Stochastic Average Gradient : A Simple Empirical Investigation

Pascal Junior Tikeng Notsawo

Stochastic Average Gradient : A Simple Empirical Investigation

Pascal Junior Tikeng Notsawo
pascal.junior.tikeng.notsawo@umontreal.ca

DIRO, Université de Montréal, Montréal, Quebec, Canada

Abstract

Despite the recent growth of theoretical studies and empirical successes of neural networks, gradient backpropagation is still the most widely used algorithm for training such networks. On the one hand, we have deterministic or full gradient (FG) approaches that have a cost proportional to the amount of training data used but have a linear convergence rate, and on the other hand, stochastic gradient (SG) methods that have a cost independent of the size of the dataset, but have a less optimal convergence rate than the determinist approaches. To combine the cost of the stochastic approach with the convergence rate of the deterministic approach, a stochastic average gradient (SAG) has been proposed. SAG is a method for optimizing the sum of a finite number of smooth convex functions. Like SG methods, the SAG method's iteration cost is independent of the number of terms in the sum. In this work, we propose to compare SAG to some standard optimizers used in machine learning. SAG converges faster than other optimizers on simple toy problems and performs better than many other optimizers on simple machine learning problems. We also propose a combination of SAG with the momentum algorithm and Adam. These combinations allow empirically higher speed and obtain better performance than the other methods, especially when the landscape of the function to optimize presents obstacles or is ill-conditioned [1].

1 Introduction

In many domains, several problems can be reduced to the minimization of the sum of a finite number of functions

$$g = \frac{1}{n}\sum_{i=1}^{n} f_i$$

That is

$$\underset{x\in\Omega\subset\mathbb{R}^p}{\text{minimize}} \quad g(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x) \tag{1}$$

Gradient descent (Cauchy, 1847; Bottou, 1998; Nemirovski et al., 2009; Duchi et al., 2011b; Kingma and Ba, 2014) optimize such functions with a rule of the form :

$$x^{k+1} = x^k - \alpha_k D^k$$

where α_k is the step size at iteration k; and D^k a function of the past gradients $G_1, \ldots, G_k$ of g at $x^1, \ldots, x^k$, respectively, or of the estimators of these gradients; such that $\mathbb{E}[D^k|x^{k-1}] = \nabla g(x^k)$.

[1] This work is reproducible at `https://github.com/Tikquuss/sag_torch`

More specifically, $G_k = \nabla g(x^k)$ is the gradient of g at x^k, the parameter update at time k given the optimization algorithm of choice, and $\{\alpha_k, k \geq 0\}$ is a predefined deterministic sequence of positive real numbers such that $\sum_{k=1}^{\infty} \alpha_k = \infty$ and $\sum_{k=1}^{\infty} \alpha_k^2 < \infty$. The first of these two conditions is to make sure that the total displacement $\sum_{k=1}^{\infty} \alpha_k \nabla g(x^k)$ can be unbounded, so the optimal solution can be reached even if we start far away from it. The second condition (the finite sum of squares) is to decrease fast enough for the algorithm to converge. For convex functions, gradient descent converges to a global minimum (if one exists).

Problem 1 is very common in deep learning, where the goal is to minimize the regularized cost function

$$\mathcal{J}(\theta) = \mathbb{E}_{s\sim F}[\ell(s,\theta)] + \lambda r(\theta) = \int \ell(s,\theta) dF(s) + \lambda r(\theta)$$

where the function $\ell(s,\theta)$ measures how well the neural network with parameters θ predicts the label of a data sample s, F is the cumulative distribution function of the data distribution, $r(\theta)$ is the regularizer (e.g. ℓ_2-regularization $\frac{1}{2}\|\theta\|^2$), and $\lambda \in \mathbb{R}_+$ the regularization strength. In practice, F is generally unknown, and the empirical distribution of a given dataset $\mathcal{D}$ is used. The regularized empirical risk obtained can be written as a sum of $|\mathcal{D}|$ functions

$$\mathcal{J}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{s\in\mathcal{D}} \big[\ell(s,\theta) + \lambda r(\theta)\big]$$

This is the case, for example, of the least squares regression, with

$$\mathcal{D} = \{(x_i, y_i) \in \mathbb{R}^p \times \mathbb{R}\}_{i=1}^n \text{ and } \ell((x,y),\theta) = \|x^T\theta - y\|_2^2$$

or the logistic regression where ℓ is the negative log-likelihoods [2] :

$$\mathcal{D} = \{(x_i, y_i) \in \mathbb{R}^p \times \{-1,1\}\}_{i=1}^n \text{ and } \ell((x,y),\theta) = \log(1+\exp(-yx^T\theta))$$

One of the challenges that gradient-based methods face in practice is the ill-conditioned surfaces, when the hessian of the function to optimize has some large positive eigenvalues (i.e. high-curvature directions) and some eigenvalues close to 0 (i.e. low-curvature directions). In this case, vanilla gradient descent bounces back and forth in high curvature directions and slowly progresses in low curvature directions. In addition to these ill-conditioned surfaces, there are obstacles such as saddle points and critical surfaces (cliffs, valleys, plateaus, ravines, and other flat regions), extremely sharp or flat minima.

The aim of this work is to empirically investigate the performance of stochastic average gradient (SAG) (Schmidt et al., 2013) on this type of problem. We limit ourselves for the first time on simple toys finite data problems where each f_i is smooth and convex, although in modern applications, n, the number of data points (or training examples) can be extremely large (e.g. datasets used to train large-scale deep learning models like GPT-3 (Brown et al., 2020)), while there is often a large amount of redundancy between examples. In addition to this basic setting, we will also be interested in toys cases where the sum g is strongly convex, with the use of a strongly-convex regularizer such as the squared ℓ_2-norm, resulting in problems of the form :

$$\underset{x\in\mathbb{R}^p}{\text{minimize}} \quad \frac{\lambda}{2}\|x\|^2 + \frac{1}{n}\sum_{i=1}^n f_i(x) = \frac{1}{n}\sum_{i=1}^n \Big[\frac{\lambda}{2}\|x\|^2 + f_i(x)\Big] \tag{2}$$

The resulting function g will be strongly convex, provided that the individual functions f_i are convex.

We then extend our investigations to slightly more complex problems where we optimize deep neural networks on toys dataset. Many deep models are guaranteed to have an extremely large number of local minima. It has been proven that this is not necessarily a problem. Most local minima are of good quality (almost equivalent in cost to the global minimum) (Dauphin et al., 2014). The biggest obstacle to the optimization of g in deep learning remains the presence of saddle points. In low

[2]The decision boundary is $x^T\theta = 0$, i.e. we want $x^T\theta > 0$ for $y = 1$ and $x^T\theta < 0$ for $y = -1$, thas is $yx^T\theta > 0 \iff \text{sigmoid}(yx^T\theta) = 1/(1+\exp(-yx^T\theta)) > 1/2$. To maximize $\text{sigmoid}(yx^T\theta) \in [0,1]$, we minimize $-\log\big(\text{sigmoid}(yx^T\theta)\big) \in \mathbb{R}^+$, which gives our loss function.

dimensions (small p), local minima are more common, while in high dimensions, local minima are rare and saddle points more common. Most of the training time is spent on traversing flat valleys of the Hessian matrix or circumnavigating tall mountains via an indirect arcing path, and the trajectory of traversing such flat valleys and circumventing such mountains may be long and result in excessive training time (Srihari, 2020).

The rest of the paper is organized as follows. We define some terms used in our work in section 2, then we present SAG in section 3, the related works in section 4, the convergence analysis and the implementation details in sections 5 and 6 respectively. We finally present the experiments settings and the results in section 7, then summarise and conclude our work in section 8.

2 Definitions

We assume $g : \mathbb{R}^p \to \mathbb{R}$ unless otherwise noted. The function g is convex if for all $x, y \in \mathrm{domain}(g)$ and all $t \in [0, 1]$

$$g(tx + (1-t)y) \leq tg(x) + (1-t)g(y)$$

or equivalently if for all $x, y \in \mathrm{domain}(g)$,

$$g(x) \geq g(y) + \nabla g(y)^T (x - y)$$

if g is differentiable. If the inequality holds strictly (i.e. $<$ rather than $\leq$) for all $t \in (0, 1)$ and $x \neq y$, then we say that g is strictly convex, so strict convexity implies convexity. Geometrically, convexity means that the line segment between two points on the graph of g lies on or above the graph itself. If g is convex, then any local minimum of g in any convex set $X \subset \mathrm{domain}(g)$ is also a global minimum. Strict convexity means that the line segment lies strictly above the graph of g, except at the segment endpoints. If g is strictly convex, then at most, one local minimum of g in X exists. Consequently, if it exists, it is the unique global minimum of g in X [3].

For $\mu > 0$, the function g is μ-strongly convex if the function

$$x \mapsto g(x) - \frac{\mu}{2}\|x\|^2$$

is convex, or equivalently if for all $x, y \in \mathrm{domain}(g)$,

$$g(x) \geq g(y) + \nabla g(y)^T (x - y) + \frac{\mu}{2}\|x - y\|^2$$

if g is differentiable. Strong convexity doesn't necessarily require the function to be differentiable, and the gradient is replaced by the sub-gradient when the function is non-smooth. Intuitively speaking, strong convexity means a quadratic lower bound exists on the growth of the function. This directly implies that a strong convex function is strictly convex since the quadratic lower bound growth is, of course, strictly greater than the linear growth [4].

Let $G(x) = \nabla g(x) \in \mathbb{R}^p$ and $\mathcal{H}(x) = \nabla^2 g(x) \in \mathbb{R}^{p \times p}$ be respectively the gradient and the local hessian matrix of g at x, assuming that g is twice-differentiable. If $G(x) = 0$, then x is a critical/stationary point of g. In this case, the determinant $d(x)$ of $\mathcal{H}(x)$ is equal to the Gaussian curvature of the surface of g considered as a manifold. The eigenvalues of $\mathcal{H}(x)$ are the principal curvatures of the g at x, and the eigenvectors are the principal directions of curvature. If $d(x) > 0$, x is a local maximum of g if $\mathcal{H}(x)$ is negative definite (all its eigenvalues are negative), and a local minimum of g if $\mathcal{H}(x)$ is a positive definite (all its eigenvalues are positive). Some local optimums can be very flat (i.e. there is a large enough neighbourhood of x that contains only local optima) or sharp (the loss function near x has a high condition number, i.e. very small perturbation of x can cause large variation in g). If $d(x) < 0$ (some eigenvalues are positive and others are negative), x is a saddle point of g. If $d(x) = 0$ (there is at least one zero eigenvalue, i.e. $\mathcal{H}(x)$ is undefined), we can't conclude, and the point x could be any of a minimum, maximum or saddle point. If the hessian matrix of g is positive semi-definite at any point of $\mathrm{domain}(g)$, then g is convex and the point x such that $G(x) = 0$ is its global minimum. If it is instead negative semi-definite at any point of $\mathrm{domain}(g)$, then g is concave and the point x such that $G(x) = 0$ is its global maximum.

[3] `https://ai.stanford.edu/~gwthomas/notes/convexity.pdf`

[4] `https://xingyuzhou.org/blog/notes/strong-convexity`

3 Motivation

Gradient descent (Bottou, 1998) is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. FG method (Cauchy, 1847) uses iterations of the form

$$x^{k+1} = x^k - \alpha_k \nabla g(x^k) = x^k - \frac{\alpha_k}{n} \sum_{i=1}^{n} \nabla f_i(x^k)$$

FG is generally called batch gradient descent in deep learning since it calculates the error for each example in the training dataset but only updates the model after all training examples have been evaluated. Therefore, its cost per iteration is $\mathcal{O}(n)$.

Assuming that a minimizer x^* exists and g is convex, then under standard assumptions, the sub-optimality achieved on iteration k of the FG method with a constant step size is given by a sublinear convergence rate (Nesterov, 2004; Schmidt et al., 2013)

$$g(x^k) - g(x^*) = \mathcal{O}(1/k)$$

When g is strongly convex, the error also satisfies a linear convergence rate (also known as a geometric or exponential rate because a fixed fraction cuts the error on each iteration) (Nesterov, 2004; Schmidt et al., 2013)

$$g(x^k) - g(x^*) = \mathcal{O}(\rho^k) \text{ for some } \rho < 1$$

This ρ depends on the condition number of g, i.e. on how sensitive the output of g is on its input [5]. One drawback of the FG approach is that it requires computing all the gradients at each iteration, which can be tedious when n is very large.

The basic SG method for optimizing 1 uses iterations of the form

$$x^{k+1} = x^k - \alpha_k \nabla f_{i_k}(x^k)$$

where at each iteration an index i_k is sampled uniformly from the set $\{1, \ldots, n\}$. The randomly chosen gradient $\nabla f_{i_k}(x^k)$ yields an unbiased estimate of the true gradient $\nabla g(x^k)$:

$$\mathbb{E}_{i_k \sim \mathcal{U}(\{1,\ldots,n\})}[\nabla f_{i_k}(x^k)] = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x^k) = \nabla g(x^k)$$

Under standard assumptions and for a suitably chosen decreasing step-size sequence $\{\alpha_k, k \geq 0\}$ (Nemirovski et al., 2009; Schmidt et al., 2013), the SG iterations have an expected sub-optimality for convex objectives of

$$\mathbb{E}[g(x^k)] - g(x^*) = \mathcal{O}(1/\sqrt{k})$$

and an expected sub-optimality for strongly-convex objectives of

$$\mathbb{E}[g(x^k)] - g(x^*) = \mathcal{O}(1/k)$$

These sublinear rates are slower than the corresponding rates for FG. Under certain assumptions, these convergence rates are optimal in a model of computation where the algorithm only accesses the function through unbiased measurements of its objective and gradient. Thus, we should not expect to be able to obtain the convergence rates of the FG method if the algorithm only relies on unbiased gradient measurements. Can we have one gradient per iteration and achieve the same rate as FG?

