function non increasing. The choice $\ell = 1/2$ (convergence) ensures the fastest convergence to the closest local minimum. It is advised to start from $\ell = 1$ and decrease to $\ell = 1/2$ (see Section 4.1).
- $\ell > 1$ (*Hyperexploration*). The expected behavior is a loss function increase and large variations of the parameters. This mode can be used to escape local basin of attraction in annealing methods (see Section 4.2).
- $0 < \ell < 1/2$ (*Hyperconvergence*). This mode slows down the convergence. In the stochastic setting, if the practitioner has to resort to setting $\ell < 1/2$ in order to obtain convergence, then some stochastic effects are of importance in the optimization procedure (see Section 4.3).

2.2 Analysis of the rescaling

Several remarks are necessary to understand the limitations and applications of rescaling.

The algorithm does not converge in the deterministic setting. Note that in the deterministic one dimensional case, when $\mathcal{J}$ is convex (i.e. the curvature is positive), $\beta_3 = 0$ and $\ell = 1/2$, the algorithm boils down to the Newton method. It is known that the Newton method may fail to converge, even for strictly convex smooth functions. For example if we choose

$$\mathcal{J}(\Theta) = \sqrt{1 + \Theta^2},$$

the iterates of the Newton method are given by $\Theta_{k+1} = -\Theta_k^3$, which diverges as soon as $|\Theta_0| > 1$. This problem comes from the fact that the curvature $c(\Theta_k - t\dot{\Theta}_k, \dot{\Theta}_k)$ has to be computed for each $t \in [0, \tau_k]$ in order to estimate L_k in (8) but this maximum is estimated by its value at $t = 0$. In this example, $c(\Theta_k, \dot{\Theta}_k)$ is a bad estimator for L_k as it is too small and the resulting step is too large.

Another issue in DNN is the massive use of piecewise linear activation functions which can make the Hessian vanish and in this case, the rescaled algorithm may diverge. For example if the loss function is locally linear, then $c_k = 0$ in line 4 and if we choose $\beta_3 = 0$ then $r_k = +\infty$ in line 7.

The algorithm does not converge in the stochastic setting. Let X be a vector-valued random variable and $\mathcal{J}$ is the function

$$\mathcal{J}(\Theta) = \frac{1}{2}\mathbb{E}\left[\|\Theta - X\|^2\right],$$

then for any value of β_3 and for the choice $\ell = \frac{1}{2}$, the rescaled algorithm yields the update $\Theta_k = \mathbb{E}_{\mathcal{B}_k}[X]$, when the optimal value is $\Theta^\star = \mathbb{E}[X]$. The algorithm oscillates around $\Theta^\star$, with oscillations depending on the variance of the gradient. This oscillating stochastic effect is well known and is the basic analysis of SGD. Since the proposed rescaling analysis is performed in a deterministic setting, it is not designed to offer any solution to this problem.

Enforcing convergence by Robbins-Monro conditions In order to enforce convergence, we can use the results of [36]. It is then sufficient to sow instructions like

$$\alpha \leq \ell r_k k^\delta \leq \beta, \tag{12}$$

with fixed $\alpha, \beta > 0$ and $\delta \in]1/2, 1[$. This is the choice followed by [8] for instance. Note that convergence analysis for curvature-dependent step is, to our knowledge, studied only in [2], for the non-stochastic time-continuous setting.

Fostering convergence with L_2 regularization In the deterministic case, the non-convergence of the algorithm is caused by vanishing eigenvalues of the Hessian. This issue can be fixed by adding a term $\frac{\lambda}{2}\|\Theta\|^2$ to the function $\Theta \mapsto \mathcal{J}(\Theta)$, with $\lambda > 0$. This method is coined as L_2 regularization with parameter λ. This method shifts the eigenvalues of the Hessian of $\mathcal{J}$ by the parameter λ. Close to the minimum, every eigenvalue of the Hessian is then positive. Although L_2 regularization does not guarantee convergence, it promotes it.

Gradient preconditioning In case of gradient preconditioning $\dot{\Theta}_k = P_k^{-1} g_k$ with $P_k \simeq \nabla^2 \mathcal{J}(\Theta_k)$, the advantage of rescaling is that the practitioner is allowed to approximate the Hessian up to a multiplicative factor. Indeed suppose that instead of providing a good estimate of the Hessian, the practitioner multiplies it at each iteration by an arbitrary factor $\alpha_k \in \mathbb{R}$. In this case, $\dot{\Theta}_k$ is multiplied by α_k^{-1} but the curvature c_k does not change. This means that $\hat{c}_k$ is independent of the previous $(\alpha_s)_{s\leq k}$. Finally, the rescaling r_k is multiplied by α_k. Hence the output of the algorithm $r_k \dot{\Theta}_k$ is independent of the sequence $(\alpha_s)_{s\leq k}$. Therefore, the practitioner does not need to worry about finding the right multiplicative factor, it is accounted for by the rescaling method.

Negativeness of the curvature (line 4) The main difference beween a first-order analysis (8) and a second-order (5) lies in handling the case when the curvature is negative. The first-order analysis, which we choose, relies on using absolute value of the curvature, when second-order analysis relies on more intricated methods, see [1, 7, 25, 10]. Note that the absolute value is taken inside the batch average in line 4 and not outside. Otherwise data in the batch where $\langle \nabla \mathcal{J}^2(\Theta_k)\dot{\Theta}_k, \dot{\Theta}_k\rangle$ is negative could compensate the data where it is positive, leading to a bad estimation of the curvature.

Heuristics in the estimation of L_k (lines 5 and 6) The estimator of L_k must comply with two antagonist requirements. The first one is to average the curvature over the different batches to effectively compute the true curvature of $\mathcal{J}$. The second one is to use the local curvature at point Θ_k and in the direction $\dot{\Theta}_k$ which requires to forget old iterations. This advocates the use of an exponential moving average in line 5 with the parameter β_3. The maximum in line 6 is reminiscent of the construction of AMSGrad [35] from Adam [19], and it stabilizes batches where $c_k \gg \tilde{c}_k$. In order to be consistent with the remark in *Gradient preconditioning*, the averaged quantity is the one which does not depend on the unknown multiplicative factor α_k.

3 Computing the curvature

In this section, we focus on the computation of $c(\Theta, \dot{\Theta})$ by automatic differentiation and its cost.

3.1 Main results

A Neural Network $\mathcal{N}$ is a directed acyclic graph and at each node of the graph, the data are transformed and fed to the rest of the graph. The data at the output x_n are then compared to y. Since there is no cycle in the graph, there is no mathematical restriction to turn such graph into a list. The set of parameters for layer s is denoted as θ_s, and we denote $\Theta = (\theta_s)_{s=0..n}$ the set of parameters of $\mathcal{N}$. The action of $\mathcal{N}$ is expressed by the recurrence:

$$x_{s+1}(\Theta) = \mathcal{F}_s(x_s(\Theta), \theta_s), \quad 0 \leq s \leq n-1 \tag{13}$$

where $\mathcal{F}_s$ is the action of the s^{th} layer of $\mathcal{N}$. The output x_n is then compared to a target via a loss function $\mathcal{F}_n$ and we denote $x_{n+1} \in \mathbb{R}$ the result of this loss function.

Let $X(\Theta) = (x_s(\Theta))_{s=0..n+1}$ denote the set of data as it is transformed through the neural network. The intermediate data $x_s(\Theta)$ (resp. parameter θ_s) are supposed to belong to an Hilbert space $\mathcal{H}_s$ (resp. $\mathcal{G}_s$). We then have for each $0 \leq s \leq n$

$$\mathcal{F}_s : \mathcal{H}_s \times \mathcal{G}_s \to \mathcal{H}_{s+1} \text{ and } \mathcal{H}_{n+1} = \mathbb{R}.$$

The gradient of $\mathcal{J}$ with respect to Θ is computed using automatic differentiation. This requires to define the differentials of $\mathcal{F}_s$ with respect to its variables. Let $\partial_x \mathcal{F}_s : \mathcal{H}_s \to \mathcal{H}_{s+1}$, resp. $\partial_\theta \mathcal{F}_s : \mathcal{G}_s \to \mathcal{H}_{s+1}$, be the differential of $\mathcal{F}$ at the point $(x_s(\Theta), \theta_s)$ w.r.t. x, resp. θ. Denote $(\partial_x \mathcal{F}_s)^* : \mathcal{H}_{s+1} \to \mathcal{H}_s$

and $(\partial_\theta \mathcal{F}_s)^* : \mathcal{H}_{s+1} \to \mathcal{G}_s$ the adjoints of the differentials of $\mathcal{F}_s$. These adjoints are defined for all $\phi \in \mathcal{H}_{s+1}$ as the unique linear mapping that verifies:

$$\begin{aligned} \langle \partial_x \mathcal{F}_s^* \phi, \psi \rangle_{\mathcal{H}_s} &= \langle \phi, \partial_x \mathcal{F}_s \psi \rangle_{\mathcal{H}_{s+1}} \quad \forall \psi \in \mathcal{H}_s \\ \langle \partial_\theta \mathcal{F}_s^* \phi, \psi \rangle_{\mathcal{G}_s} &= \langle \phi, \partial_\theta \mathcal{F}_s \psi \rangle_{\mathcal{H}_{s+1}} \quad \forall \psi \in \mathcal{G}_s. \end{aligned}$$

Denote by $\nabla^2 \mathcal{F}_s$ the second order derivative tensor of $\mathcal{F}_s$ at the point (x_s, θ_s). The backward of the data $\hat{X} = (\hat{x}_s)_{s=1..n+1}$ and the backward-gradient $\hat{\Theta} = (\hat{\Theta}_s)_{s=0..n}$ are defined by:

$$\begin{cases} \hat{x}_s = (\partial_x \mathcal{F}_s)^* \hat{x}_{s+1} & \text{with } \hat{x}_{n+1} = 1 \\ \hat{\theta}_s = (\partial_\theta \mathcal{F}_s)^* \hat{x}_{s+1}. \end{cases} \tag{14}$$

In Algorithm 2, the standard backpropagation algorithm is given as well as the modifications needed to compute the curvature. The proof of this algorithm is given in Section 3.2.

Algorithm 2 Backpropagation with curvature computation

```
1: Compute and store the data $X = (x_s)_s$ with a forward pass (13).
2: Compute and store the backward $\hat{X} = (\hat{x}_s)_s$ and $\hat{\Theta} = (\hat{\theta}_s)_s$ using (14).
3: Then $\nabla \mathcal{J}(\Theta) = \hat{\Theta}$.
4: Choose any direction of update $\dot{\Theta} = (\dot{\theta}_s)_s$.
5: Compute the tangent $\dot{X} = (\dot{x}_s)_s$ with the following forward pass:
       $\dot{x}_{s+1} = (\partial_x \mathcal{F}_s)\dot{x}_s + (\partial_\theta \mathcal{F}_s)\dot{\theta}_s, \quad \dot{x}_0 = 0$        (15)
6: Then $\langle \nabla^2 \mathcal{J}(\Theta)\dot{\Theta}, \dot{\Theta} \rangle = \sum_s \langle \hat{x}_{s+1}, \nabla^2 \mathcal{F}_s(\dot{x}_s, \dot{\theta}_s) \otimes (\dot{x}_s, \dot{\theta}_s) \rangle_{\mathcal{H}_{s+1}}$.
```

By Algorithm 2, the computation of the curvature $c(\theta, \dot{\theta})$ requires 3 passes in total and the storage of X and $\hat{X}$ whereas the computation of the gradient requires 2 passes and the storage of X. Hence the memory footprint is multiplied by 2 and the computation time by 1.5. We show in Section 3.3 how to design a divide-and-conquer algorithm that changes this cost to $(2C, 1M)$.