Mini-batch gradient descent is a variation of the SG algorithm that splits the training dataset into small batches used to calculate model error and update model coefficients. In other words, we select a batch $\mathcal{B} \subset \{1, \ldots, n\}$ randomly at each iteration and do the update as follows:

$$x^{k+1} = x^k - \frac{\alpha_k}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla f_i(x^k)$$

[5] L/μ (change in output = condition number $\times$ change in input)

But this allows to make a trade-off between the cost per iteration and the convergence rate: either we choose $\mathcal{B}$ is too big, and we get a better rate and a big cost of $\mathcal{O}(|\mathcal{B}|)$ per iteration, or we choose $\mathcal{B}$ so that $|\mathcal{B}|$ is too small, and we get a lower rate and a cost in $\mathcal{O}(1)$ per iteration.

The SAG iterations take the form

$$x^{k+1} = x^k - \frac{\alpha_k}{n} \sum_{i=1}^{n} y_i^k$$

where at each iteration a random index i_k is selected (not necessarily uniformly from $\{1, \dots, n\}$ as we will see below) and we set

$$y_i^k = \begin{cases} \nabla f_i(x^k) & \text{if } i = i_k \\ y_i^{k-1} & \text{otherwise.} \end{cases}$$

Like the FG method, the step incorporates a gradient with respect to each function. But, like the SG method, each iteration only computes the gradient with respect to a single example and the cost of the iterations is independent of n : we take a step in the direction of the average of y_i^k.

With the mini-batch version of SAG, the update becomes

$$y_i^k = \begin{cases} \nabla f_i(x^k) & \text{if } i \in \mathcal{B} \\ y_i^{k-1} & \text{otherwise.} \end{cases}$$

4 Related works

In the following D^k is a function of the past gradients $G_1, \dots, G_k$ of g at $x^1, \dots, x^k$, respectively, or of the estimators of these gradients. In the papers introducing these algorithms, $D^k = G_k$ in general, i.e. D^k is deterministic. But their SG version can be developed with $D^k = \nabla f_{i_k}(x^k)$ for a randomly sampled $i_k \in \{1, \dots, n\}$, or their mini-batch version with $D^k = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla f_i(x^k)$ for a random sample $\mathcal{B} \subset \{1, \dots, n\}$, or their SAG version with an appropriate choice of past gradients to use and how to use them.

SG methods that incorporate a each iteration k a momentum term $m^k = x^k - x^{k-1} = -\alpha_{k-1} D^{k-1}$ use iterations of the form (Polyak, 1964; Sutton, 1986)

$$x^{k+1} = x^k - \alpha_k D^k + \beta_k m^k$$

It is common to set all $\beta_k = \beta_1$ for some constant $\beta_1 \in [0, 1)$, and in this case, we can rewrite the SG with momentum (Tseng, 1998) method as

$$x^{k+1} = x^k - \sum_{j=0}^{k} \alpha_j \beta_1^{k-j} D^j$$

The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction. Formally, the momentum algorithm introduces a variable v that plays the role of velocity: the direction and speed at which the parameters move through parameter space. The hyperparameter β_1 determines how quickly the contributions of previous gradients exponentially decay. The above update rule can be rewritten in terms of the velocity as ($v^0 = 0$):

$$v^{k+1} = \beta_1 v^k - \alpha_k D^k$$
$$x^{k+1} = x^k + v^{k+1}$$

Since we have with this

$$v^{k+1} = -\sum_{j=0}^{k} \alpha_j \beta_1^{k-j} D^j$$

The SAG version of momentum becomes

$$x^{k+1} = x^k + \frac{\alpha_k}{n} \sum_{i=1}^{n} y_i^k$$

where at each iteration, a random index i_k is selected, and we set

$$y_i^k = \begin{cases} v_i^{k+1} = \beta_1 v_i^k - \alpha_k D_i^k & \text{if } i = i_k, \text{ with } D^k = \nabla f_{i_k}(x^k) \\ y_i^{k-1} & \text{otherwise.} \end{cases}$$

Nesterov accelerated gradient or Nesterov momentum (Nesterov, 1983; Sutskever et al., 2013) is a variant of the momentum algorithm that use an interim update $\tilde{x}^k = x^k + \beta_1 v^k$ to compute de gradient D^k at each iteration. That is :

$$\begin{aligned} \tilde{x}^k &= x^k + \beta_1 v^k \\ \tilde{D}^k &= \nabla g(\tilde{x}^k) \\ v^{k+1} &= \beta_1 v^k - \alpha_k \tilde{D}_k \\ x^{k+1} &= x^k + v^{k+1} \end{aligned}$$

The AdaGrad algorithm (Duchi et al., 2011a), individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared values. The update rule of AdaGrad is given by ($r^0 = 0$, r^k accumulates squared gradient, division and square root are applied element-wise, ϵ is a very small number used to avoid divisions by 0) :

$$\begin{aligned} r^{k+1} &= r^k + D^k \odot D^k \\ x^{k+1} &= x^k - \frac{\alpha_k}{\sqrt{r^{k+1}} + \epsilon} \odot D^k \end{aligned}$$

The RMSProp algorithm (Hinton, 2012) modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average. RMSProp uses an exponentially decaying average to discard history from the extreme past to converge rapidly after finding a convex bowl as if it were an instance of the AdaGrad algorithm initialized within that bowl. Compared to AdaGrad, using the moving average introduces a new hyperparameter, $\beta_2 \in (0, 1]$, that controls the length scale of the moving average. The step of squared gradient accumulation is modified as follows:

$$r^{k+1} = \beta_2 r^k + (1 - \beta_2) D^k \odot D^k$$

Adadelta (Zeiler, 2012) is an extension of Adagrad and RMSProp that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size ($u^0 = 0$).

$$\begin{aligned} r^{k+1} &= \beta_2 r^k + (1 - \beta_2) D_k \odot D_k \\ \Delta^{k+1} &= \frac{\sqrt{u^k + \epsilon}}{\sqrt{r^{k+1} + \epsilon}} \\ u^{k+1} &= \beta_2 u^k + (1 - \beta_2) \Delta^{k+1} \\ x^{k+1} &= x^k - \alpha_k \Delta^{k+1} \end{aligned}$$

Adam (Kingma and Ba, 2017) is a combination of RMSProp and momentum. First, in Adam, momentum is incorporated directly as an estimate of the gradient's first-order moment (with exponential weighting). Second, Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin.

$$\begin{aligned} \tilde{x}^k &= x^k + \beta_1 v^k \\ \tilde{D}^k &= \nabla g(\tilde{x}^k) \\ r^{k+1} &= \beta_2 r^k + (1 - \beta_2) \tilde{D}^k \odot \tilde{D}^k \\ v^{k+1} &= \beta_1 v^k - \frac{\alpha_k}{\sqrt{r^{k+1}} + \epsilon} \odot \tilde{D}^k \\ x^{k+1} &= x^k + v^{k+1} \end{aligned}$$

The most common Adam iteration update is written in term of momentum as

$$
\begin{aligned}
m^k &= \beta_1 m^{k-1} + (1-\beta_1) D^k \\
r^k &= \beta_2 r^{k-1} + (1-\beta_2) D^k \odot D^k \\
x^k &= x^{k-1} - \frac{\alpha_k}{\sqrt{r^k} + \epsilon} \odot m^k
\end{aligned}
$$

Adamax (Kingma and Ba, 2014, 2017) is a variant of Adam based on infinity norm.

$$
\begin{aligned}
m^k &= \beta_1 m^{k-1} + (1-\beta_1) D^k \\
u^k &= \max(\beta_2 u^{k-1}, |D^k| + \epsilon) \\
x^k &= x^{k-1} - \frac{\alpha_k}{(1-\beta_1^k) u^k} \odot m^k
\end{aligned}
$$

AMSGrad (Reddi et al., 2018) is a version of Adam that keeps a running maximum of the squared gradients instead of an exponential moving average.

$$
\begin{aligned}
m^k &= \beta_1 m^{k-1} + (1-\beta_1) D^k \\
\tilde{m}^k &= \max(\tilde{m}^{k-1}, m^k) \\
r^k &= \beta_2 r^{k-1} + (1-\beta_2) D^k \odot D^k \\
x^k &= x^{k-1} - \frac{\alpha_k}{\sqrt{r^k} + \epsilon} \odot \tilde{m}^k
\end{aligned}
$$

All Adaptive methods can be summarized as follows (Défossez et al., 2020). As hyper-parameters, we have $0 \leq \beta_1 < \beta_2 \leq 1$, and a non negative sequence $(\alpha_k)_{k \in \mathbb{N}^*}$. We define three vectors $m_k, r_k, x_k \in \mathbb{R}^p$ iteratively. Given $x^0 \in \mathbb{R}^p$ as our starting point, $m^0 = 0$, and $r^0 = 0$, we define for all iterations $k \in \mathbb{N}^*$

$$
\begin{aligned}
m_i^k &= \beta_1 m_i^{k-1} + D_i^k \\
r_i^k &= \beta_2 r_i^{k-1} + \left(D_i^k\right)^2 \\
x_i^k &= x_i^{k-1} - \alpha_k \frac{m_i^k}{\sqrt{r_i^k} + \epsilon}
\end{aligned}
$$

.

The parameter β_1 is a heavy-ball style momentum parameter. The parameter β_2 controls the decay rate of the per-coordinate exponential moving average of the squared gradients. Taking $\beta_1 = 0$, $\beta_2 = 1$ and $\alpha_k = \alpha$ gives Adagrad (Duchi et al., 2011b). The original Adam algorithm (Kingma and Ba, 2014) uses a weighed average, rather than a weighted sum :

$$
\tilde{m}_i^k = (1-\beta_1) \sum_{j=1}^{k} \beta_1^{k-j} D_i^{j-1} = (1-\beta_1) m_i^k
$$

We can achieve the same definition by taking $\alpha_{adam} = \alpha \cdot \frac{1-\beta_1}{\sqrt{1-\beta_2}}$, since

$$
\frac{\tilde{m}_i^k}{\sqrt{\tilde{r}_i^k}} = \frac{1-\beta_1}{\sqrt{1-\beta_2}} \frac{m_i^k}{\sqrt{r_i^k}}
$$

with

$$
\tilde{r}_i^k = (1-\beta_2) r_i^k \text{ and } \tilde{m}_i^k = (1-\beta_1) m_i^k
$$

The original Adam algorithm further includes two corrective terms to account for the fact that m^k and r^k are biased towards 0 for the first few iterations. Those corrective terms are equivalent to taking a step-size α_k of the form

$$\alpha_{k,adam} = \alpha \cdot \frac{1-\beta_1}{\sqrt{1-\beta_2}} \cdot \overbrace{\frac{1}{\sqrt{1-\beta_1^k}}}^{\text{corrective term for } m^k} \cdot \underbrace{\sqrt{1-\beta_2^k}}_{\text{corrective term for } r^k}$$

Early work on adaptive methods (e.g. (McMahan and Streeter, 2010)) showed that Adagrad achieves an optimal rate of convergence of $\mathcal{O}(1/\sqrt{k})$ for convex optimization. Ward et al. (2020) proved that Adagrad converges to a critical point for non convex objectives with a rate $\mathcal{O}(ln(k)/\sqrt{k})$ when using a scalar adaptive step-size. Défossez et al. (2020) show a rate of $\mathcal{O}(p\ ln(k)/\sqrt{k})$ for Adam, and show that in expectation, the squared norm of the objective gradient averaged over the trajectory has an upper-bound which is explicit in the constants of the problem, parameters of the optimizer, the dimension p, and the total number of iterations k.

5 SAG convergence rate

We assume that each function f_i in (1) is convex and differentiable (this makes g also convex and differentiable), and that each gradient ∇f_i is Lipschitz-continuous with constant L_i, meaning that for all x and y in $\mathbb{R}^p$ and each i we have

$$\|\nabla f_i(x) - \nabla f_i(y)\| \leq L_i \|x - y\| \tag{3}$$

This makes ∇g also Lipschitz-continuous with any constant $L \geq \frac{1}{n}\sum_{i=1}^n L_i$, like $\max_i L_i$. Also, each gradient ∇f_i is Lipschitz-continuous with constant $L \geq \max_i L_i$. This is a fairly weak assumption on the f_i functions, and in cases where the f_i are twice-differentiable it is equivalent to saying that the eigenvalues of the hessians of each f_i are bounded above by L. We will also assume the existence of at least one minimizer x^* that achieves the optimal function value.

In addition to the above basic convex case, we will also consider the case where the average function $g = \frac{1}{n}\sum_{i=1}^n f_i$ is strongly-convex with constant $\mu > 0$, meaning that the function $x \mapsto g(x) - \frac{\mu}{2}\|x\|^2$ is convex. For twice-differentiable g, this is equivalent to requiring that the eigenvalues of the hessian of g are bounded below by μ. This is a stronger assumption that is often not satisfied in practical applications. Nevertheless, in many applications we are free to choose a regularizer of the parameters, and thus we can add an ℓ_2-regularization term as in (2) to transform any convex problem into a strongly-convex problem (in this case we have $\mu \geq \lambda$). Note that strong-convexity implies the existence of a unique x^* that achieves the optimal function value.

Let $\bar{x}_k = \frac{1}{k}\sum_{i=0}^{k-1} x^i$ be the average iterate and $\sigma^2 = \frac{1}{n}\sum_{i=1}^n \|\nabla f_i(x^*)\|$ the variance of the gradient norms at the optimum x^*.The convergence results consider two different initializations for the y_i^0 variables:

- setting $y_i^0 = 0$ for all i
- or setting them to the centered gradient at the initial point x^0 : $y_i^0 = \nabla f_i(x^0) - \nabla g(x^0)$

The convergence results are expressed in terms of expectations $\mathbb{E}$ with respect to the internal randomization of the algorithm (the selection of the random variables i_k), and not with respect to the data which is assumed to be deterministic and fixed. The L we use in the following is a Lipschitz-continuous constant common to all ∇f_i, as $\max_i L_i$.