Theorem 1 *If $(1C, 1M)$ represents respectively the computational time and memory footprint of the standard backpropagation method, Algorithm 2 costs either $(1.5C, 2M)$ or $(2C, 1M)$.*

This result is of importance since it states that computing the exact curvature is at least as cheap as using numerical differences of the gradient [8].

3.2 Proof of Algorithm 2

The goal of this section is to analyse the complexity of computing the curvature term and to prove Algorithm 2.

Forward pass We recall that the forward pass is computed through the recurrence

$$x_{s+1}(\Theta) = \mathcal{F}_s(x_s(\Theta), \theta_s), \quad 0 \le s \le n.$$

Moreover the objective function is defined as $\mathcal{J}(\Theta) = x_{n+1}(\Theta)$. The computation of X through the recurrence (13) is denoted as the *forward pass*.

Tangent pass Given Θ, a set of data $X(\Theta)$, and an arbitrary direction $\dot{\Theta} = (\dot{\theta}_s)_{s=0..n}$, the tangent $\dot{X} = (\dot{x}_s)_{s=0..n}$ is defined as

$$\dot{x}_s = \lim_{\tau \to 0} \frac{x_s(\Theta + \tau\dot{\Theta}) - x_s(\Theta)}{\tau}.$$

For each layer s, recall that $\partial_x \mathcal{F}_s$ (resp. $\partial_\theta \mathcal{F}_s$) is the differential of $\mathcal{F}_s$ with respect to the parameter x (resp. θ) at the point $(x_s(\Theta), \theta_s)$. From now, we omit the notation of the point at which the differential is taken in order to simplify the notations. By the chain rule theorem, we have that if $\dot{x}_s$ exists, then the forward recurrence (13) yields

$$\dot{x}_{s+1} = (\partial_x \mathcal{F}_s)\dot{x}_s + (\partial_\theta \mathcal{F}_s)\dot{\theta}_s, \quad \dot{x}_0 = 0. \tag{16}$$

A recurrence on s allows obtaining existence of $\dot{X}$ and the scaling

$$X(\Theta + \tau\dot{\Theta}) = X(\Theta) + \tau\dot{X} + O(\tau^2).$$

Hence, if $X(\Theta)$ is computed and $\dot{\Theta}$ is chosen, then $\dot{X}$ – the tangent in direction $\dot{\Theta}$ – can be computed via the forward recurrence (16) and we have

$$\langle \nabla\mathcal{J}(\Theta), \dot{\Theta}\rangle = \dot{x}_{n+1}. \tag{17}$$

The recurrence (16) which allows the computation of $\dot{X}$ is coined as the *tangent pass.*

Adjoint/backward pass In order to compute the gradient, one resorts to the backpropagation algorithm which allows reversing the recurrence (16) that defines the tangent and computing directly $\hat{\Theta} = (\hat{\theta}_s)_{s=0..n}$ such that

$$\langle \nabla\mathcal{J}(\Theta), \dot{\Theta}\rangle = \dot{x}_{n+1} = \sum_s \langle \dot{\theta}_s, \hat{\theta}_s\rangle_{\mathcal{G}_s}.$$

The vector $\hat{\Theta}$ is then equal to $\nabla\mathcal{J}(\Theta)$, provided that one uses the scalar product induced by the sum of the scalar products of all $\mathcal{G}_s$. To compute $\hat{\Theta}$, we use $\partial_x\mathcal{F}_s^* : \mathcal{H}_{s+1} \to \mathcal{H}_s$ and $\partial_\theta\mathcal{F}_s^* : \mathcal{H}_{s+1} \to \mathcal{G}_s$ the adjoints of the differentials of $\mathcal{F}_s$. The backward of the data $\hat{X} = (\hat{x}_s)_{s=1..n+1}$ and the backward-gradient $\hat{\Theta} = (\hat{\Theta}_s)_{s=0..n}$ are defined by the reversed recurrence:

$$\begin{cases} \hat{x}_s = (\partial_x\mathcal{F}_s)^*\hat{x}_{s+1} & \text{with } \hat{x}_{n+1} = 1 \\ \hat{\theta}_s = (\partial_\theta\mathcal{F}_s)^*\hat{x}_{s+1}. \end{cases} \tag{18}$$

The definition of the adjoint and the formula of the tangent (16) give the following equality:

$$\begin{aligned} \langle \dot{x}_{s+1}, \hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}} &= \langle (\partial_x\mathcal{F}_s)\dot{x}_s + (\partial_\theta\mathcal{F}_s)\dot{\theta}_s, \hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}} \\ &= \langle \dot{x}_s, (\partial_x\mathcal{F}_s)^*\hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}} + \langle \dot{\theta}_s, (\partial_\theta\mathcal{F}_s)^*\hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}} \\ &= \langle \dot{x}_s, \hat{x}_s\rangle_{\mathcal{H}_s} + \langle \dot{\theta}_s, \hat{\theta}_s\rangle_{\mathcal{G}_s} \end{aligned} \tag{19}$$

Summing up the above equations for every s, we obtain:

$$\dot{x}_{n+1} = \langle \dot{x}_{n+1}, \hat{x}_{n+1}\rangle_{\mathcal{H}_{n+1}} = \langle \dot{x}_0, \hat{x}_0\rangle_{\mathcal{H}_{n+1}} + \sum_s \langle \dot{\theta}_s, \hat{\theta}_s\rangle_{\mathcal{G}_s} = \sum_s \langle \dot{\theta}_s, \hat{\theta}_s\rangle_{\mathcal{G}_s},$$

where we use $\dot{x}_0 = 0$ and $\hat{x}_{n+1} = 1$. We then obtain the celebrated backward propagation formula

$$\nabla\mathcal{J}(\Theta) = \hat{\Theta}.$$

The complexity analysis of the computation of the gradient by the backward formula shows that it requires the computation and the storage of the forward pass in order to be able to evaluate $(\partial_x\mathcal{F}_s)^*$ and $(\partial_\theta\mathcal{F}_s)^*$ at the point $(x_s(\Theta), \theta_s)$.

Computing the curvature Equation 17 is the implicit definition of $\nabla\mathcal{J}$, where $\dot{X}$ is defined by the recurrence (16). The trick of automatic differentiation is to use the the backward $\hat{X}$ defined in recurrence (18) to reverse (16). This inversion is performed in (19) and it allows not computing the tangent $\dot{X}$. We show in this paragraph that the backward $\hat{X}$ also reverses the recurrence defining the second order term $\ddot{X}$ defined in (20) below. Once the direction $\dot{\Theta}$ is chosen, the curvature term can be computed by only a forward pass. To this end, for any direction $\dot{\Theta}$, introduce $\ddot{X} = (\ddot{x}_s)_{s=0..n+1}$ as:

$$\ddot{x}_s = \lim_{\tau\to 0} \frac{x_s(\Theta + \tau\dot{\Theta}) - x_s(\Theta) - \tau\dot{x}_s}{\tau^2}, \tag{20}$$

where $\dot{x}_s$ is the tangent defined in (16). Recall that $\nabla^2\mathcal{F}_s : \mathcal{H}_s \times \mathcal{G}_s \to \mathcal{H}_{s+1}$ is the bilinear symmetric mapping that represents the second order differentiation of $\mathcal{F}_s$ at point $(x_s(\Theta), \theta_s)$. It is defined as the only bilinear symmetric mapping that verifies for every (h_x, h_θ) the relation

$$\begin{aligned}\mathcal{F}_s(x_s(\Theta) + h_x, \theta_s + h_\theta) =& \mathcal{F}_s(x_s(\Theta), \theta_s) + \partial_x\mathcal{F}_s h_x + \partial_\theta\mathcal{F}_s h_\theta + \frac{1}{2}\nabla^2\mathcal{F}_s(h_x, h_\theta) \otimes (h_x, h_\theta) \\ &+ o(\|h_x\|^2 + \|h_\theta\|^2)\end{aligned}$$

It is easy to prove that $\ddot{x}_s$ exists and verifies:

$$\ddot{x}_{s+1} = (\partial_x\mathcal{F}_s)\ddot{x}_s + \frac{1}{2}\nabla^2\mathcal{F}_s(\dot{x}_s, \dot{\theta}_s) \otimes (\dot{x}_s, \dot{\theta}_s), \qquad \text{with } \ddot{x}_0 = 0. \tag{21}$$

Indeed, denote $\xi_s = x_s(\Theta + \tau\dot{\Theta}) - x_s(\Theta) - \tau\dot{x}_s$ so that

$$\ddot{x}_s = \lim_{\tau\to 0}\frac{\xi_s}{\tau^2},$$

we have

$$\begin{aligned}\xi_{s+1} &= \mathcal{F}_s(x_s(\Theta + \tau\dot{\Theta}), \theta_s + \tau\dot{\theta}_s) - \mathcal{F}_s(x_s(\Theta), \theta_s) - \tau(\partial_x\mathcal{F}_s)\dot{x}_s - \tau(\partial_\theta\mathcal{F}_s)\dot{\theta}_s \\ &= \mathcal{F}_s(\xi_s + x_s(\Theta) + \tau\dot{x}_s, \theta_s + \tau\dot{\theta}_s) - \mathcal{F}_s(x_s(\Theta), \theta_s) - \tau(\partial_x\mathcal{F}_s)\dot{x}_s - \tau(\partial_\theta\mathcal{F}_s)\dot{\theta}_s \\ &= (\partial_x\mathcal{F}_s)\xi_s + \frac{\tau^2}{2}\nabla^2\mathcal{F}_s\left(\frac{\xi_s}{\tau} + \dot{x}_s, \dot{\theta}_s\right) \otimes \left(\frac{\xi_s}{\tau} + \dot{x}_s, \dot{\theta}_s\right) + o(\tau^2 + \|\xi_s\|^2).\end{aligned} \tag{22}$$

By a forward recurrence on (22), starting with $\xi_0 = 0$, we have that $\xi_s = O(\tau^2)$ so that $\ddot{x}_s$ exists. Dividing (22) by τ^2 and taking the limit yields (21).

Upon replacing $\dot{X}$ by $\ddot{X}$, the trick used in (19) can be applied and translates into:

$$\begin{aligned}\langle \ddot{x}_{s+1}, \hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}} &= \langle (\partial_x\mathcal{F}_s)\ddot{x}_s + \frac{1}{2}\nabla^2\mathcal{F}_s(\dot{x}_s, \dot{\theta}_s) \otimes (\dot{x}_s, \dot{\theta}_s), \hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}} \\ &= \langle \ddot{x}_s, (\partial_x\mathcal{F}_s)^*\hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}} + \frac{1}{2}\langle \nabla^2\mathcal{F}_s(\dot{x}_s, \dot{\theta}_s) \otimes (\dot{x}_s, \dot{\theta}_s), \hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}} \\ &= \langle \ddot{x}_s, \hat{x}_s\rangle_{\mathcal{H}_s} + \frac{1}{2}\langle \nabla^2\mathcal{F}_s(\dot{x}_s, \dot{\theta}_s) \otimes (\dot{x}_s, \dot{\theta}_s), \hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}}\end{aligned}$$

Summing up these equations in s and using $\ddot{x}_0 = 0$ and $\hat{x}_{n+1} = 1$, we obtain

$$\begin{aligned}\ddot{x}_{n+1} &= \langle \ddot{x}_{n+1}, \hat{x}_{n+1}\rangle_{\mathcal{H}_{n+1}} \\ &= \langle \ddot{x}_0, \hat{x}_0\rangle_{\mathcal{H}_{n+1}} + \sum_s \frac{1}{2}\langle \nabla^2\mathcal{F}_s(\dot{x}_s, \dot{\theta}_s) \otimes (\dot{x}_s, \dot{\theta}_s), \hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}} \\ &= \sum_s \frac{1}{2}\langle \nabla^2\mathcal{F}_s(\dot{x}_s, \dot{\theta}_s) \otimes (\dot{x}_s, \dot{\theta}_s), \hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}}.\end{aligned}$$

In order to conclude and prove Algorithm 2, it is sufficient to remark that $\mathcal{J}(\Theta) = x_{n+1}(\Theta)$ so that $\frac{1}{2}\langle \nabla^2\mathcal{J}(\Theta)\dot{\Theta}, \dot{\Theta}\rangle = \ddot{x}_{n+1}$.