Theorem 5.1. *With a constant step size of $\alpha = \frac{1}{16L}$, the SAG iterations satisfy for $k \geq 1$:*

$$\mathbb{E}[g(\bar{x}^k)] - g(x^*) \leq \frac{32n}{k} C_0 \in \mathcal{O}\left(\frac{1}{k}\right) \tag{4}$$

where if we initialize with $y_i^0 = 0$ for all i we have

$$C_0 = g(x^0) - g(x^*) + \frac{4L}{n}\|x^0 - x^*\|^2 + \frac{\sigma^2}{16L}$$

and if we initialize with $y_i^0 = \nabla f_i(x^0) - \nabla g(x^0)$ for all i we have

$$C_0 = \frac{3}{2}\big[g(x^0) - g(x^*)\big] + \frac{4L}{n}\|x^0 - x^*\|^2$$

Further, if g is μ-strongly convex we have

$$\mathbb{E}[g(x^k)] - g(x^*) \leq \left(1 - min\left\{\frac{\mu}{16L}, \frac{1}{8n}\right\}\right)^k C_0 \in \mathcal{O}\left(\left(1 - min\left\{\frac{\mu}{16L}, \frac{1}{8n}\right\}\right)^k\right)$$

The proof of this theorem is given in Schmidt et al. (2013) [Appendix B] and involves finding a Lyapunov function for a non-linear stochastic dynamical system defined on the y_i^k and x_k variables that converges to zero at the above rates, and showing that this function dominates the expected sub-optimality $\mathbb{E}[g(x^k)] - g(x^*)$. The equation (4) is stated for the average $\bar{x}^k$, with a trivial change to the proof technique, but it can be shown to also hold for any iterate x^k where $g(x^k)$ is lower than the average function value up to iteration k, $\frac{1}{k}\sum_{i=0}^{k-1} g(x^i)$. Thus, in addition to $\bar{x}^k$ the result also holds for the best iterate.

The bounds are valid for any L greater than or equal to the minimum L satisfying (3) for each i, implying an $\mathcal{O}(1/k)$ and linear convergence rate for any $\alpha \leq 1/16L$, but the bound becomes worse as L grows. Although initializing each y_i^0 with the centered gradient may have an additional cost and slightly worsens the dependency on the initial sub-optimality $(g(x^0) - g(x^*))$, it removes the dependency on the variance σ^2 of the gradients at the optimum.

While the theorem is stated in terms of the function values, in the μ-strongly-convex case we also obtain a convergence rate on the iterates because we have

$$\frac{\mu}{2}\|x^k - x^*\|^2 \leq g(x^k) - g(x^*)$$

The SAG iterations have a worse constant factor because of the dependence on n. An appropriate choice of x^0 can improve the dependence on n : we can set x^0 to the result of n iterations of an appropriate SG method. In this setting, the expectation of $g(x^0) - g(x^*)$ is $\mathcal{O}(1/\sqrt{n})$ in the convex setting, while both $g(x^0) - g(x^*)$ and $\|x^0 - x^*\|^2$ would be in $\mathcal{O}(1/n)$ in the strongly-convex setting.

If we use this initialization of x^0 and set $y_i^0 = \nabla f_i(x^0) - \nabla g(x^0)$, then in terms of n and k the SAG convergence rates take the form $\mathcal{O}(\sqrt{n}/k)$ and $\mathcal{O}(\rho^k/n)$ in the convex and strongly-convex settings, instead of the $\mathcal{O}(n/k)$ and $\mathcal{O}(\rho^k)$ rates implied by the theorem.

An interesting consequence of using a step-size of $\alpha = 1/16L$ is that it makes the method adaptive to the strong-convexity constant μ. For problems with a higher degree of local strong-convexity around the solution x^*, the algorithm will automatically take advantage of this and yield a faster local rate. This can even lead to a local linear convergence rate if the problem is strongly-convex near the optimum but not globally strongly-convex. This adaptivity to the problem difficulty is in contrast to SG methods whose sequence of step sizes typically depend on global constants and thus do not adapt to local strong-convexity. *We will test this on the Rosenbrock function in log scale, for which the SG method turns indefinitely around the global minimum and never reaches it.*

6 SAG implementation Details

Schmidt et al. (2013) discuss modifications that lead to better practical performance than this basic algorithm, including ways to reduce the storage cost, how to handle regularization, how to set the step size, using mini-batches, and using non-uniform sampling.

Algorithm 1: Basic SAG method for minimizing $\frac{1}{n}\sum_{i=1}^n f_i(x)$ with step size α

begin
  $d = 0$ /* d is use to track the quantity $\sum_{i=1}^n y_i$ */
  $y_i = 0$ for $i = 1, 2, \ldots, n$
  for $k = 0, 1, \ldots$ **do**
    Sample i from $\{1, 2, \ldots, n\}$
    $d = d - y_i + \nabla f_i(x)$
    $y_i = \nabla f_i(x)$
    $x = x - \frac{\alpha}{n} d$
  end
end

Re-weighting on early iterations The more logical normalization is to divide d by m, the number of data points that we have seen at least once (which converges to n once we have seen the entire data set), when $y_i^0 = 0$

$$x = x - \frac{\alpha}{m} d$$

Exact and efficient regularization

$$x = x - \alpha\left(\frac{d}{m} + \lambda x\right) = (1 - \alpha\lambda)x - \frac{\alpha}{m} d$$

Mini-batches for vectorized computation and reduced storage

$$x^{k+1} = x^k - \frac{\alpha_k}{n} \sum_{i=1}^{n} y_i^k \text{ with } y_i^k = \begin{cases} \nabla f_i(x^k) & \text{if } i \in \mathcal{B} \\ y_i^{k-1} & \text{otherwise.} \end{cases}$$

Structured gradients and just-in-time parameter updates For many problems the storage cost of $\mathcal{O}(np)$ for the y_i^k vectors is prohibitiven but we can often use the structure of the gradients ∇f_i to reduce this cost. For example, let consider a linearly-parameterized model of the form

$$\underset{x \in \Omega \subset \mathbb{R}^p}{\text{minimize}} \quad g(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(a_i^T x) \tag{5}$$

Since each a_i is constant, for these problems we only need to store the scalar $\nabla f_{i_k}(u_i^k)$ for $u_i^k = a_{i_k}^T x$ rather than the full gradient $a_i \nabla f_i(u_i^k)$. This reduces the storage cost from $\mathcal{O}(np)$ down to $\mathcal{O}(n)$. Examples of linearly-parameterized models include the least-squares regression [6], the logistic regression [7], feed forward neural networks, etc.

7 Experiments settings Results

We will use the following acronyms to designate our algorithms :

- sgd : vanilla SGD
- momentum : SGD with momentum (Polyak, 1964; Sutton, 1986; Tseng, 1998)
- nesterov : Nesterov Accelerated SGD (Nesterov, 1983; Sutskever et al., 2013)
- asgd : Averaged SGD proposed by Polyak and Juditsky (1992)
- rmsprop : RMSProp (Hinton, 2012)
- rmsprop_mom : RMSProp with momentum
- rprop : resilient backpropagation algorithm (Riedmiller and Braun, 1993)
- adadelta : Adadelta (Zeiler, 2012)
- adagrad : Adagrad (Duchi et al., 2011b)
- adam : Adam (Kingma and Ba, 2014, 2017)
- amsgrad : AMSGrad (Reddi et al., 2018)
- adamax : Adamax (Kingma and Ba, 2014)
- custom_adam : custom adam algorithm without amsgrad and that include the two corrective terms for m^k and r^k
- adam_inverse_sqrt : Adam that decays the learning rate based on the inverse square root of the update number. It also supports a warmup phase where the learning rate is linearly increase from some initial learning rate ($warmup\_init\_lr$) until the configured learning rate (lr). Thereafter, the learning rate is decay proportional to the number of updates, with a decay factor set to align with the configured learning rate.

[6] $\ell(s = (x, y), \theta) = h(x^T\theta)$ *with* $h(z) = (z - y)^2$
[7] $\ell(s = (x, y), \theta) = h(x^T\theta)$ *with* $h(z) = log(1 + exp(-yz))$

- During warmup:

$$lrs = linspace(start = warmup\_init\_lr, end = lr, steps = warmup\_updates)$$

$$lr = lrs[step]$$

- After warmup:

$$lr = \frac{decay\_factor}{\sqrt{update\_num}} \text{ where } decay\_factor = lr * sqrt(warmup\_updates)$$

- adam_cosine : Adam that assign learning rate based on a cyclical schedule that follows the cosine function (Loshchilov and Hutter, 2016). It also supports a warmup phase where the learning rate is linearly increase from some initial learning rate ($warmup\_init\_lr$) until the configured learning rate (lr). Thereafter, the learning rate is decay proportional to the number of updates, with a decay factor set to align with the configured learning rate.
  - During warmup:

$$lrs = linspace(start = warmup\_init\_lr, end = lr, steps = warmup\_updates)$$

$$lr = lrs[step]$$

  - After warmup:

$$lr = lr\_min + 0.5 * (lr\_max - lr\_min) * (1 + cos(t\_curr/t\_i))$$

    where t_curr is current percentage of updates within the current period range and t_i is the current period range, which is scaled by t_mul after every iteration.
- sag : SAG (Schmidt et al., 2013)
- sag_sgd : combinaition of SAG and momentum SGD with

$$y_i^k = \begin{cases} v^{k+1} = \beta_1 v_i^k - \alpha_k D_i^k & \text{if } i = i_k, \text{ with } D^k = \nabla f_{i_k}(x^k) \\ y_i^{k-1} & \text{otherwise.} \end{cases}$$

- sag_adam : combinaition of SAG and Adam with

$$y_i^k = \begin{cases} \frac{m_i^k}{\sqrt{r_i^k + \epsilon}} & \text{if } i = i_k \\ y_i^{k-1} & \text{otherwise.} \end{cases}$$

7.1 Test functions for optimization

7.1.1 Rosenbrock function

The vanilla rosenbrok function is given by $g_n(x) = \sum_{i=1}^{n/2} \left[100(x_{2i} - x_{2i-1}^2)^2 + (x_{2i-1} - 1)^2\right]$, with the gradient $\nabla_i g_n(x) = 200(x_i - x_{i-1}^2) \cdot \mathbb{1}_{i \in 2\mathbb{N}} - \left[400x_i(x_{i+1} - x_i^2) - 2(x_i - 1)\right] \cdot \mathbb{1}_{i \in 2\mathbb{N}-1}$, and $x^* \in \{(1, \dots, 1), (-1, 1, \dots, 1)\} \subset \{x, \nabla g_n(x) = 0\}$ [8]. A more involved variant is given by $g_n(x) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$, with the gradient $\nabla_i g_n(x) = 200(x_i - x_{i-1}^2) \cdot \mathbb{1}_{i>1} - \left[400x_i(x_{i+1} - x_i^2) - 2(x_i - 1)\right] \cdot \mathbb{1}_{i<n}$, and $x^* = \{1, \dots, 1)\} \subset \{x, \nabla g_n(x) = 0\}$ [9]. The number of stationary points of this function grows exponentially with dimensionality n, most of which are unstable saddle points (Kok and Sandrock, 2009).

We optimized the Rosenbrock function in a logarithmic scale (to create a ravine, figure 2). The function is unimodal, and the global minimum is very sharp and surrounded in the direction of the ravine by many local minima. At the beginning of optimization, we fall very quickly into the ravine because the surface is well-conditioned. Then, depending on the learning rate and the optimizer used (as well as the associated hyperparameters), we go down the ravine very slowly. Indeed, without momentum or velocity, we do not go directly down to the minimum since the gradient is almost

[8] When the coordinates range from 0 to $n-1$, $g_n(x) = \sum_{i=0}^{n/2-1} \left[100(x_{2i+1} - x_{2i}^2)^2 + (x_{2i} - 1)^2\right]$ and $\nabla_i g_n(x) = 200(x_i - x_{i-1}^2) \cdot \mathbb{1}_{i \in 2\mathbb{N}+1} - \left[400x_i(x_{i+1} - x_i^2) - 2(x_i - 1)\right] \cdot \mathbb{1}_{i \in 2\mathbb{N}}$.

[9] When the coordinates range from 0 to $n-1$, $g_n(x) = \sum_{i=0}^{n-2} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$ and $\nabla_i g_n(x) = 200(x_i - x_{i-1}^2) \cdot \mathbb{1}_{i>0} - \left[400x_i(x_{i+1} - x_i^2) - 2(x_i - 1)\right] \cdot \mathbb{1}_{i<n-1}$.

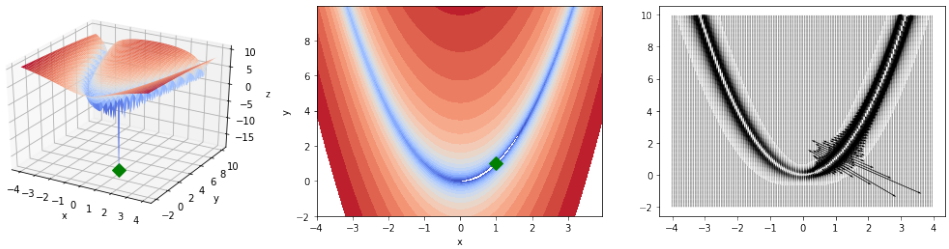


Figure 1: Left) Rosenbrock function in log scale ($n = 2$), Center) Contours, Right) Gradient field (note how this vector is pronounced in norm near the global minimum, which is important to understand why even near this global optimum many optimizers can felt to reach it)

zero along the ravine direction but very large in the perpendicular directions: we go from left to right (perpendicular to the ravine) while going down a little, but very slowly. Moreover, we turn there almost indefinitely once we are near the minimum. With adaptive gradient, we go down to the minimum very quickly because this direction problem is corrected (due to momentum, left-right ravine perpendicular directions cancel out): if the learning rate is too small, we will also go down very slowly (small gradient in the flat ravine direction). Unlike SGD, here, we always reach the minimum (and stay there). Also, for some learning rates and initializations, there is a double descent (Nakkiran et al., 2020) in error (euclidean distance between the global minimum and the current position at a given time) when landing in the ravine.

Adadelta and adagrad were very slow compared to sag. We can see in figure 2a a comparative progression of these three algorithms. After 100 000 iterations adadelta and adagrad were still going down to the valley, while SAG did it in less than 1000 iterations, which is 100 times faster than both. Adadelta manages to reach the minimum, which sag never finally does.

Nosterov is faster on the well-conditioned part of the surface and arrives faster in the neighbourhood of the target than sag, momentum and asgd (figure 2b). On the other hand, it stabilizes at a higher loss than these methods. Sag and asgd have almost the same trajectory. Momentum follows the same trajectory as these two methods from the beginning but stabilizes at a smaller loss. The combination sag_sgd (with momentum) speeds up the arrival in the neighbourhood of the minimum but stabilizes at the same level as momentum.

Rmsprop is slower than sag, but ends up with a smaller error than sag (figure 2c). Adding momentum to rmsprop (rmsprop_mom) improves its speed significantly. Rprop is also very fast and gives a smaller error than sag and sag_sgd.

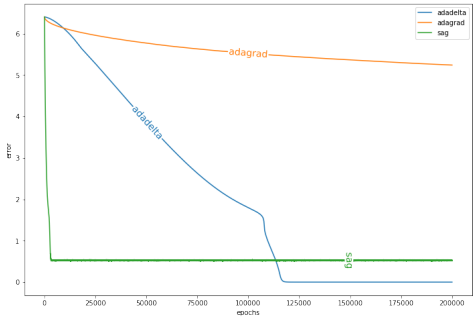
On the well-conditioned part of the surface, sag is faster than adam, adamax and amsgrad, but these methods reach the minimum (get zero final loss), which is not the case for sag (figure 2d). The sag_adam combination almost reaches the minimum, but is very chaotic and has periodic jumps that are similar to the slingshot mechanism (Thilak et al., 2022). amsgrad is much slower than adam and adamax.

custom_adam, adam_inverse_sqrt, adam_cosine also have the same periodic disruption phenomenon as sag_adam (figure 2e).

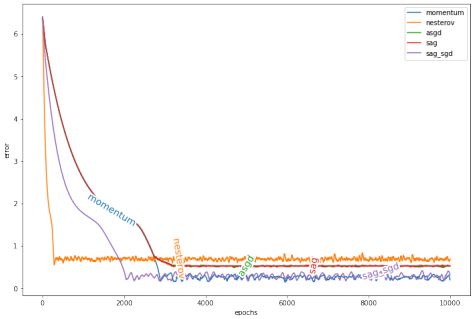
The methods that succeed in reaching the minimum are rmsprop, rprop, adadelta, adam, amsgrad, adamax, rmsprop_mom (figure 4). The methods that come close to it without reaching it are adam_inverse_sqrt, custom_adam, adam_cosine, sag_adam, momentum. The comparative convergence speeds are presented in figures 3 and 5, which is an approximation of the number of iterations performed before reaching stabilization.

7.1.2 Rastrigin function

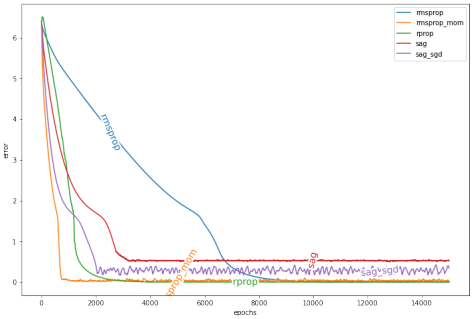
The rastrigin function is given by $g_n(x) = na + \sum_{i=1}^{n} \left[x_i^2 - a\cos(2\pi x_i)\right] = na + x^T x - a1_n^T \cos(2\pi x)$ with $a \in \mathbb{R}$. Its gradient is $\nabla g_n(x) = 2x + 2\pi a \sin(2\pi x)$, and $x^* = \{0, \dots, 0)\} \subset \{x, \nabla g_n(x) = 0\}$.



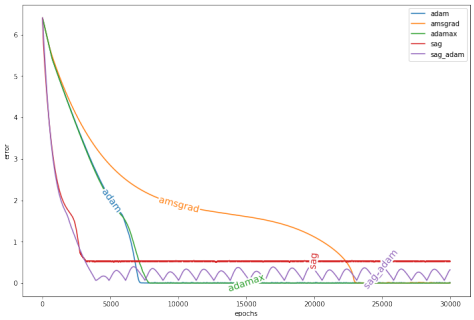
(a) adadelta and adagrad vs sag



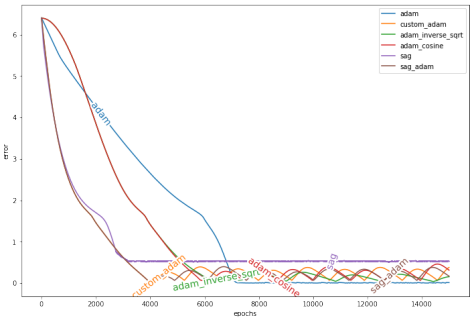
(b) momentum, nesterov and asgd vs sag



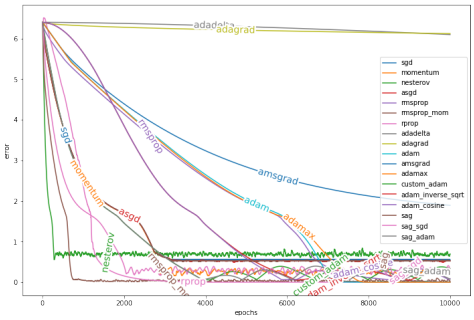
(c) rmsprop, rmsprop_mom and rprop vs sag



(d) adam, amsgrad and adamax vs sag and sag_adam



(e) adam, custom_adam, adam_inverse_sqrt, adam_cosine



(f) summary

Figure 2: Rosenbrock function

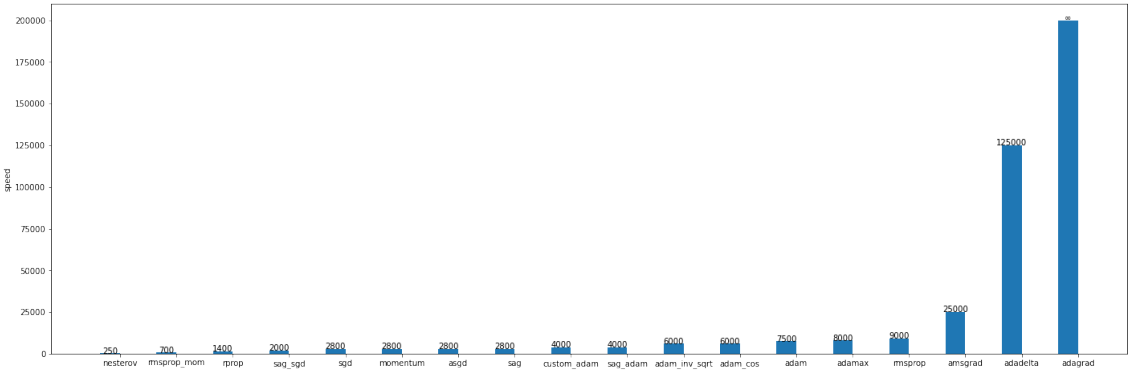


Figure 3: Comparative visualization of convergence speeds on the rosenbrock function

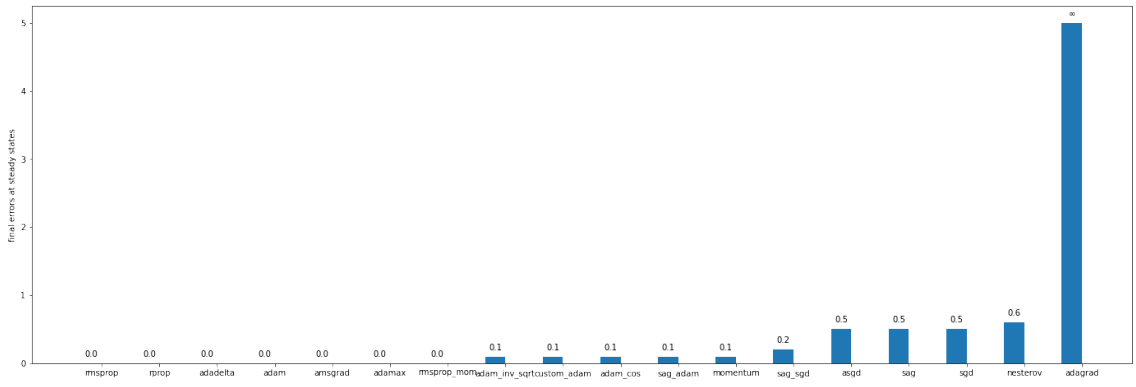


Figure 4: Final errors at steady states on the rosenbrock function

We optimized the Rastrigin function in a logarithmic scale (to create many local minimums and make the global minimum sharp, figure 7). The function is unimodal (in terms of global minimum), and the global minimum is very sharp and surrounded symmetrically by many local minima. At the beginning of optimization, we fall very quickly into the one local minimum. Then, depending on the learning rate and the optimizer used (and the associated hyperparameters), we can move successively from one minimum to another until we reach the global minimum.

Again, adadelta and adagrad are very slow compared to sag. We can see in figure 7a a comparative progression of these three algorithms. After 400 000 iterations adadelta and adagrad were still going down to the valley, while SAG did it in less than 1000 iterations, which is 400 times faster than both. adadelta manages to reach the minimum, which sag never finally does. Rprop is very bad here, it never leaves the first local minimum in which it falls. This is the method that obtains the largest error.

Nosterov is fast to reach the local minimum than sag, momentum and asgd (figure 7b), and stabilizes at the same error as these methods. sag and asgd have almost the same trajectory. Momentum follows slightly the same trajectory as these two methods from the beginning and stabilizes at the same error. The combination sag_sgd (with momentum) speeds up the arrival in the neighbourhood of the minimum and allows to obtain stabilization with a lower error. This means that it escapes more local minimums than the methods with which it is compared.

Rmsprop is slightly slower than sag and ends up with a bigger error than sag (figure 7c). Adding momentum to rmsprop (rmsprop_mom) improves its speed significantly, but we end up with the same error.

sag is faster than adam, adamax and amsgrad and gets a smaller error than them (figure 7d). The sag_adam combination is much faster with less error. It is also one of the only methods to approach the global minimum (i.e. to escape so many obstacles). Amsgrad is much slower than adam and adamax, but ends up with the same error as them.

Custom_adam is faster than adam_inverse_sqrt, adam_cosine, but ends up with the same error as them (figure 7e).

No method has reached the global minimum (figure 9). The methods that come close to it without reaching it are adam_inverse_sqrt, custom_adam, adam_cosine and sag_adam. The comparative convergence speeds are presented in figures 8 and 10, which approximate the number of iterations performed before reaching stabilization.

7.2 Toys machine learning problems

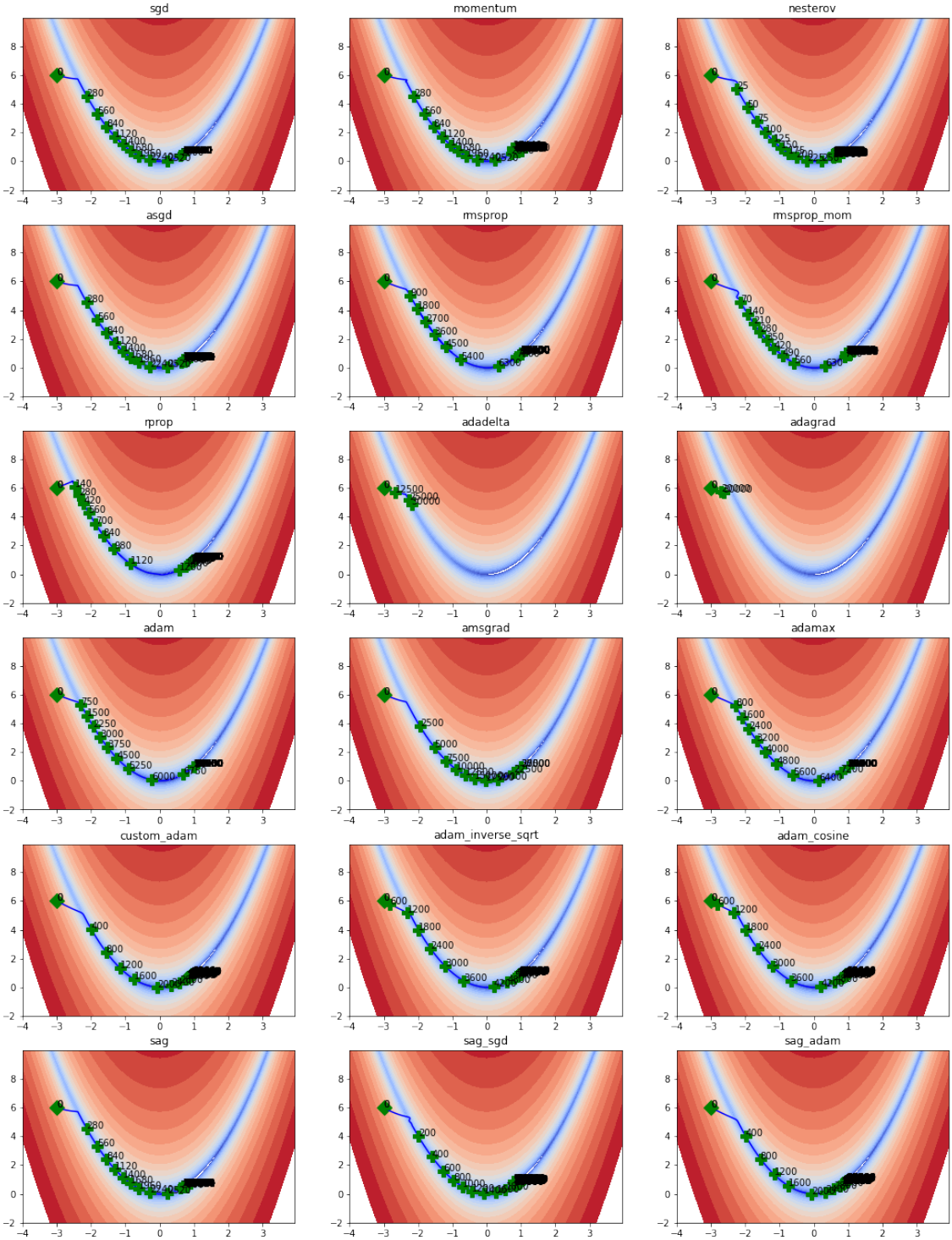


Figure 5: Comparative visualization of the progression of each algorithm on the rosenbrock function