More on automatic differentiation In Section A.1, the reader will find a method to compute the matrix-vector product with the Hessian. This method is not new and is known as the *Pearlmutter's trick* [9, 31]. We prove this trick in our setting in order to link our computations with other automatic-differentiation techniques. Moreover, we also give some of the expression of $\nabla^2\mathcal{F}_s(\dot{x}_s, \dot{\theta}_s) \otimes (\dot{x}_s, \dot{\theta}_s)$ for standard layers in Appendix A.2 to settle the notations.

3.3 Proof of Theorem 1

Recall that $(1C, 1M)$ is the complexity of a gradient computation, we show how to change the overall cost of computing the curvature from $(1.5C, 2M)$ to $(2C, 1M)$ by a divide-and-conquer algorithm. In order to simplify the analysis, several simplifications are made.

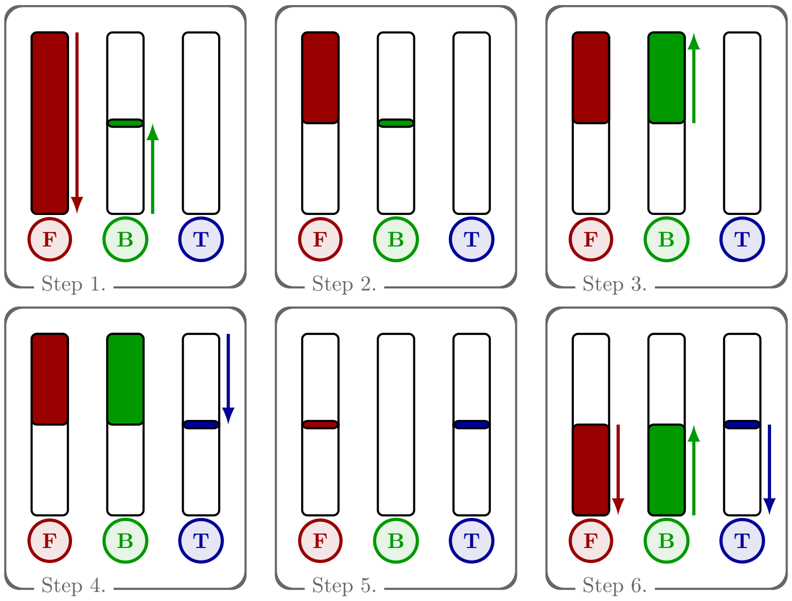


Figure 1: Illustration of the divide-and-conquer algorithm that changes the cost of computating the curvature from $(1.5C, 2M)$ to $(2C, 1M)$. The rectangles above the letters **F, B, T** represent the three different passes (in order: forward, backward and tangent). The memory usage is represented by color-filling in the rectangles, the computations are represented by arrows on the right of the passes. In total, the filled area never exceeds 1 rectangle, hence memory usage is $1M$. The total length of the arrows is 4 times the length of a rectangle, this represents 4 passes. The computational time is then twice the computational time of the standard backward algorithm.

- There are three kind of passes, the forward pass in (13) that computes X, the backward pass in (18) that computes $\hat{X}$ and $\hat{\Theta}$ and the tangent-curvature pass described in Algorithm 2 that computes the curvature. We suppose that each of these passes have roughly the same computational cost $C/2$. This assumption is subject to discussion. In one hand the backward and tangent passes require each twice as much matrix multiplication as the forward pass. On the other hand, soft activation functions are harder to compute in the forward pass.
- We assume that storing X or $\hat{X}$ has the same memory footprint $1M$. Notably, we suppose that the cost of storing the parameters Θ or the gradient $\hat{\Theta}$ is negligeable with respect to the storage of the data through the network. This assumption can only be made for optimization with large enough batches $\mathcal{B}_k$.
- We suppose that we can divide the neural network in two pieces that each costs half the memory and half the computational time. This means that we are able to find L, such that the storage of $(x_s)_{s\leq L}$ and the storage of $(x_s)_{s\geq L}$ have same memory footprint $M/2$. Moreover we suppose that performing a pass for $s \geq L$ or for $s \leq L$ costs $C/4$ computational time. This assumption is reasonable and simplifies the analysis but it is of course possible to exhibit pathological networks that won't comply with this assumption.
- We suppose that the only cost in data transfer comes from the initialization of the parameters Θ, the initial data x_0 and the direction of descent $\dot{\Theta}$. Note that the computation of $\dot{\Theta}$ requires the computation of the gradient $\hat{\Theta}$.

We now describe how to compute the curvature with $(2C, 1M)$ and no extra data transfer. We display the current memory load and the elapsed computational time at the end of each phase. A visual illustration of this algorithm is proposed in Figure 1.

0. Transfer the data x_0 and Θ.
1. Compute $X = (x_s)_s$ and store it. For $s \geq L$, compute the backward via (14) without storing it. Cost is $(\frac{3}{4}C, 1M)$

2. Flush from memory $(x_s)_{s\geq L}$. Cost is $(\frac{3}{4}C, \frac{1}{2}M)$
3. For $s \leq L$, compute the backward via (14) and store it. Cost is $(1C, 1M)$
4. Choose the descent direction and transfer the data $\dot{\Theta}$. Compute the tangent via (16) for $s \geq L$. Cost is $(\frac{5}{4}C, M)$
5. Flush from memory $(\hat{x}_s)_s$ and $(x_s)_{s<L}$. Cost is $(\frac{5}{4}C, 0M)$
6. For $s \geq L$, compute the forward, the backward and store them. Compute the tangent for $s \geq L$. Cost is $(2C, 1M)$

4 Numerical experiments

4.1 Convergence/exploration trade off

4.1.1 The RED algorithm

In order to test the convergence/exploration trade-off, we reproduce the benchmark of [8]. We set ourselves in the case where the initial parameters are randomly chosen, so that the practitioner wants a smooth transition from exploration ($\ell = 1$) to convergence ($\ell = \frac{1}{2}$). We choose in Algorithm 3 a simple, per epoch, exponential decay rule of the learning rate ℓ from 1 to 1/2. This algorithm is coined as RED (Rescaled with Exponential Decay). We purposely unplug any other tricks of the trade, notably Robbins-Monro convergence conditions. Indeed, a Robbins-Monro decay rule would interfere with our analysis. Algorithm RED is not a production algorithm, it serves at testing the "natural" convergence properties of rescaling. In Appendix C.4, we provide a comparison of RED with a standard SGD that has Robbins-Monro decaying conditions. Due to the remark in Section 2.2, we make clear that L^2-regularization is used. If $\Theta \mapsto \mathcal{L}_s(\Theta)$ is the original loss function, then the function $\mathcal{J}_s$ is defined as $\mathcal{J}_s(\Theta) \overset{\text{def}}{=} \mathcal{L}_s(\Theta) + \frac{\lambda}{2}\|\Theta\|^2$.

Algorithm 3 RED (rescaled-exponential-decay) for SGD or RMSProp preconditioning **and no convergence guaranty**

```
1: **Input parameters** $\beta_2 = 0.999$ (RMSProp parameter), RMSProp (boolean), $\lambda > 0$ ($L^2$-regularization), $N$ (total number of epochs), $\varepsilon = 10^{-8}$ (numerical stabilization).
2: **Initialization** $\hat{v}_0 = 0$, $\Theta_0$ random, $\ell = 1$ initial learning rate and $\eta = 1/2$ the step multiplicative factor between the first and the last iterations.
3: **for** $k = 1, 2, ..$ **do**
4:     $g_k = \mathbb{E}_{s\in\mathcal{B}_k}[\nabla \mathcal{J}_s(\Theta_k)]$        ▷ gradient
5:     **if** RMSProp **then**        ▷ RMSProp preconditioning
6:         $\hat{v}_k = \beta_2\hat{v}_{k-1} + (1-\beta_2)g_k^2$ and $\tilde{v}_k = \hat{v}_k/(1-\beta_2^k)$ and $P_k = \mathrm{diag}(\sqrt{\tilde{v}_k} + \varepsilon)$
7:     **else**
8:         $P_k = \mathrm{Id}$
9:     **end if**
10:    $\dot{\Theta}_k = P_k^{-1}g_k$        ▷ direction of update
11:    **Use Algorithm 1 and compute** $r_k$        ▷ rescaling
12:    $\Theta_{k+1} = \Theta_k - \ell r_k \dot{\Theta}_k$        ▷ parameters update
13:    At the end of each epoch $\ell \leftarrow \eta^{\frac{1}{N}}\ell$
14: **end for**
```

The numerical experiments are done on the benchmark of [8]. It consists in four test cases, a MNIST classifier [22], a CIFAR-10 classifier [21] with VGG11 [39] architecture, a CIFAR-100 classifier with VGG19 and the classical autoencoder of MNIST described in [16]. The ReLU units are replaced by smooth versions in order to compute the curvature term, and L^2 regularization is added to each test. The models are trained with a batch size of 256 and the number of epochs is set to 200 for MNIST classification and 500 for the others. The precise set of parameters that allows reproductibility is described in Appendix B. We also give indications of the computational time on an NVIDIA Quadro RTX 5000. Each experiment is run 3 times with different random seeds and we display the average of the tests with a bold line, the limits of the shadow area are given by the maximum and the minimum over the runs. When displaying the training loss or the step histories, an exponential moving average with a factor 0.99 is applied in order to smooth the curves and gain in visibility. Note that the training and testing loss functions are displayed with the L^2 regularization term. On all figures the x-axis is the number of epochs. Remember however that the computational cost is not the same for the different optimizers, see Theorem 1.

4.1.2 Interpretation of the RED experiments

In this first set of experiments, we compare the RED method given in Algorithm 3 with standard SGD and RMSProp. In order to recover these two latter algorithms, set $r_k = 1$ in line 11 of Algorithm 3. The hyperparameters, namely the initial learning rate ℓ and its decay factor η, are optimized on the training loss with a grid search over the 20% first epochs, these algorithms are coined as "standard algorithms". The results are displayed in Figure 2 for the standard algorithms (orange for SGD, blue for RMSProp) and their RED version (red for SGD, green for RMSProp).

Training loss The analysis of the training loss shows that RED is competitive to the standard SGD and RMSProp methods. Note however that the hyperparameters of the standard methods have been chosen as to optimize the behavior of the training loss, hence we cannot expect the RED method to outperform the manually-tuned methods.

Step We always observe an increase in the step for the first few epochs (50 for MNIST, 10 for CIFAR). This step increase coincides with the important decrease of the training and testing loss functions. We interpret this behavior as a search for a basin of attraction of a local minimum. It should be noted that the step of the standard CIFAR100 and autoencoder is an order of magnitude smaller than their RED counterpart. Indeed larger steps on these methods cause the algorithm to diverge. This seems to indicate that the stage of the first 10 epochs where the step is small is of importance and is well captured by the RED algorithm. Note that this behavior is the one that is implemented when using warm-up techniques [26]. The analysis of the step seems to showcase the power of adaptive rescaling and indicate that warm-up techniques can be handled by the rescaling. This potential is investigated in Section 4.2.

Testing loss and accuracy The rescaling aims at minimizing quickly the training loss, no conclusions can be drawn from the analysis of the test dataset. Nevertheless, on the CIFAR experiments, an overfitting phenomenon starts from the 25th epoch approximatively. The overfitting is clearer and more pronounced on the RED method. This is in accordance with the analysis of the step size: the rescaled method seems to have converged to the maximum of the expressivity of the network at the 50th epoch. Concerning the accuracy, it is well known that adaptive methods have poor generalization performances in the overparameterized setting in comparison to SGD [46]. Indeed the standard RMSProp achieves lower performance on the test dataset of CIFAR100. Surprisingly, the RED-RMSProp algorithm does not have this property.

As a conclusion of these tests, RED, which is a naive implementation of convergence/exploration trade-off works surprisingly well on this benchmark. We purposely disconnected Robbins-Monro decay rule and let the algorithm run way past overfitting. It still exhibits good convergence properties.

4.1.3 Other numerical tests

In Appendix C.1 we investigate the use of momentum with SGD and Adam on the CIFAR100 classifier. The proposed method is only available to deal with direction of descent and the directions of update given by momentum based algorithms are not necessarily direction of descent, yielding poor convergence results.

In Appendix C.2, we perform tests with smaller batches and we exhibit pathological cases where the rescaled method is highly impacted by stochasticity. The main conclusion is that the performance of the method collapses when the batch is too small compared to the number of classes. This problem in the curvature computation arises at the last layer of the neural network (linear classifier).

In Appendix C.3 we study the effet of the L_2 regularization on the CIFAR10 classifier, showing numerically that the potential theorical issues raised in Section 2.2 do not impede convergence.

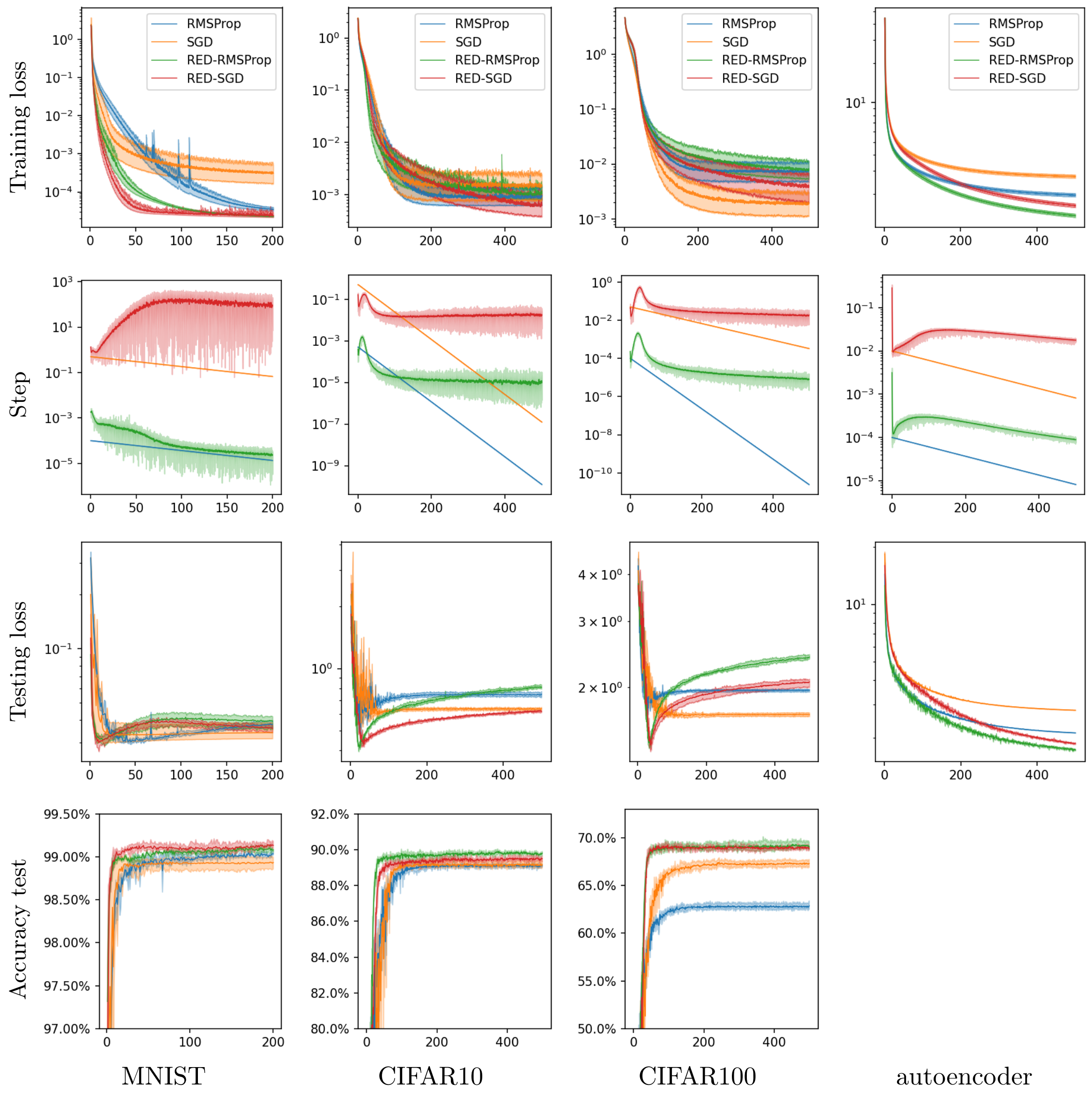


Figure 2: Training loss, step size, testing loss and test accuracy for the RED and manually-tuned SGD and RMSProp optimizers. Each column gives the different test cases (resp. MNIST, CIFAR10, CIFAR100 and autoencoder). The RED method which has no tuning gives competitive results in comparison with the manually-tuned SGD and RMSProp optimizers.

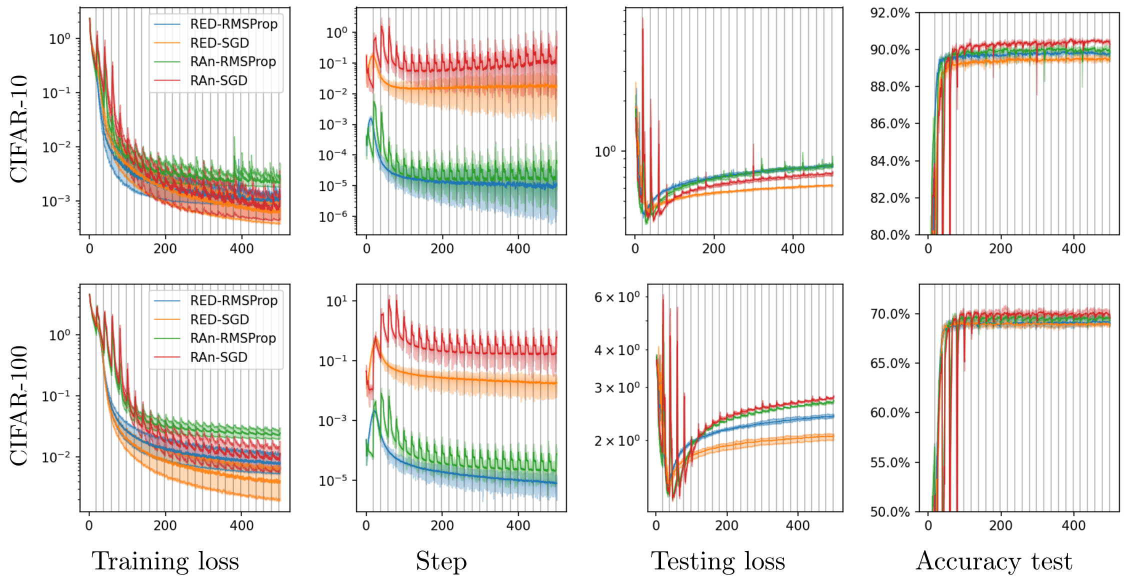


Figure 3: Annealing (RAn) vs Exponential decay (RED) method. The annealing method increases the loss functions during the hyperexploration ($\ell = 2$) phase (after the vertical gray lines). This empirically proves that the factor $\ell = 1$ is the limiting factor that allows exploration without increasing the loss function. The basin of attraction of the RAn method is different of the one of RED, except possibly for CIFAR10 with RMSProp.

Finally, in Appendix C.4 we perform some comparisons with an existing BB method [8] and with a SGD with Robbins-Monro decay condition.

4.2 Hyperexploration mode

In order to showcase hyperexploration, we propose a vanilla annealing method. We replace in Algorithm 3 (RED) the line 13 (update of the parameter ℓ) by setting periodically $\ell = 1$ for 5 epochs, $\ell = \frac{1}{2}$ for 13 epochs and $\ell = 2$ for 2 epochs. These three phases are coined respectively as *exploration*, *convergence* and *hyperexploration*. We favor sharp changes when letting ℓ oscillate in order to easily interpret the results. This simple annealing method is coined RAn (Rescaled Annealing). We display in Figure 3 the results for CIFAR10 and CIFAR100. On Figure 3 the shift between the choice $\ell = \frac{1}{2}$ and $\ell = 2$ is represented by a vertical gray line. We also display the results for the RED algorithm for comparaison. Of importance in Figure 3 is the behavior of the loss function. The latter increases at each *hyperexploration* phase, and converges during the *exploration* and *convergence* phase. A similar effect is also present but less pronounced on the testing loss and accuracy. The increase of the training loss function for $\ell = 2$ is in accordance with the theory, and is at the core of annealing methods that aim at escaping local minima. These tests validate the fact that $\ell = 1$ is an upper-bound for the *exploration* choice.

4.3 Hyperconvergence mode

In this example, we wish to study a more realistic dataset for which stochastic issues are of essence. To that end, we use the ImageNet 1K database and load a state-of-the-art pretrained ResNet-50. This network achieves a 80.858% top-1 accuracy and a 95.434% top-5 accuracy. From the study of Appendix C.2, summarized in Section 4.1.3, we know that important stochastic problems will occur in the last layer (Linear Classifier or LC) of the DNN. Hence, we erase the parameters of the linear classifier and aim at re-training it while freezing the weights of the feature extractor (upstream section of the network). This setting is reminiscent of a toy *transfer learning* problem and aims at training a simple neural network with a state-of-the-art dataset.

From the coupon collector's problem with 1000 classes, we know that the expectation of T, the smallest batch size that obtains at least one element in each class, is approximatively 7.3K when

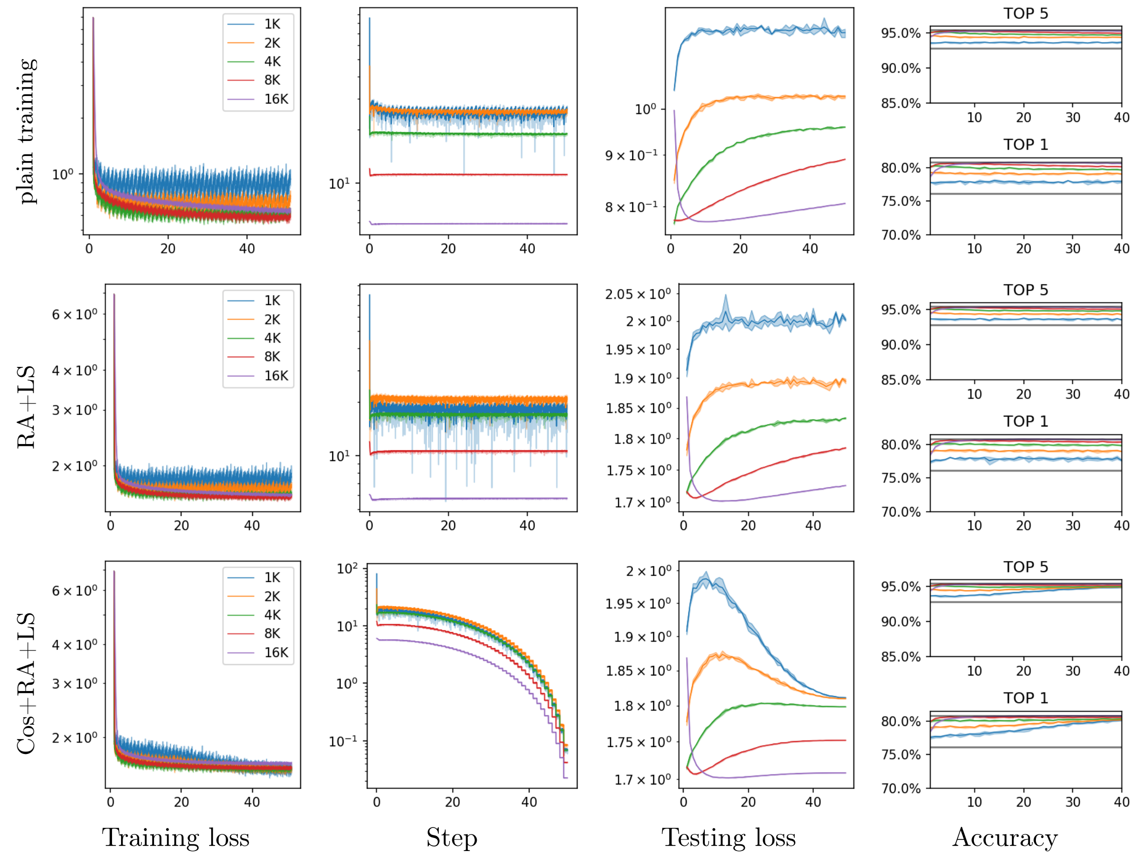


Figure 4: Training a linear classifier on ImageNet 1K with a ResNet-50 feature extractor and with different batch size for SGD.