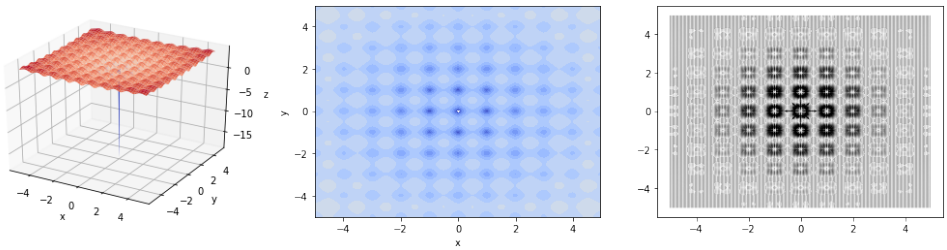
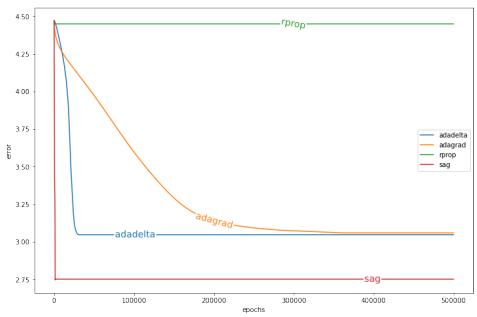
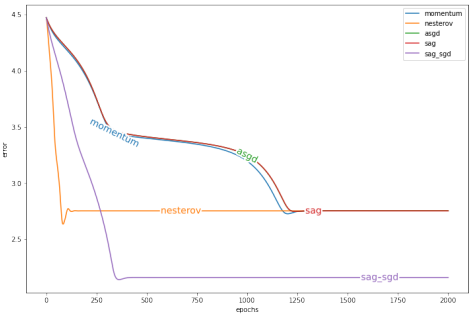


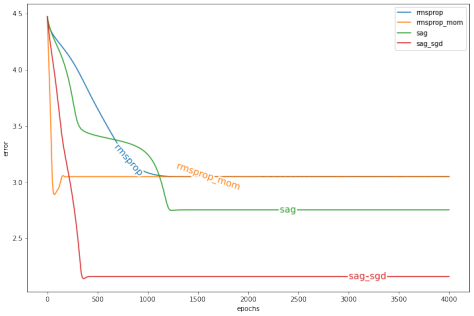
Figure 6: Left) Rastrigin function in log scale ($A = 10, n = 2$), Center) Contours, Right) Gradient field



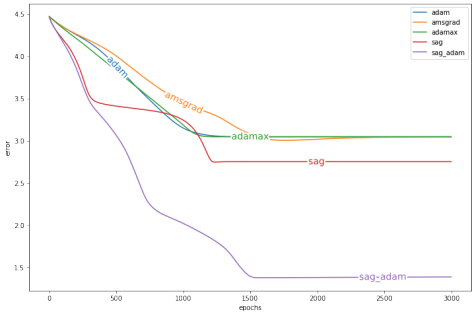
(a) adadelta, adagrad and rprop vs sag



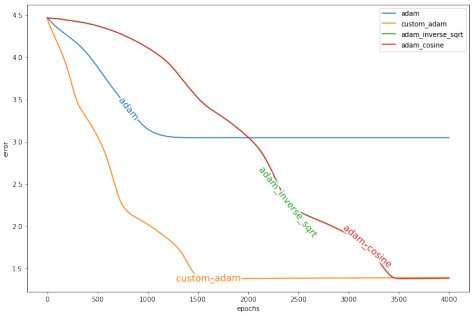
(b) momentum, nesterov and asgd vs sag



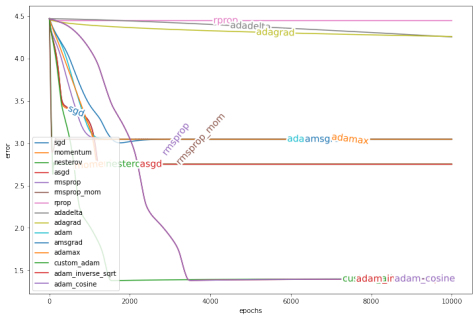
(c) rmsprop, rmsprop_mom vs sag



(d) adam, amsgrad and adamax vs sag and sag_adam



(e) adam, custom_adam, adam_inverse_sqrt, adam_cosine



(f) summary

Figure 7: Rastrigin function

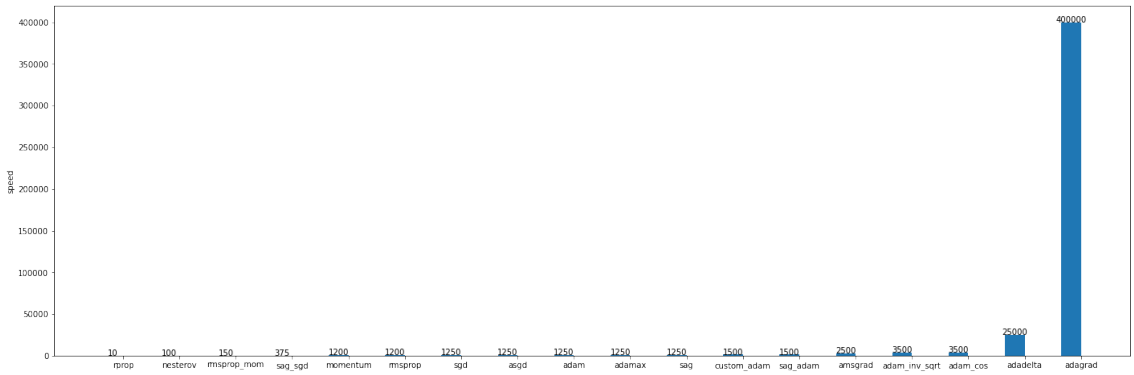


Figure 8: Comparative visualization of convergence speeds on the rastrigin function

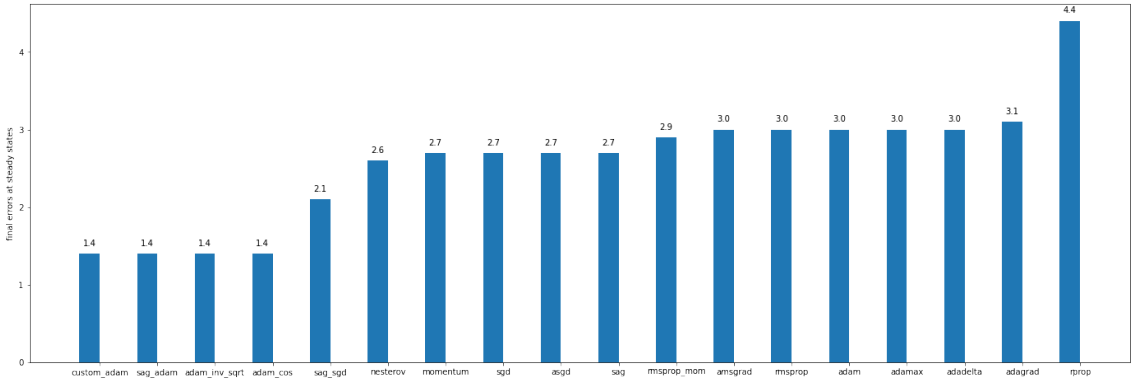


Figure 9: Final errors at steady states on the rastrigin function

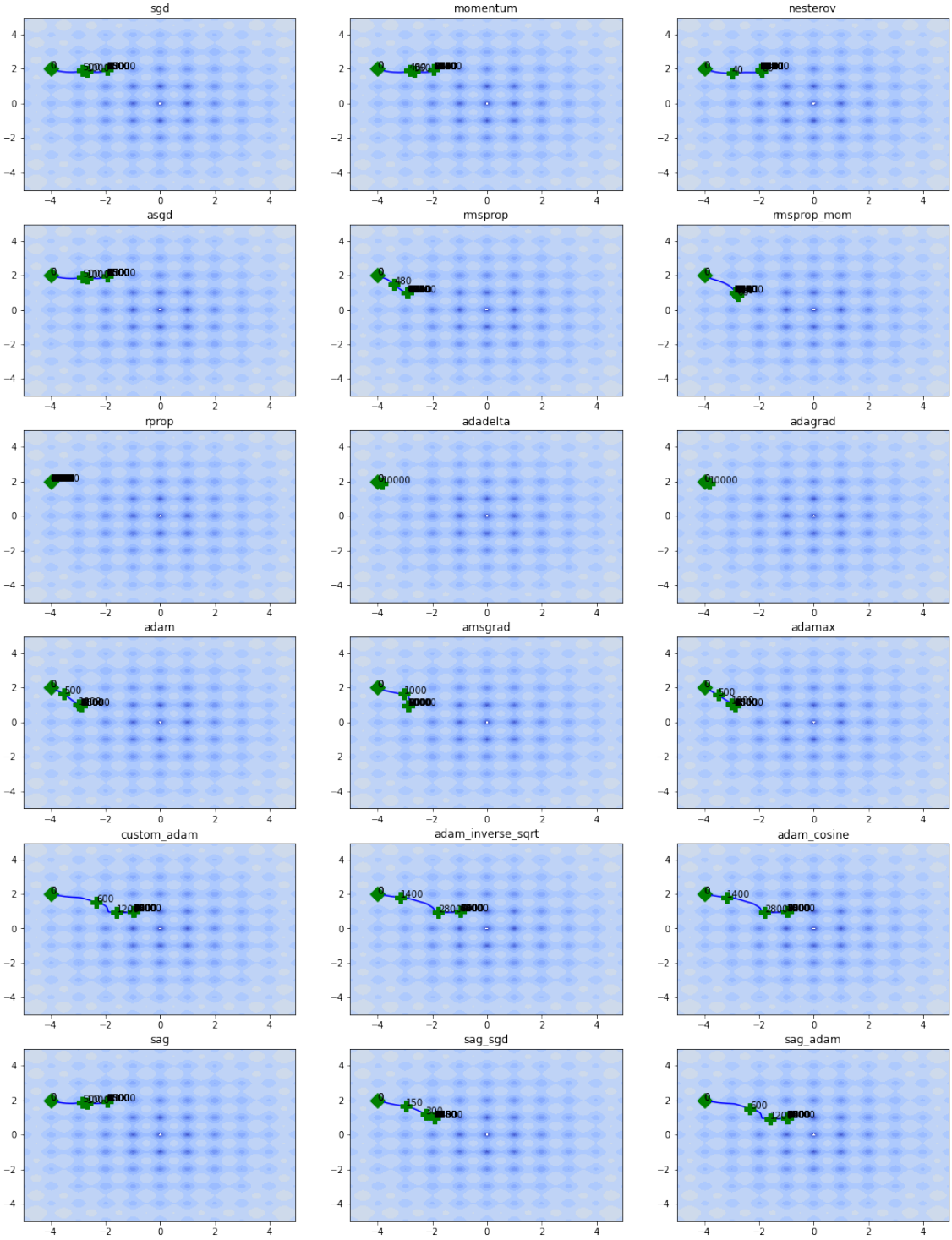


Figure 10: Comparative visualization of the progression of each algorithm on the rastrigin function

7.2.1 Scikit-learn dataset

We extracted the following datasets from scikit-learn (Pedregosa et al., 2011). The reader is invited to refer to the official scikit-learn website [10] for more information about these data (sources, ...).

- wine (classification): recognize the wine class given the features like the amount of alcohol, magnesium, phenol, colour intensity, etc.
- iris (classification): It contains sepal and petal lengths and widths for three classes of plants (Setosa, Versicolour, and Virginica)
- digits (classification): digit classification
- boston (regression): house prices in Boston based on the crime rate, nitric oxide concentration, number of rooms, distances to employment centers, tax rates, etc. The output feature is the median value of homes.
- diabete (regression): sklearn diabete dataset
- linnerud (regression): physical exercise Linnerud dataset

Dataset	# features	# classes	size	train size (80%)	val size (20%)
wine	13	3	178	142	36
iris	4	3	150	120	30
digits	64	10	1797	1437	360

Table 1: Information about the sklearn datasets (classification)

Dataset	# features	# output	size	train size (80%)	val size (20%)
boston	13	1	506	404	102
diabete	10	1	442	353	89
linnerud	3	3	20	16	4

Table 2: Information about the sklearn datasets (regression)

We trained a one-layer perceptron with a hidden layer of dimension 50, a leaky rectified linear unit (Leaky ReLU) activation (with a negative slope of 0.01) (Maas, 2013) and a dropout of probability 0.1 (Srivastava et al., 2014), this for 2000 epochs.