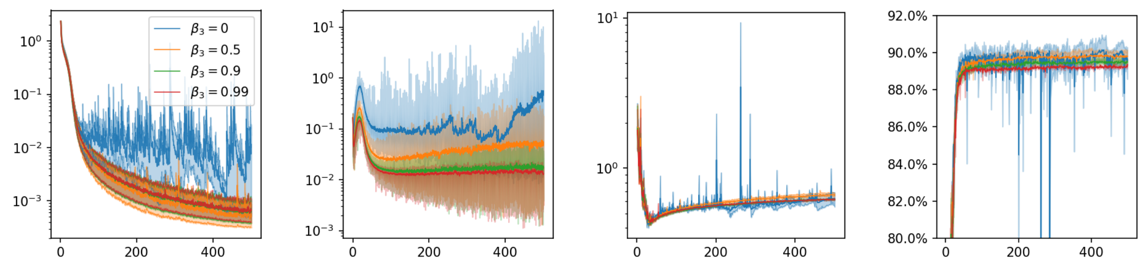
classes are drawn independently and uniformly. We expect stochastic issues to appear when batches are of size smaller than 7.3K. The batch size used to pretrain the network is 1K, hence we test the rescaling for batch of size 1K, 2K, 4K, 8K and 16K. Since stochastic effects should be seriously mitigated for the 8K and 16K cases, these two cases represent a baseline for the training.

We first discard every trick and test the rescaling for the different batch size. We adopt a fixed rescaled learning rate strategy of $\ell = \frac{1}{2}$ in order to converge as fast as possible. The result is given in Figure 4 top line and referred as *plain training*. Of importance in the top line of Figure 4 are the oscillations in the training loss, which are less pronounced as the batch size increases. Note also the stability of the top-1 and top-5 accuracies around a value that depends on the batch size. For the 16K experiment, the linear classifier achieves the top-1 and top-5 accuracies of the pre-trained weigths.

We then implement several tricks of the trade, namely repeated augmentation (RA) [17] and label smoothing (LS) [41]. The result are displayed in Figure 4, middle line. These two tricks do not seem to have any effect on the training of the linear classifier.

In the bottom line of Figure 4, we implement a decrease of the learning rate with a Cosine annealing (Cos) technique, in addition to (RA) and (LS). The (Cos) technique reduces the learning rate and enforces the *hyperconvergence mode*. As far as the accuracies are concerned, reducing the learning rate allows the algorithm to converge when the batches are small and is useless when the batch size is greater than 8K. This test corroborates the findings of [40] and the tests of Appendix C.2.

In this benchmark, one of the important advantages of rescaling is to be able to perform several tests (batch reduction, repeated augmentation, label smoothing) without having to set the learning rate for each test.



(a) Training loss (b) Step (c) Testing loss (d) Accuracy test

Figure 5: Training loss, step size, testing loss and test accuracy on the CIFAR10 classifier with RED-SGD. The tests are conducted with different values of the curvature averaging parameter β_3. A value $\beta_3 = 0$ yields instability and $\beta_3 \in [0.5, 0.99]$ has little impact on the convergence rate.

4.4 Influence of the averaging factor of the curvature

This section is dedicated to the study of the impact of the averaging factor of the curvature β_3 on the algorithm. A low value $\beta_3 \simeq 0$ yields an estimation of the curvature that is less dependent of the past iterations at the expense of having a higher variance. A value close to 1 results in a low variance estimation but that has a bias due to old iterations. In Figure 5, the CIFAR10 classifier is optimized using RED-SGD with values of $\beta_3 \in \{0, 0.5, 0.9, 0.99\}$. Interestingly, the parameter that gives the fastest increase of the test accuracy is $\beta_3 = 0$ at the cost of more instabilities. Although higher values of β_3 lead to an underestimation of the step size, the difference of performance on the training loss is insignificant. Overall, a value $\beta_3 \in [0.5, 0.99]$ has little impact on the convergence rate of the algorithm and a default value of $\beta_3 = 0.9$ can be considered.

5 Conclusion and discussion

We developed a framework that allows automatic rescaling of the learning rate of a descent method with the use of the curvature, which is an easily affordable second order information computed by automatic differentiation. This rescaling yields a data and direction adapted learning rate with a physical meaning. The practitioner can choose the behavior of the algorithm by setting the value of this rescaled learning rate. A value between 1/2 and 1 results in convergence, a value above 1 yields hyperexploration of the space of parameters and a value below 1/2 enforces convergence when stochasticity is of importance.

In the numerical examples of Section 4.1 a choice of exponential decrease is competitive to simple manual tuning of the learning rate in the case of SGD and RMSProp preconditioning. In Section 4.2, we show that the choice $\ell > 1$ allows escaping basin of attraction of local minima. The more intricated benchmark of Section 4.3 show that rescaling doesn't save us from reducing the learning rate but that it allows to control the environment and compare different experiments.

The main limitation of this method is that it does not allow use of momentum. Indeed momentum methods do not necessarily yield directions of descent and do rely on per-iteration minimization of Lyapunov functions [32]. Implementing momentum methods with curvature computation is a challenge reserved for future works. Another drawback is the need to use $\mathcal{C}^2$ activation functions, notably excluding ReLU. Finally, the curvature computation, also affordable in theory, requires additional implementations on top of ready-to-use machine learning librairies, which restricts, for now, our method to rather simple networks.

Acknowledgement

This work was supported by the ANR Micro-Blind. F. de Gournay acknowledges the support of AI Interdisciplinary Institute ANITI funding, through the French “Investing for the Future— PIA3” program under the Grant Agreement ANR-19-PI3A-0004. This work was performed using HPC resources from GENCI-IDRIS (Grant 2021-AD011012210R1).

References

[1] Zeyuan Allen-Zhu. Natasha 2: Faster non-convex optimization than sgd. *Advances in neural information processing systems*, 31, 2018.

[2] F Alvarez and A Cabot. Steepest descent with curvature dynamical system. *Journal of optimization theory and applications*, 120(2):247–273, 2004.

[3] Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

[4] Jonathan Barzilai and Jonathan M Borwein. Two-point step size gradient methods. *IMA journal of numerical analysis*, 8(1):141–148, 1988.

[5] Sue Becker and Yann Le Cun. Improving the convergence of back-propagation learning with second order methods. Technical Report CRG-TR-88-5, Department of Computer Science, University of Toronto, 1988.

[6] Fahimeh Biglari and Maghsud Solimanpur. Scaling on the spectral gradient method. *Journal of Optimization Theory and Applications*, 158(2):626–635, 2013.

[7] Yair Carmon, John C Duchi, Oliver Hinder, and Aaron Sidford. "convex until proven guilty": Dimension-free acceleration of gradient descent on non-convex functions. In *International Conference on Machine Learning*, pages 654–663. PMLR, 2017.

[8] Camille Castera, Jérôme Bolte, Cédric Févotte, and Edouard Pauwels. Second-order step-size tuning of sgd for non-convex optimization. *Neural Processing Letters*, pages 1–26, 2022.

[9] Bruce Christianson. Automatic hessians by reverse accumulation. *IMA Journal of Numerical Analysis*, 12(2):135–150, 1992.

[10] Frank E Curtis and Daniel P Robinson. Exploiting negative curvature in deterministic and stochastic optimization. *Mathematical Programming*, 176(1):69–94, 2019.

[11] Yuhong Dai, Jinyun Yuan, and Ya-Xiang Yuan. Modified two-point stepsize gradient methods for unconstrained optimization. *Computational Optimization and Applications*, 22(1):103–109, 2002.

[12] Alexandre Défossez, Léon Bottou, Francis Bach, and Nicolas Usunier. A simple convergence proof of adam and adagrad. *arXiv preprint arXiv:2003.02395*, 2020.

[13] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[14] Robert Gower, Nicolas Le Roux, and Francis Bach. Tracking the gradients using the hessian: A new look at variance reducing stochastic methods. In *International Conference on Artificial Intelligence and Statistics*, pages 707–715. PMLR, 2018.

[15] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation.* SIAM, 2008.

[16] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[17] Elad Hoffer, Tal Ben-Nun, Itay Hubara, Niv Giladi, Torsten Hoefler, and Daniel Soudry. Augment your batch: better training with larger batches. *arXiv preprint arXiv:1901.09335*, 2019.

[18] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. *Advances in neural information processing systems*, 26, 2013.

[19] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[20] Shankar Krishnan, Ying Xiao, and Rif A Saurous. Neumann optimizer: A practical optimization algorithm for deep neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[21] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Pennsylvania State University, 2009.

[22] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. at&t labs, 2010.

[23] Ting Li and Zhong Wan. New adaptive barzilai–borwein step size and its application in solving large-scale optimization problems. *The ANZIAM Journal*, 61(1):76–98, 2019.

[24] Jinxiu Liang, Yong Xu, Chenglong Bao, Yuhui Quan, and Hui Ji. Barzilai–borwein-based adaptive learning rate for deep learning. *Pattern Recognition Letters*, 128:197–203, 2019.

[25] Mingrui Liu and Tianbao Yang. On noisy negative curvature descent: Competing with gradient descent for faster non-convex optimization. *arXiv preprint arXiv:1709.08571*, 2017.

[26] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[27] Ke Ma, Jinshan Zeng, Jiechao Xiong, Qianqian Xu, Xiaochun Cao, Wei Liu, and Yuan Yao. Stochastic non-convex ordinal embedding with stabilized barzilai-borwein step size. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[28] James Martens et al. Deep learning via hessian-free optimization. In *International conference on machine learning (ICML)*, volume 27, pages 735–742, 2010.

[29] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[30] Yann Ollivier. Riemannian metrics for neural networks i: feedforward networks. *Information and Inference: A Journal of the IMA*, 4(2):108–153, 2015.

[31] Barak A Pearlmutter. Fast exact multiplication by the hessian. *Neural computation*, 6(1):147–160, 1994.

[32] Boris Polyak and Pavel Shcherbakov. Lyapunov functions: An optimization theory perspective. *IFAC-PapersOnLine*, 50(1):7456–7461, 2017.

[33] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

[34] Marcos Raydan. The barzilai and borwein gradient method for the large scale unconstrained minimization problem. *SIAM Journal on Optimization*, 7(1):26–33, 1997.