The results are presented in the following figures :

- wine: 11, 12, 13, 14, 15, 16, 17
- iris: 18, 19, 20, 21, 22, 23, 24
- digits: 25, 26, 27, 28, 29, 30, 31
- boston: 32, 33, 34, 35, 36, 37, 38
- linnerud: 39, 40, 41, 42, 43, 44, 45
- diabete: 46, 47, 48, 49, 50, 51, 52

[10] `https://scikit-learn.org/stable/datasets/toy_dataset.html`

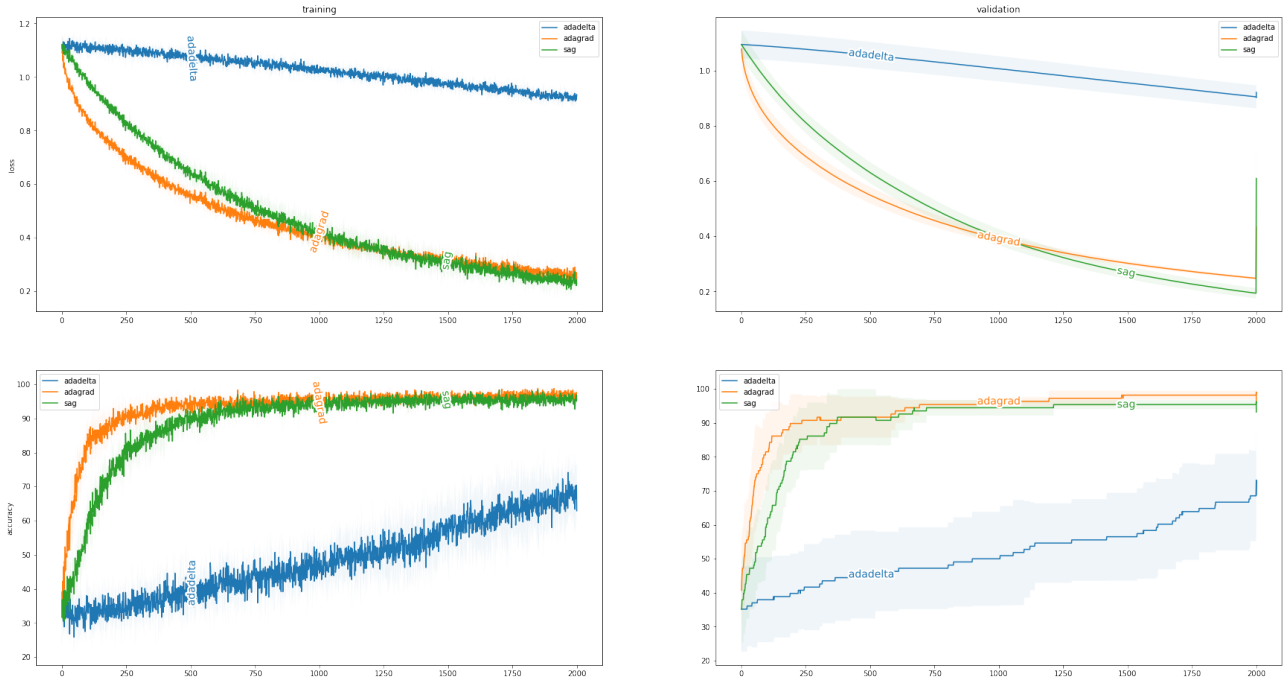


Figure 11: adadelta, adagrad, sag (wine)

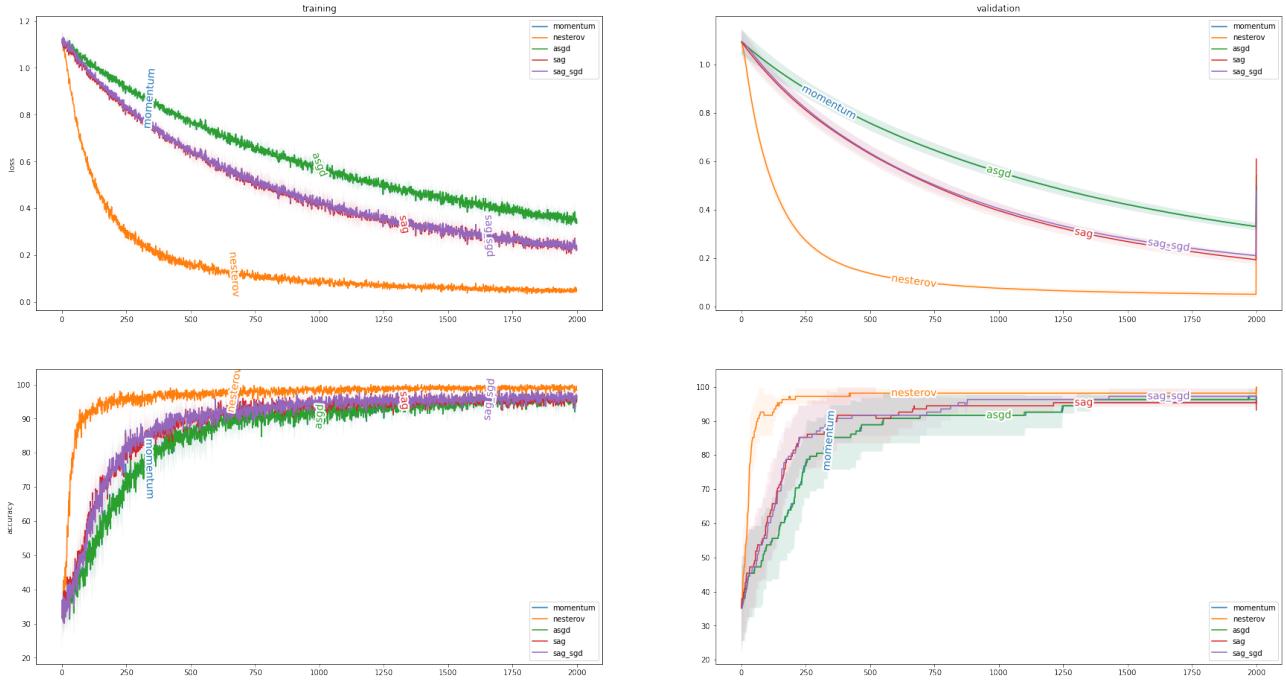


Figure 12: momentum, nesterov, asgd, sag, sag, sgd (wine)

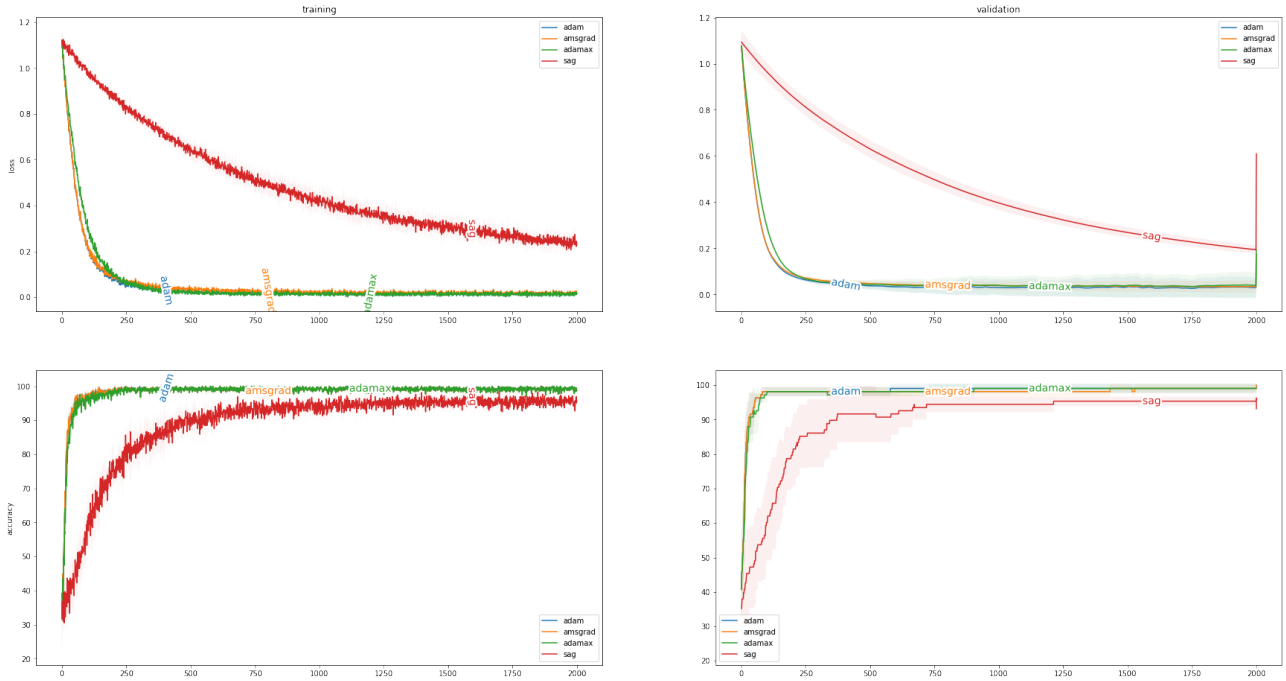


Figure 13: adam, amsgrad, adamax, sag, sag_adam (wine)

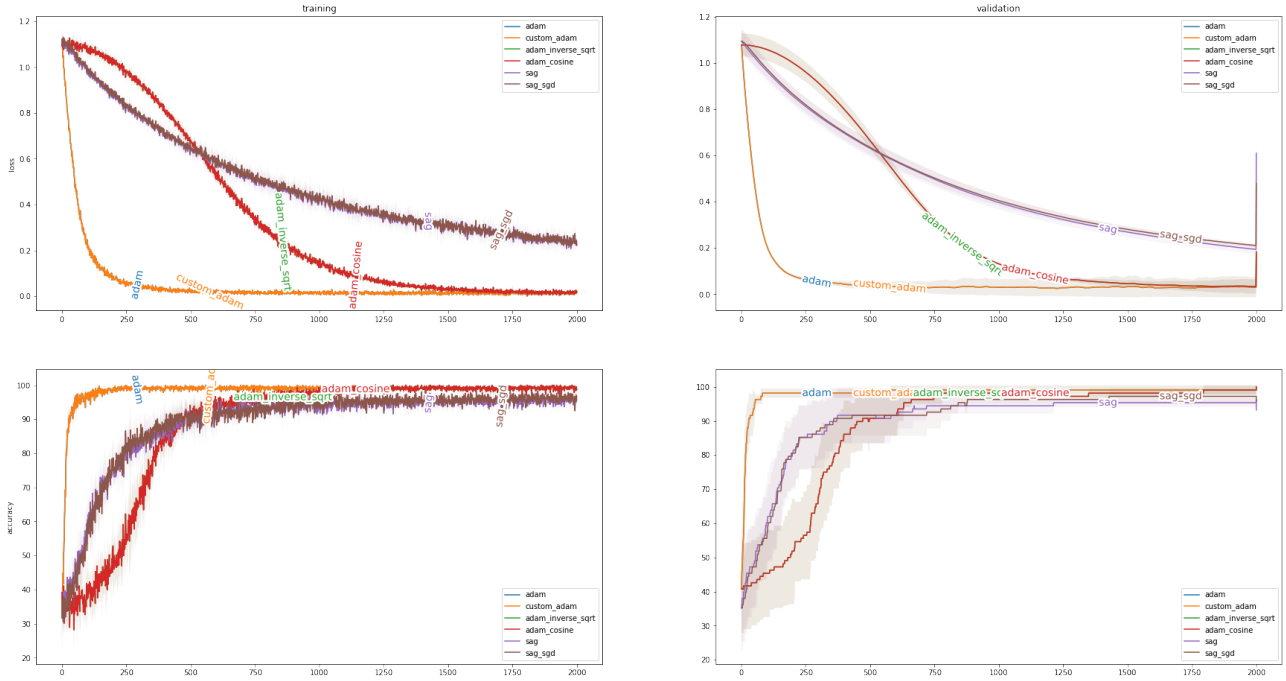


Figure 14: adam, custom_adam, adam_inverse_sqrt, adam_cosine, sag, sag_sgd, sag_adam (wine)

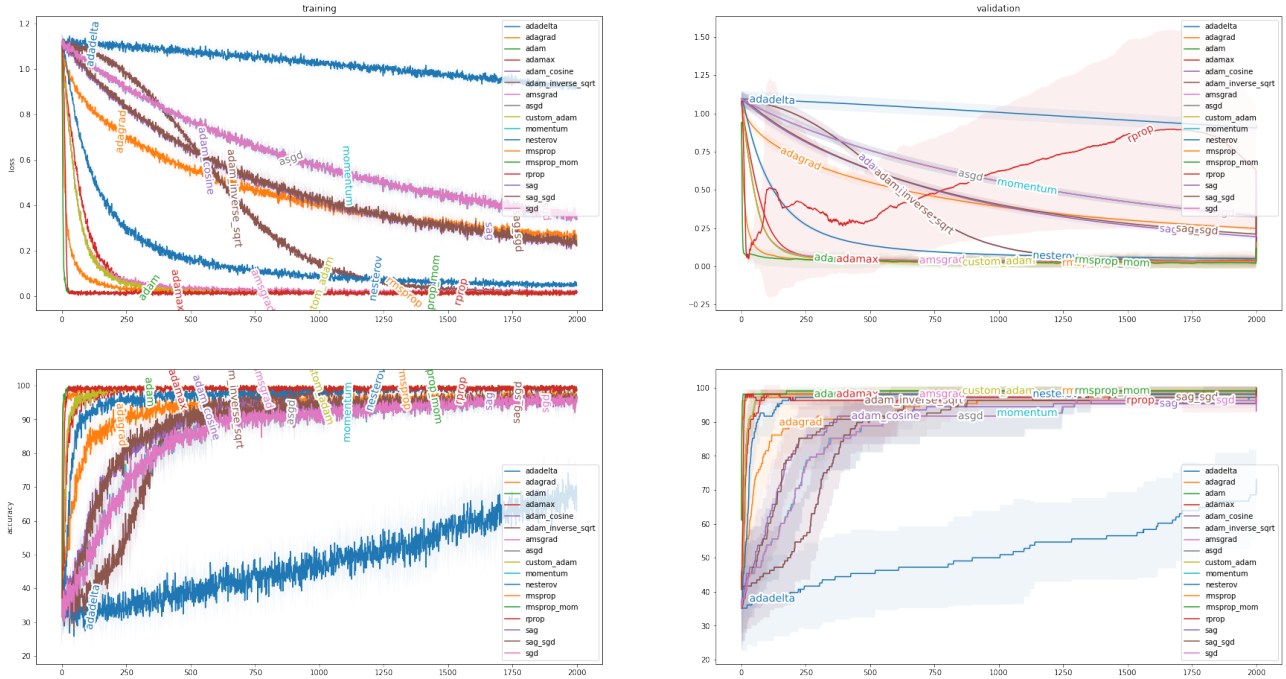


Figure 15: Summary (wine)