[35] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[36] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[37] Nicolas Roux, Pierre-Antoine Manzagol, and Yoshua Bengio. Topmoumoute online natural gradient algorithm. *Advances in neural information processing systems*, 20, 2007.

[38] Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. In *International conference on machine learning (ICML)*, pages 343–351. PMLR, 2013.

[39] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[40] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[41] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[42] Conghui Tan, Shiqian Ma, Yu-Hong Dai, and Yuqiu Qian. Barzilai-borwein step size for stochastic gradient descent. *Advances in neural information processing systems*, 29, 2016.

[43] Tijmen Tieleman and G Hinton. Divide the gradient by a running average of its recent magnitude. coursera neural netw. *Mach. Learn*, 6:26–31, 2012.

[44] Oriol Vinyals and Daniel Povey. Krylov subspace descent for deep learning. In *Artificial intelligence and statistics*, pages 1261–1268. PMLR, 2012.

[45] Andrea Walther. Computing sparse hessians with automatic differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 34(1):1–15, 2008.

[46] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. *Advances in neural information processing systems*, 30, 2017.

[47] Yunhai Xiao, Qiuyu Wang, and Dong Wang. Notes on the dai–yuan–yuan modified spectral gradient method. *Journal of computational and applied mathematics*, 234(10):2986–2992, 2010.

[48] Zhuang Yang, Cheng Wang, Zhemin Zhang, and Jonathan Li. Random barzilai–borwein step size for mini-batch algorithms. *Engineering Applications of Artificial Intelligence*, 72:124–135, 2018.

[49] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael W Mahoney. Adahessian: An adaptive second order optimizer for machine learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

[50] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

A Second order computation

A.1 Hessian-vector dot product

In this section, we turn our attention to showing how to compute $\nabla^2\mathcal{J}(\Theta)\dot{\Theta}$ in our setting. The results are known as the *Pearlmutter's trick* [9, 31]. We emphasize that the computation of the curvature is simplier than the Hessian-vector product. In our setting, the trick that allows the computation of the Hessian-vector product is based on the following ideas

- The mapping $\dot{\Theta} \mapsto \frac{1}{2}\langle \nabla^2\mathcal{J}(\Theta)\dot{\Theta}, \dot{\Theta}\rangle$ is bilinear. If we differentiate with automatic differentiation this mapping with respect to $\dot{\Theta}$, we retrieve $\nabla^2\mathcal{J}(\Theta)\dot{\Theta}$.
- Because $X(\Theta)$ is fixed, the aforementioned mapping is defined by a single forward recurrence. Hence, only one additional backward recurrence should be sufficient to compute $\nabla^2\mathcal{J}(\Theta)\dot{\Theta}$.

In order to make explicit this backward recurrence, we need to introduce two vectors $A_s \in \mathcal{H}_s$ and $B_s \in \mathcal{G}_s$ that are defined by the implicit equation:

$$\langle A_s, a\rangle_{\mathcal{H}_s} + \langle B_s, b\rangle_{\mathcal{G}_s} = \langle \nabla^2\mathcal{F}_s(\dot{x}_s, \dot{\theta}_s) \otimes (a,b), \hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}} \quad \forall (a,b) \in \mathcal{H}_s \times \mathcal{G}_s.$$

The existence and uniqueness of (A_s, B_s) is just Riesz theorem applied to the linear form on $\mathcal{H}_s \times \mathcal{G}_s$:

$$(a,b) \mapsto \langle \nabla^2\mathcal{F}_s(\dot{x}_s, \dot{\theta}_s) \otimes (a,b), \hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}}.$$

Construct $\tilde{X} = (\tilde{x}_s)_s$ and $\tilde{\Theta} = (\tilde{\theta}_s)_s$ by a backward recurrence using

$$\begin{cases} \tilde{x}_s = (\partial_x \mathcal{F})^* \tilde{x}_{s+1} + A_s & \text{with } \tilde{x}_{n+1} = 0 \\ \tilde{\theta}_s = (\partial_\theta \mathcal{F})^* \tilde{x}_{s+1} + B_s. \end{cases} \tag{23}$$

Then we have

$$\nabla^2\mathcal{J}(\Theta)\dot{\Theta} = \tilde{\Theta} \tag{24}$$

In order to prove (24), we show that for any other direction $\dot{\Theta}'$, we have

$$\langle \nabla^2\mathcal{J}(\Theta)\dot{\Theta}, \dot{\Theta}'\rangle = \langle \tilde{\Theta}, \dot{\Theta}'\rangle$$

First consider $\dot{X}'$ the tangent associated with direction $\dot{\Theta}'$. We have by bilinearity of $\nabla^2\mathcal{F}_s$ and by Algorithm 2 that

$$\langle \nabla^2\mathcal{J}(\Theta)\dot{\Theta}, \dot{\Theta}'\rangle = \sum_s \langle \nabla^2\mathcal{F}_s(\dot{x}_s, \dot{\theta}_s) \otimes (\dot{x}'_s, \dot{\theta}'_s), \hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}} = \sum_s \langle A_s, \dot{x}'_s\rangle + \langle B_s, \dot{\theta}'_s\rangle \tag{25}$$

By definition of $\tilde{X}$ and $\tilde{\Theta}$ in (23) and by formula (16) for the tangent $\dot{X}'$, the following equality holds:

$$\begin{aligned} &\langle \tilde{x}_s, \dot{x}'_s\rangle + \langle \tilde{\theta}_s, \dot{\theta}'_s\rangle - \langle A_s, \dot{x}'_s\rangle - \langle B_s, \dot{\theta}'_s\rangle \\ &= \langle (\partial_x \mathcal{F})^* \tilde{x}_{s+1}, \dot{x}'_s\rangle + \langle (\partial_\theta \mathcal{F})^* \tilde{x}_{s+1}, \dot{\theta}'_s\rangle = \langle \tilde{x}_{s+1}, \dot{x}'_{s+1}\rangle \end{aligned}$$

Summing up the above equation for every s, using $\tilde{x}_{n+1} = 0$, $\dot{x}'_0 = 0$ and (25) yields (24)

A.2 Structure of the layers

In this section, we explain how to compute the curvature for some of the standard layers used in DNNs. First, we make clear the different kind of layers we use:

- **Loss layers** are parameter-free layers from $\mathcal{H}_n$ to $\mathbb{R}$, they are denoted by $\mathcal{L}$

$$\mathcal{F}_n(x, \theta) = \mathcal{L}(x).$$

- **Smooth activation layers** do not have parameters and are such that $\mathcal{H}_{s+1} = \mathcal{H}_s$. They are defined coordinate-wise through a smooth function $\Phi_s : \mathbb{R} \to \mathbb{R}$ with

$$\mathcal{F}_s(x, \theta)[i] = \Phi_s(x[i]) \quad \forall i.$$

Name	$x_{s+1}[i]$	$\hat{x}_s[j]$	$\hat{\theta}_s[k]$
Activation	$\Phi(x_s[i])$	$\Phi'(x_s[j])\hat{x}_{s+1}[j]$	N.A.
Linear	$\sum_{k,j}\theta[k]x_s[j]\mathbb{1}_{ijk}$	$\sum_{k,i}\theta[k]\hat{x}_{s+1}[i]\mathbb{1}_{ijk}$	$\sum_{j,i}\hat{x}_{s+1}[j]x_s[i]\mathbb{1}_{ijk}$
Bias	$x_s[i]+\sum_k \mathbb{1}_{ik}\theta[k]$	$\hat{x}_{s+1}[j]$	$\sum_i \mathbb{1}_{ik}\hat{x}_{s+1}[i]$
Centering	$x_s[i]-\mathbb{E}_i(x_s)$	$\hat{x}_{s+1}[j]-\mathbb{E}_j(\hat{x}_{s+1})$	N.A.
Normalizing	$\begin{cases}\gamma=(\mathbb{E}_i(x_s^2)+\varepsilon)^{-1/2}\\ x_{s+1}[i]=\gamma x_s[i]\end{cases}$	$\gamma\hat{x}_{s+1}[j]-x_s\mathbb{E}_j[\gamma^3 x_s\hat{x}_{s+1}]$	N.A.

Name	$\dot{x}_{s+1}[i]$	$r_s=\frac{1}{2}\langle\nabla^2\mathcal{F}_s(\dot{x}_s,\dot{\theta}_s)\otimes(\dot{x}_s,\dot{\theta}_s),\hat{x}_{s+1}\rangle_{\mathcal{H}_{s+1}}$
Activation	$\Phi'(x_s[i])\dot{x}_s[i]$	$\frac{1}{2}\sum_i\Phi''(x_s[i])\dot{x}_s^2[i]\hat{x}_{s+1}[i]$
Linear	$\sum_{k,j}\left(\theta[k]\dot{x}_s[j]+\dot{\theta}[k]x_s[j]\right)\mathbb{1}_{ijk}$	$\sum_{k,j,i}\dot{\theta}[k]\dot{x}_s[i]\hat{x}_{s+1}[j]\mathbb{1}_{ijk}$
Bias	$\dot{x}_s[i]+\sum_k\mathbb{1}_{ik}\dot{\theta}[k]$	0
Centering	$\dot{x}_s[i]-\mathbb{E}_i(\dot{x}_s)$	0
Normalizing	$\begin{cases}\dot{\gamma}=-\mathbb{E}_i[\dot{x}_s x_s]\gamma^3\\ \dot{x}_{s+1}[i]=\gamma\dot{x}_s[i]+\dot{\gamma}x_s[i]\end{cases}$	$\begin{cases}\ddot{\gamma}=-\mathbb{E}_i(\dot{x}_s^2)\gamma^3+3\mathbb{E}_i(\dot{x}_s x_s)\gamma^5\\ r_s=\sum_i\frac{1}{2}\left(\dot{\gamma}\dot{x}_s[i]+\ddot{\gamma}x_s[i]\right)\hat{x}_{s+1}[i]\end{cases}$

Table 1: Quantities needed in the forward, backward and second order passes for standard layers.

- **Linear layers or convolutional layers**. The set of parameters are the weights (or kernel) denoted θ. We suppose that these layers have no bias. They are abstractly defined as

$$\mathcal{F}_s(x,\theta)[i]=\sum_{k,j}\theta[k]x[j]\mathbb{1}_{ijk}$$

  where i (resp j,k) denotes the sets of indices of the outputs (resp. the input, the weights). The function $(i,j,k)\mapsto\mathbb{1}_{ijk}$ represents the assignment of the multi-index (k,j) to i. This affectation is either equal to 1 or 0, that is $(\mathbb{1}_{ijk})^2=\mathbb{1}_{ijk}$.

- **Bias layers** are layers where $\mathcal{H}_s=\mathcal{H}_{s+1}$ and are defined by

$$\mathcal{F}_s(x,\theta)[i]=x_s[i]+\sum_k\mathbb{1}_{ik}\theta[k].$$

  They are often concatenated with linear or convolutional layers. There is no restriction to split a biased linear layer into the composition of a linear layer and a bias layer.

- **Batch normalization layers.** We split a batch normalization layer into the composition of four different layers, the centering layer, the normalizing layer, a linear layer with diagonal weight matrix and a bias layer. For each output index i, the centering and normalizing layers are defined by an expectation over the batch and some input indices. This expectation is denoted as $\mathbb{E}_i$. The centering layer can be written as

$$\mathcal{F}_s(x,\theta)[i]=x_s[i]-\mathbb{E}_i(x_s).$$

  The normalizing layer is defined as

$$\mathcal{F}_s(x,\theta)[i]=\frac{x[i]}{\sqrt{\mathbb{E}_i(x^2)+\varepsilon}}.$$

For the different layers, we give the formula for the different recurrences in Table 1. We begin with the classic backward computations, they are mainly given here to settle the notations.

B Description of the numerical experiments

All the experiments were conducted and timed using Python 3.8.11 and PyTorch 1.9 on an Intel(R) Xeon(R) W-2275 CPU @ 3.30GHz with an NVIDIA Quadro RTX 5000 GPU. We also used the Jean-Zay HPC facility for additional runs.

Dataset	MNIST	CIFAR10	CIFAR100
License	CC BY-SA 3.0	MIT License	Unknown
Size of the training set	60000	50000	50000
Size of the testing set	10000	10000	10000
Number of channels	1	3	3
Size of the images	28×28	32×32	32×32
Number of classes	10	10	100

Table 2: Summary of the datasets used.

Type of problem	MNIST classification	CIFAR10 classification	CIFAR100 classification	MNIST autoencoder
Type of network	LeNet Dense	VGG11 Convolutional	VGG19 Convolutional	Dense
Activation functions	Tanh	SoftPlus $\beta = 5$	SoftPlus $\beta = 5$	ELU
L^2 regularization	$\lambda = 10^{-7}$	$\lambda = 10^{-7}$	$\lambda = 10^{-7}$	$\lambda = 10^{-7}$
Loss function	Cross entropy	Cross entropy	Cross entropy	MSE
Number of epoch	200	500	500	500
Batch size	256	256	256	256
Number of epoch for tuning	40	100	100	100
Iteration per epoch	196	235	235	196
Computing time per epoch with the standard SGD / RMSProp	5.3s / 5.4s	16.4s / 16.8s	36.3s / 36.9s	5.1s / 5.3s
Computing time per epoch with RED-SGD / RED-RMSProp	7.6s / 7.7s	30.1s / 30.5s	65s / 65s	5.7s / 5.9s

Table 3: Summary of the experiment parameters.

The models are trained with a batch size of 256 so that one epoch corresponds to 235 iterations for MNIST and 196 for CIFAR. The number of epochs is set to 200 for the MNIST classification and 500 for the others. Table 2 summarizes the characteristics of the datasets used.

Concerning the tuning of the standard methods, the step size and its decay factor were searched on a grid for the SGD and RMSProp methods. The learning rate is constant per epoch and its value at the n^{th} epoch is given by

$$\tau_n = \tau_0 d^n.$$

We searched amongst the values $\tau_0 \in \{1{\times}10^n, 5{\times}10^n\}_{-5\leq n\leq 1}$ for the step size and $d \in \{0.97, 0.98, 0.99, 1\}$ for the step decay on MNIST classification and $d \in \{0.99, 0.995, 1\}$ for the others. After 20% of the total number of epochs, the couple (τ_0, d) that achieves the best training loss decrease is chosen.

For the CIFAR experiments we used data augmentation with a random crop and an horizontal flip. In the CIFAR100 training we added a random rotation of at most $\pm 15°$.

For reproductibility, the values used in the experiments are summarized in Table 3. Unless explicitely stated, these are the default values used in the experiments of this work. The computing time per epoch is reported in Table 3 for each method. The codes of RED are not optimized, especially for the convolution layers where the backward with respect to the parameters is implemented by an additional run of the forward. This explains why RED is twice slower than the standard methods on the CIFAR classifiers which make intensive use of convolution layers.

C Additional numerical experiments

C.1 Dealing with momentum

In Section 4.1, only stochastic optimizers without momentum are presented. In this section, we discuss the extension of our algorithm to momentum based update directions, notably momentum with RMSProp preconditioning which is the celebrated Adam algorithm [19].

Incorporating momentum consists in replacing the gradient by an exponential moving average of the past iterates of the gradients with a parameter $\beta_1 \in [0, 1[$. In our setting, it amounts to replacing line 10 of Algorithm 3 by lines 4 and 5 of Algorithm 4.

Algorithm 4 Adding momentum to RED

```
1: Initialization $\hat{g}_0 = 0$.
2: for $k = 1..$ do
3:     $\cdots$
4:     $\hat{g}_k = \beta_1 \hat{g}_{k-1} + (1-\beta_1) g_k$     and     $\tilde{g}_k = \hat{g}_k / (1 - \beta_1^k)$
5:     $\dot{\Theta}_k = P_k^{-1} \tilde{g}_k$
6:     $\cdots$
7: end for
```

Momentum was introduced by Polyak [33] in the convex non-stochastic setting. It can be interpreted as an adaptation of a convex method to a non-convex stochastic problem. We coin this explanation as the *heavy-ball* analysis. Another point of view, which we denote as *variance reduction*, is that the exponential moving average $\tilde{g}_k$ is a better estimator of $\nabla \mathcal{J}(\Theta_k)$ than g_k. Indeed all the previous batches $(\mathcal{B}_m)_{m \leq k}$ are taken into account in the computation of $\tilde{g}_k$. The downside is that the averaged quantity is $\nabla \mathcal{J}_{\mathcal{B}_m}(\Theta_m)$ and not $\nabla \mathcal{J}_{\mathcal{B}_m}(\Theta_k)$, this introduces a bias in the estimation of $\nabla \mathcal{J}(\Theta_k)$. With this interpretation in mind, the parameter β_1 which drives the capacity of the exponential moving average to forget the previous iterations has to be tuned between the mini-batches gradient variance (high variance leads to high β_1) and the convergence (high values of $\|\Theta_k - \Theta_{k-1}\|$ lead to low choice of β_1). In [19], the authors propose to solve this dilemna by taking decaying values of β_1, although in practice, the parameter β_1 is constant.

Momentum: heavy ball or variance reduction? When momentum is understood as an heavy-ball method, at iteration k there are no reasons for $-\dot{\Theta}_k$ to be a direction of descent. Because our algorithm relies on the assumption that $-\dot{\Theta}_k$ is a direction of descent to choose a step, our analysis falls apart and RED should be used with care. On the other hand, if momentum is a variance reduction technique, the step has to be taken small enough in order not to bias the gradient estimation. With this latter assumption, RED can be applied.

In order to determine if, in our case, momentum acts as an heavy ball method or as a variance reduction technique, we study numerically when the standard Adam and SGD with momentum optimizers yield a direction of descent. In Figure 6 first row, the test of CIFAR100 in Section 4.1.2 is performed with a momentum $\beta_1 = 0.9$ and the hyperparameters were tuned using the same policy (see Appendix B). We display in the last column of Figure 6 first row, the percentage of direction of descent per epoch with respect to the current batch $\mathcal{B}_k$. If n is the epoch number and $\mathcal{K}_n$ the set of the iterations that are in epoch n, this percentage is given by:

$$q_n = \frac{1}{|\mathcal{K}_n|} \sum_{k \in \mathcal{K}_n} \mathbb{1}_{\langle g_k, \dot{\Theta}_k \rangle \geq 0} \tag{26}$$

We observe that on classification problems, SGD with momentum and more particularly Adam yield directions of update that are not direction of descent for $\mathcal{J}_{\mathcal{B}_k}$.

Step choice The RED algorithm needs a rule to deal with update directions which are not directions of descent. One possibility is to allow negative steps, which we discard since this would annihilate the heavy-ball effect. Another possibility, which we retain, is to take the absolute value of $\tau_k^\star$ in line 12. In a nutshell, compute the step for the opposite direction (which is a direction of descent) and use this step in the current direction. This choice is arbitrary and to properly tackle

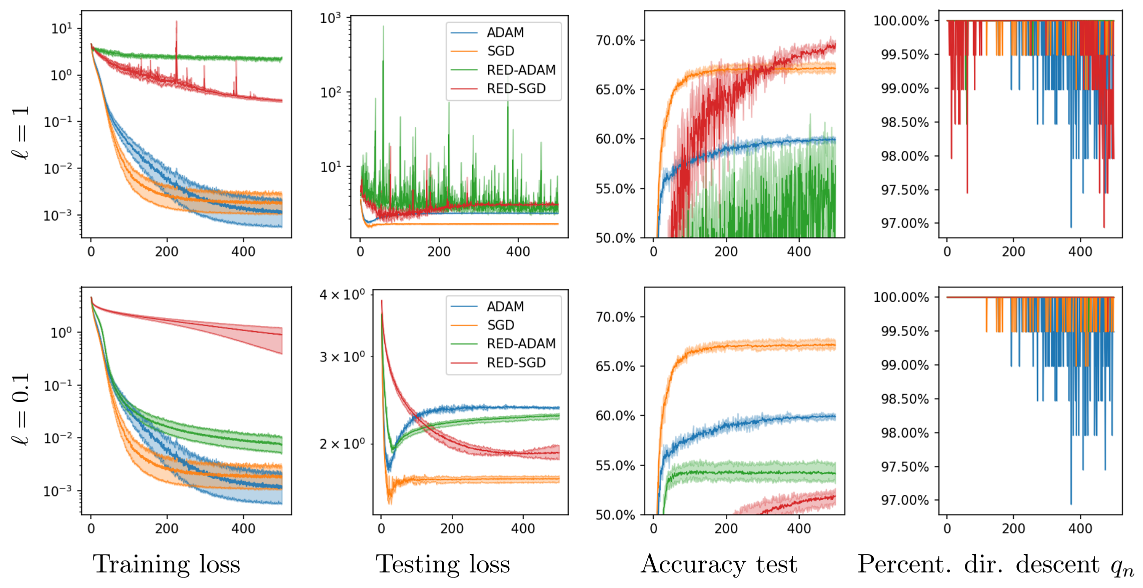


Figure 6: Tests with momentum ($\beta_1 = 0.9$) with and without the learning rate stabilization ($\ell = 1$ or $\ell = 0.1$) for RED on CIFAR100. Manually-tuned algorithms (orange for SGD, blue for Adam) and their RED version (red for SGD, green for Adam) are given. Lower learning rate in RED ensures that momentum yields direction of descent at the expense of loosing the exploration of the set of parameters.

the momentum case, interpretations using Lyapunov functions should be considered. The choice of such functions is not clear and we defer such an analysis to future work.

In Figure 6 first row the results of the optimization using RED on CIFAR100 with momentum are displayed. The parameters for the initial learning rate and its decay factor are the default ones $\ell = 1$ and $\eta = 1/2$. The RED algorithm has difficulties to converge both on the training and testing losses. We observed that the steps chosen by RED are several orders of magnitude higher than the ones obtained by manual tuning. On classification problems, RED follows directions of update that are not direction of descent.

Learning rate multiplication The impediment to using RED with momentum is that directions of update are not directions of descent. This can be solved by reducing the initial learning rate ℓ to take smaller steps τ_k so that $\|\Theta_k - \Theta_{k-1}\|$ remains small.

We propose to diminish the initial learning rate by using $\ell = 1 - \beta_1$. This choice may seem arbitrary but it is inspired by the proofs of convergence of [12] that have bounds which scale as $1 - \beta_1$. The experiments of Figure 6 last row are performed with the same set of parameters except for the initial learning rate which is set to $\ell = 0.1$. With this smaller learning rate, the algorithm is stable and converges. Of importance, RED always yield direction of descent as seen from the bottom-right of Figure 6. As ℓ was decreased, the exploration is lost, explaining these poor convergence results.

Conclusion When using momentum, decreasing the learning rate makes the experiments fit in the framework the algorithm was proposed for. As a consequence, this causes the loss of the exploration which is critical to speed-up the convergence. The correct way of dealing with momentum would be to identify the Lyapunov function that has to be minimized, which is left for future work.