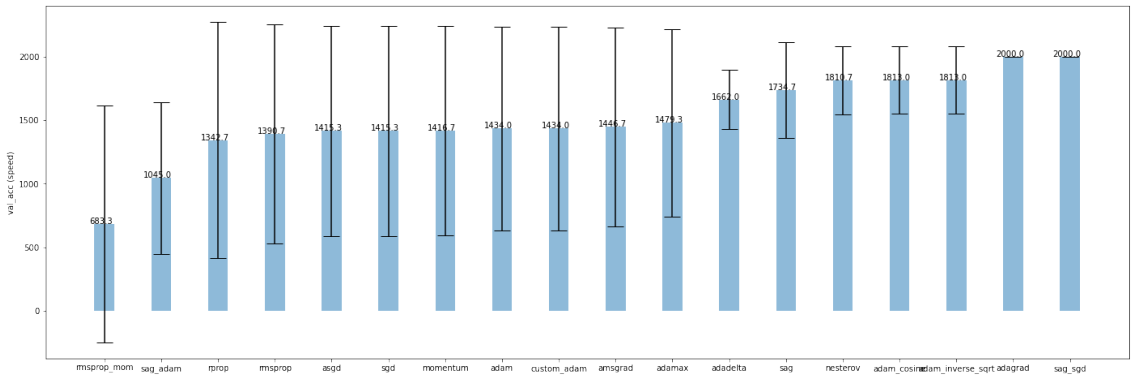


Figure 16: Comparative visualization of convergence speeds (wine)

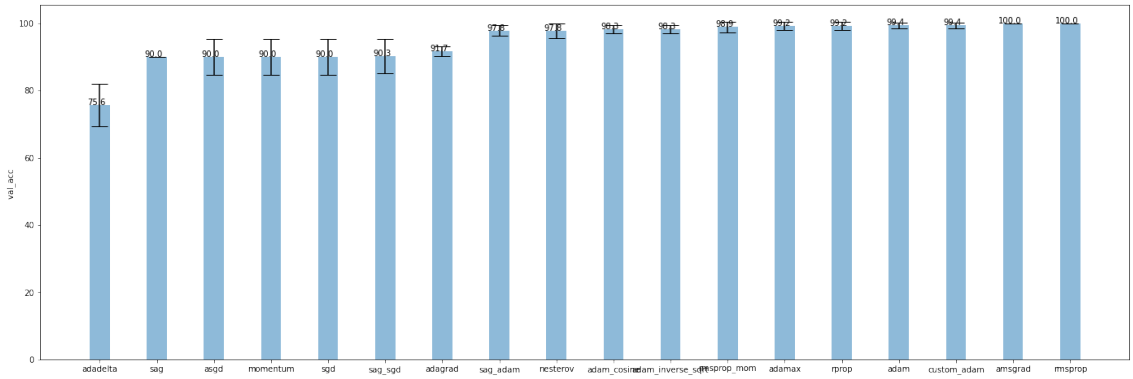


Figure 17: Performances at steady states (wine)

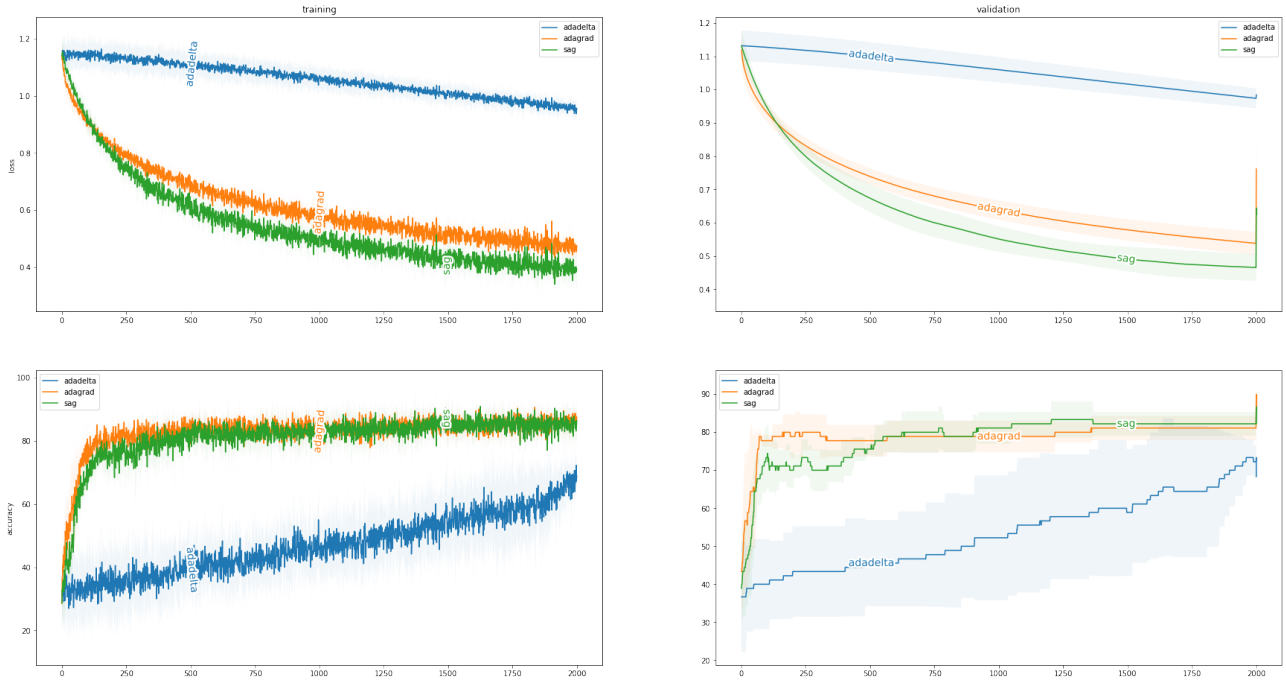


Figure 18: adadelta, adagrad, sag (iris)

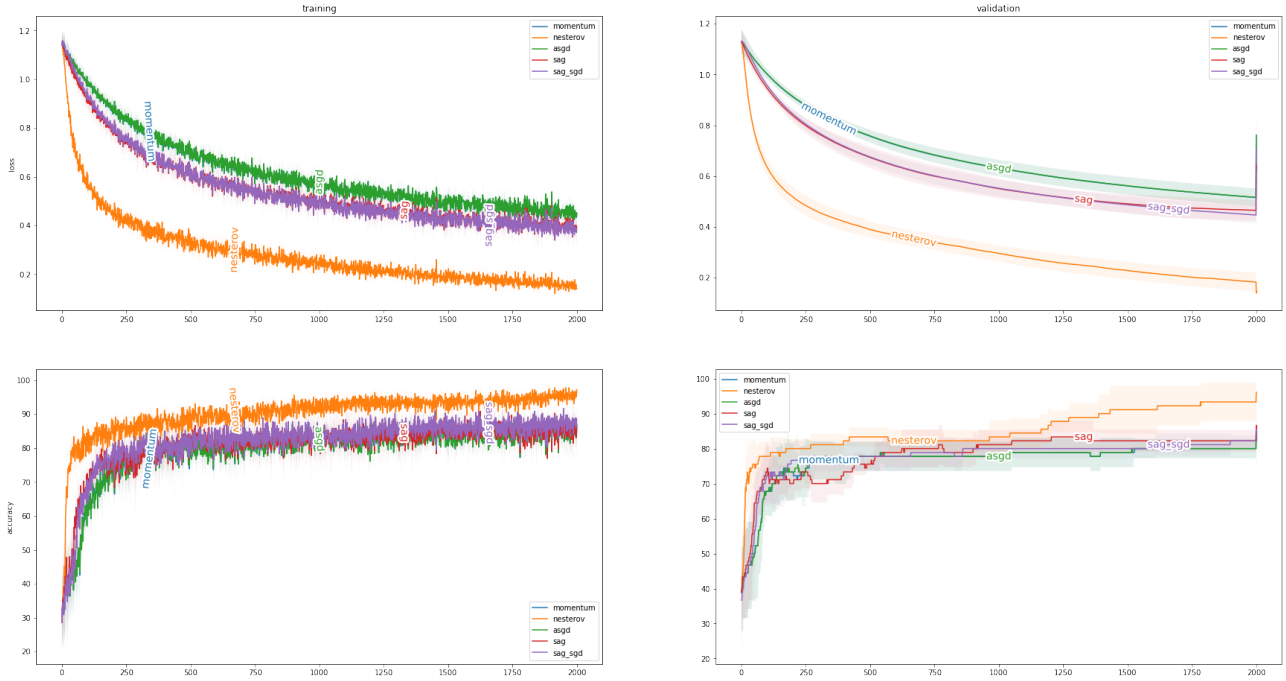


Figure 19: momentum, nesterov, asgd, sag, sag, sgd (iris)

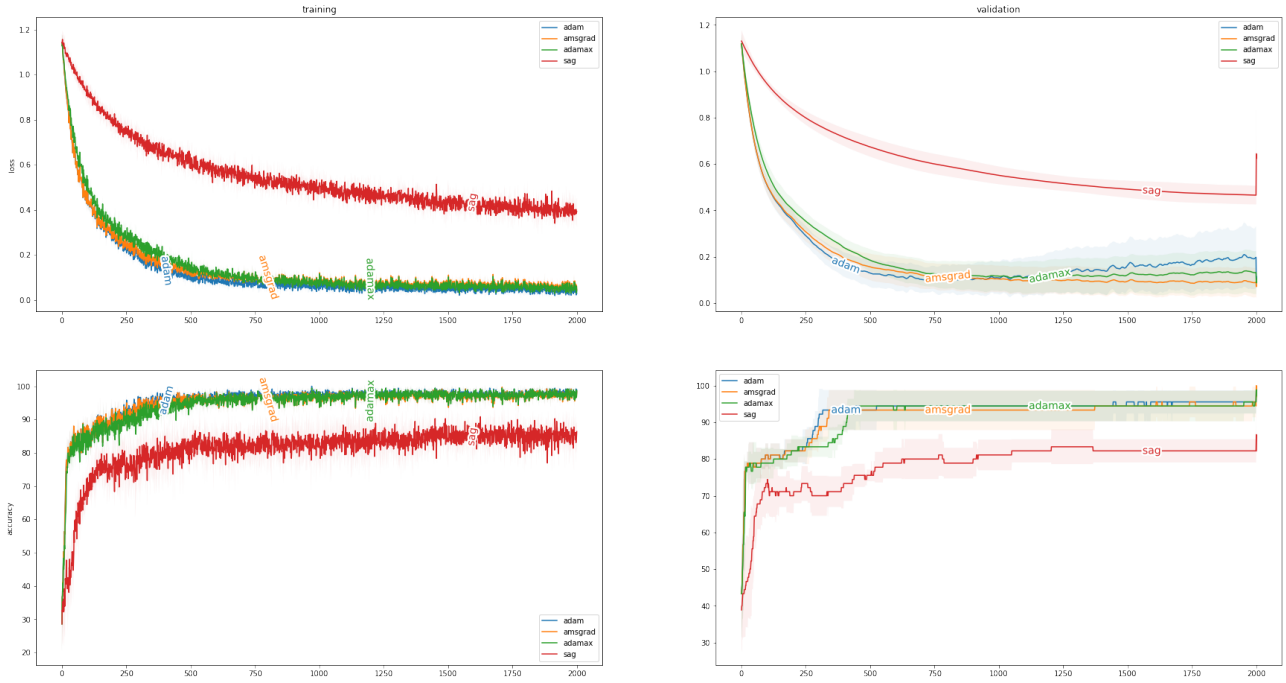


Figure 20: adam, amsgrad, adamax, sag, sag_adam (iris)