C.2 Batch reduction on CIFAR

In this section, we study batch dependence on the RED method for the CIFAR datasets. Reducing the batch size mimicks harder stochastic problems while keeping the experiment in a controlled environment. In Figure 7 (column 1 and 3), we provide the results obtained for different batch size and the evolution of the training and testing loss functions per epoch. The RED parameters are an initial learning rate $\ell = 1$ and a target learning rate $\ell = \frac{1}{2}$ after 100 epochs. A striking phenomenon

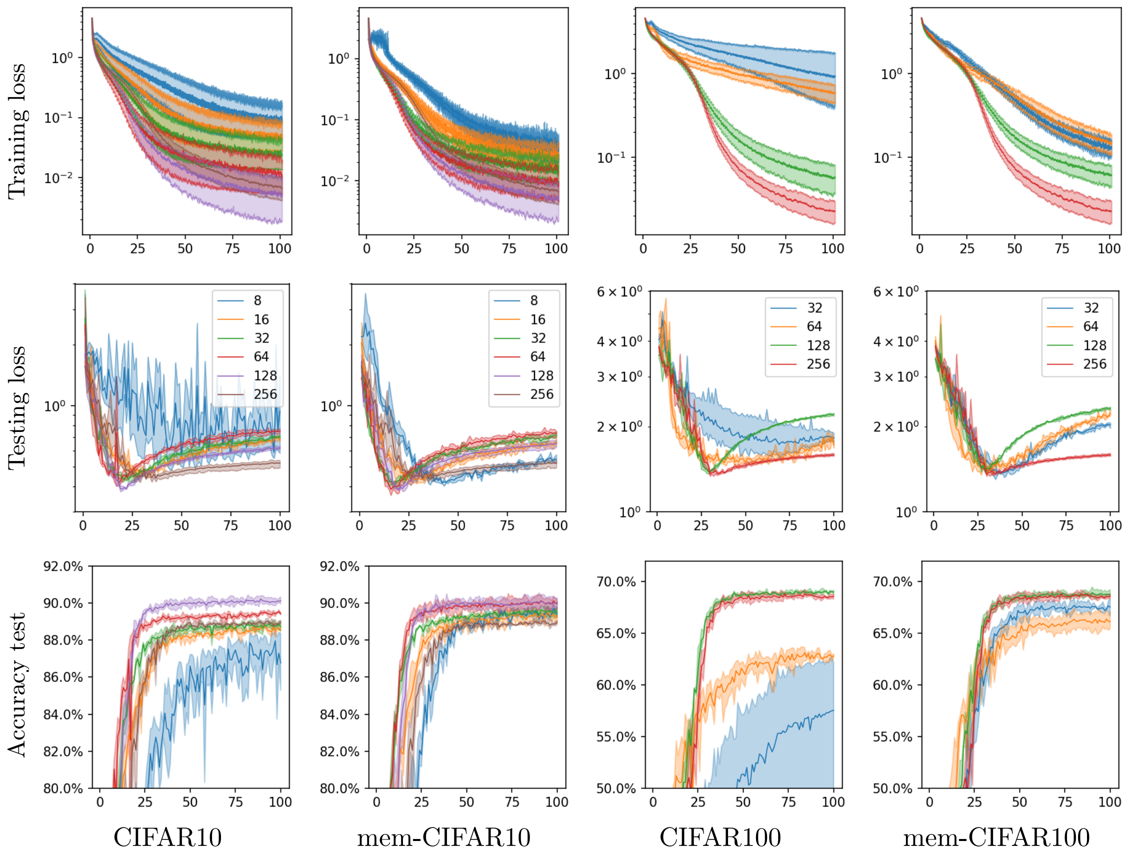


Figure 7: Batch reduction on the CIFAR10 and CIFAR100 datasets. The columns 1 and 3 are the vanilla RED-algorithm and the column 2 and 4 are the patches that (partially) solve the problem when the batch size is smaller than the number of classes.

in Figure 7 (column 1 and 3) is the loss of performance of the algorithm when the batch size is smaller than the number of classes.

Because of the relationship between the batch size and the number of classes, we wish to study if the last layer – the Linear Classifier (LC) – is the layer the most impacted by the batch size reduction. The LC optimizes the parameters of hyperplanes (one per class) which separate the information given by the remaining of the network, the Feature Extractor (FE) When the batch size is too small, some classes are not represented in the batch. The corresponding hyperplanes receive update information which is oblivious to the data of their class. We believe that this effect explains the loss of performance of the LC and a lack of precision in the computation of the curvature.

In order to verify our assumption, we implement a *memory* layer, which is set between the FE and the LC. This memory layer stores the last 256 data given by the FE. We coin this trick a memory-DNN. Because the LC is fed with this memory, it should behave as if the batch size was 256, although the memory suffers from a slight delay, due to the fact that it is not updated for the current parameters of the FE. The memory footprint and computational load of the memory-DNN is increased by a small factor, since the FE is fed with small batches and is responsible for most of the computational load and memory footprint. In Figure 7 (column 2 and 4), we collect the results of memory-DNN. Of striking importance is a better behavior of memory-DNN compared to the standard DNN when the batch size is smaller than the number of classes.

In this test, we provide a simple remedy to avoid stochasticity issues in the training of the LC in a classification problem. More important than the memory trick is the fact that rescaling the learning rate allows us to provide a unified environment to test the method. The learning rate do not have to be adapted for each experiment, which would eventually prevent us from drawing any conclusions.

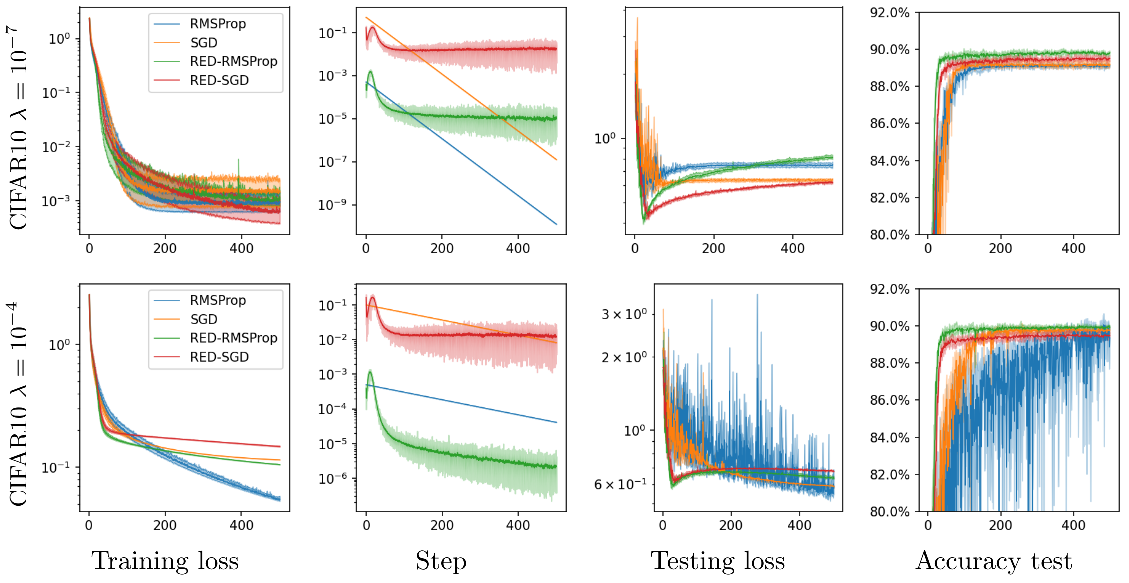


Figure 8: Influence of the L^2 regularization λ. Manually-tuned algorithms (orange for SGD, blue for RMSProp) and their RED version (red for SGD, green for RMSProp) are given. For these tests, the smaller the regularization, the best the convergence of RED.

C.3 Effect of the L^2 regularization

According to the paragraph on the L^2 regularization in Section 2.2, a regularization is introduced in our algorithm to counteract the effet of a potentially vanishing Hessian in the direction of update. This is a theoretical limitation and we study in this section the influence of this regularization on the performance of RED. We conduct the experiments of Section 4.1.2 for the CIFAR10 dataset with different values of the regularization $\lambda \in \{10^{-7}, 10^{-4}\}$. The hyperparameters of the standard SGD and RMSProp optimizers are tuned for each value of λ. We report in Figure 8 the different results, including the ones that are shown in Section 4.1.2. The grid search on the training loss that led to the choice of parameters for RMSProp and $\lambda = 10^{-4}$ yielded large step size at the cost of instabilities in the test metrics. On all test cases, we observe that a value of regularization close to zero ($\lambda = 10^{-7}$) gives good convergence results. In the considered tests, the need of a regularization seems to be more of a theoretical limitation than a practical one.

C.4 Comparison with BB and Robbins-Monro

In this section we compare our algorithm with the closest existing approach [8], named *step-tuned*, where the authors approximate the curvature with a BB method. We also compare it with a Robbins-Monro decay rule of the learning rate for SGD. We did not compare with the BB method of [48] as this method requires the computation of the gradient over the whole dataset at each epoch.

The step-tuned optimizer has several hyperparameters and we use the default ones except the learning rate as advised in [8]. The initial learning rate of [8] is searched on the same grid than the standard SGD (see Appendix B).

In Figure 9, the results of the optimization on the CIFAR10 classifier and on the autoencoder are given for two values of the L^2 regularization $\lambda \in \{10^{-7}, 10^{-4}\}$. RED algorithm is outperformed by step-tuned only on the training loss but the learning rate of step-tuned has been optimized for the training loss and we cannot expect better performance than step-tuned on this criterion. Finally, RED is more stable on every test metrics and has better generalization than step-tuned. Step-tuned [8] requires the optimization of the learning rate and because RED does not need any hyperparameter adjustment, our method is competitive with this existing work. Note also that step-tuned is not available with the RMSProp preconditioner, when RED handles any kind of preconditioning technique.

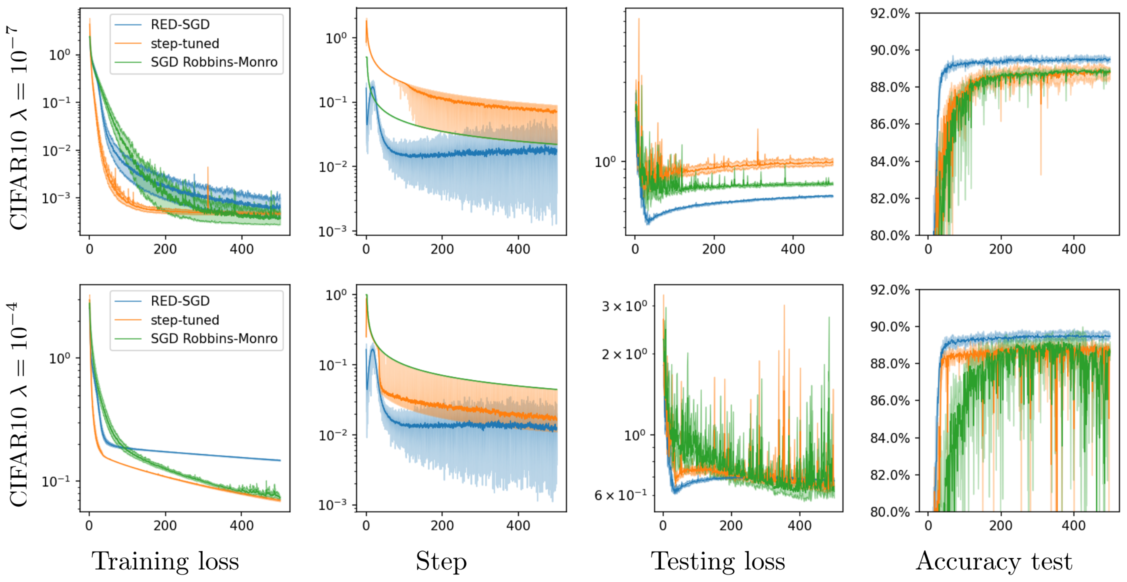


Figure 9: Comparison with step-tuned [8] method on the CIFAR10 classifier. Standard SGD (blue), RED (orange) and step-tuned (green) are given. RED is competitive with step-tuned on the accuracy but not on the training loss of the CIFAR10 classifier for which step-tuned is optimized.