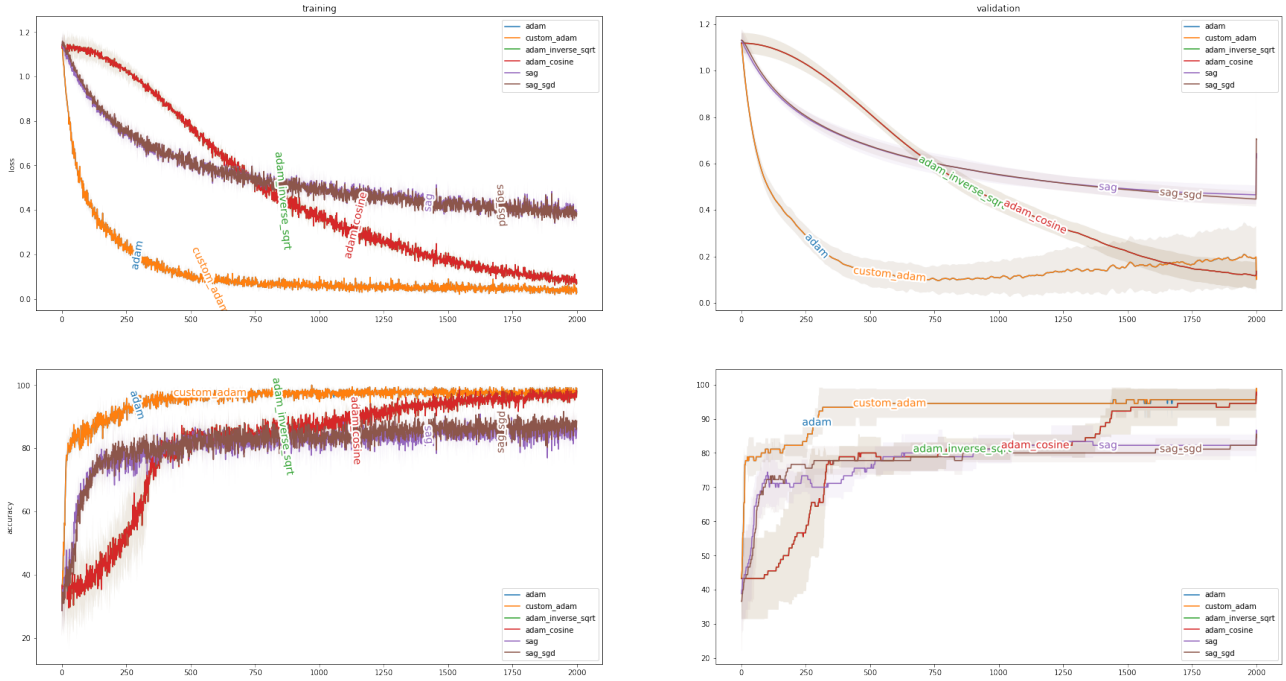


Figure 21: adam, custom_adam, adam_inverse_sqrt, adam_cosine, sag, sag_sgd, sag_adam (iris)

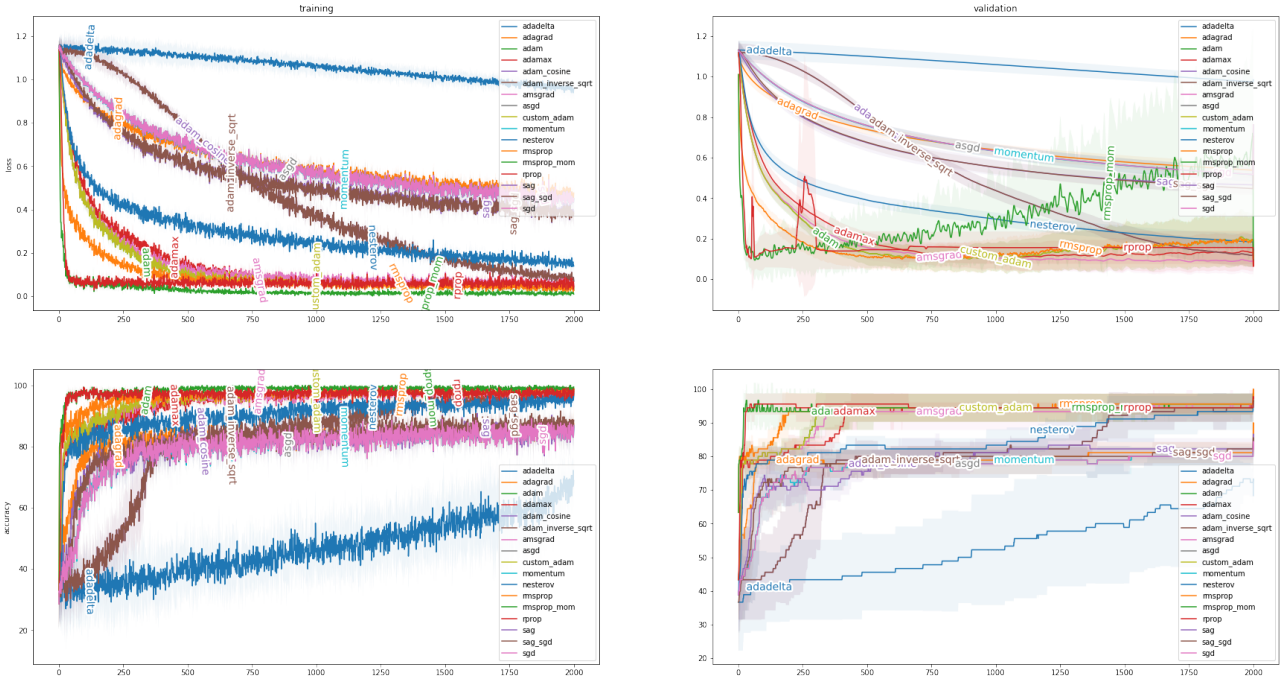


Figure 22: Summary (iris)

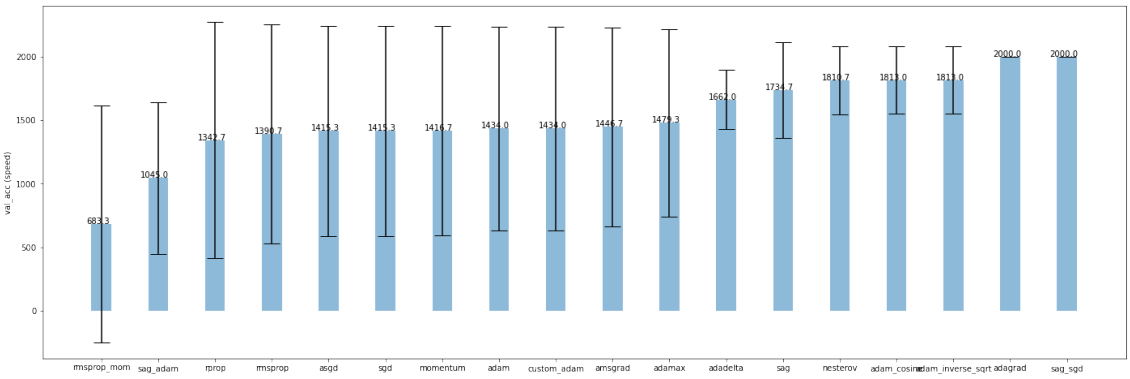


Figure 23: Comparative visualization of convergence speeds (iris)

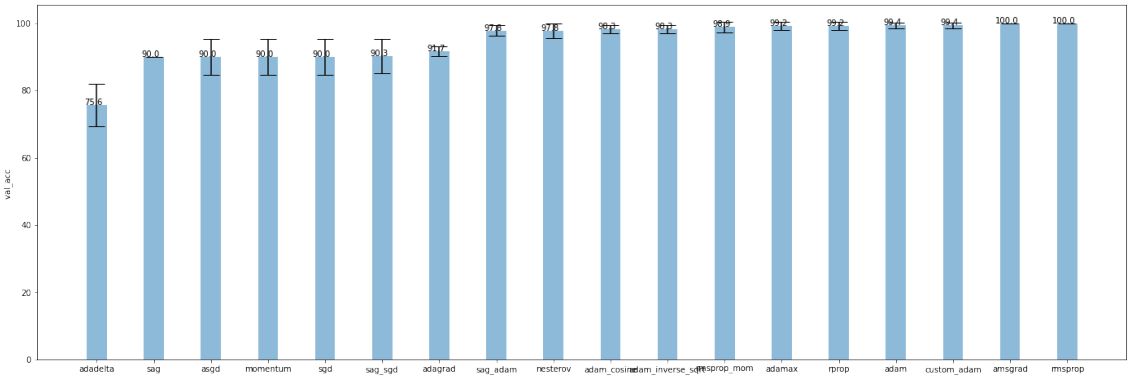


Figure 24: Performances at steady states (iris)

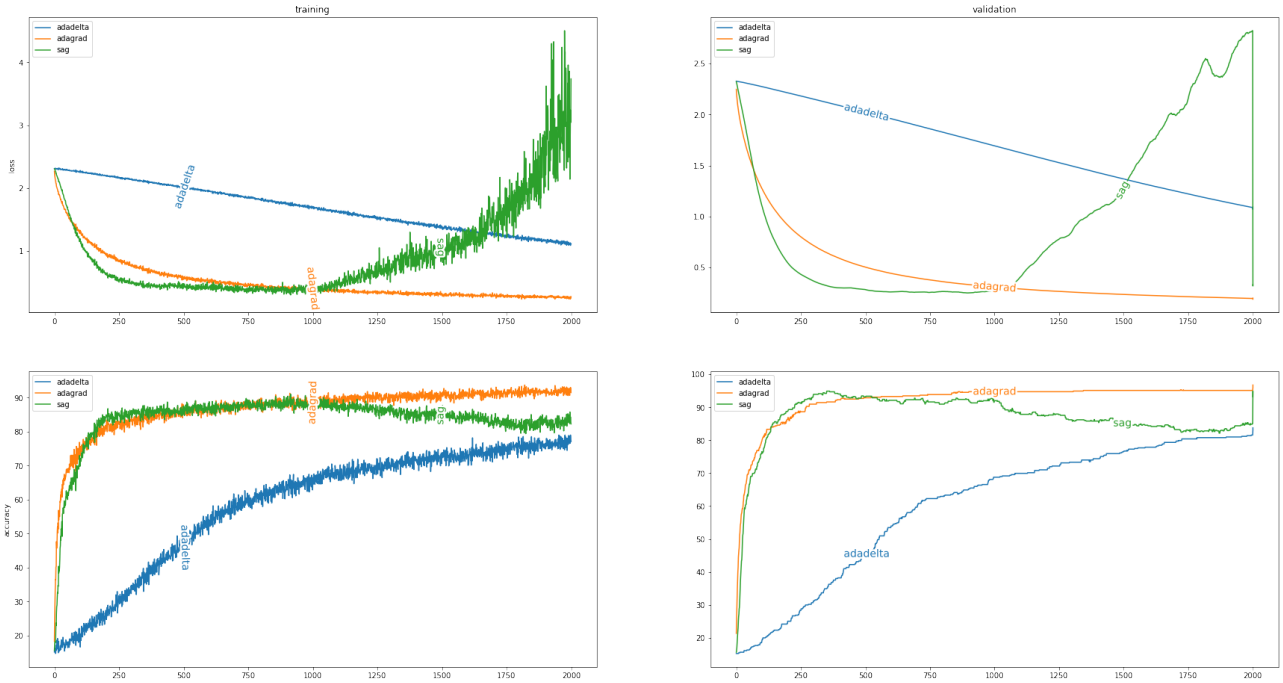


Figure 25: adadelta, adagrad, sag (digits)

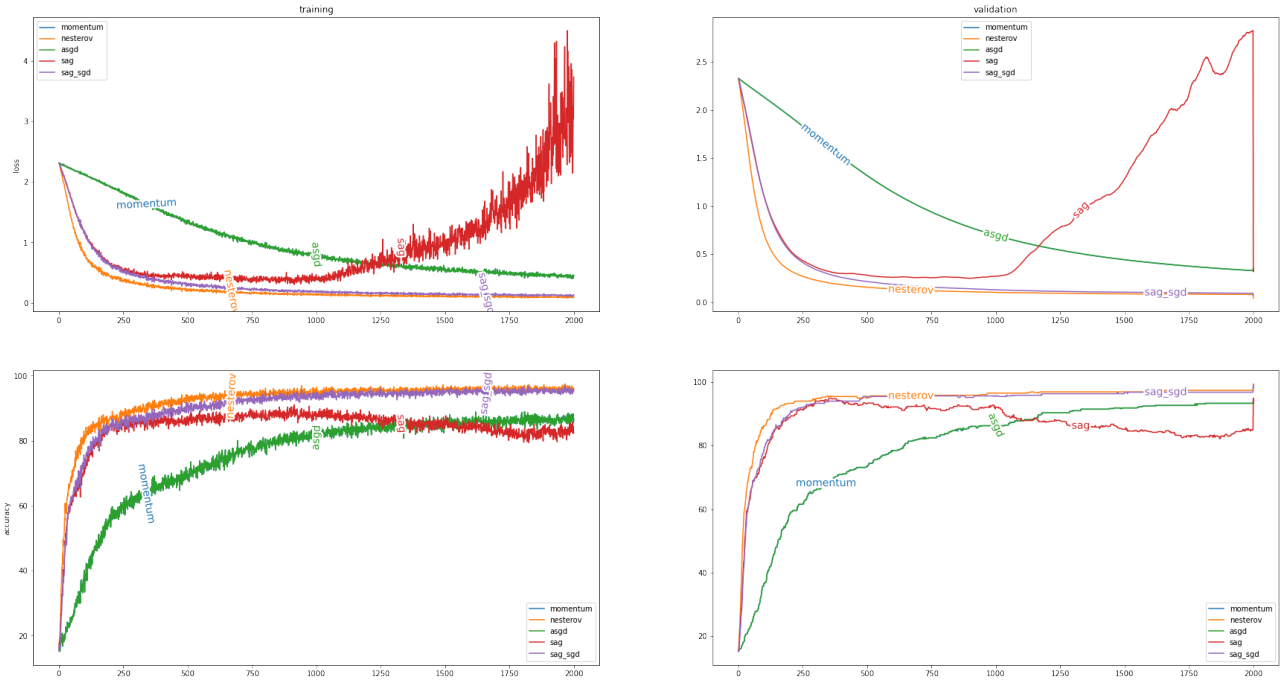


Figure 26: momentum, nesterov, asgd, sag, sag, sgd (digits)

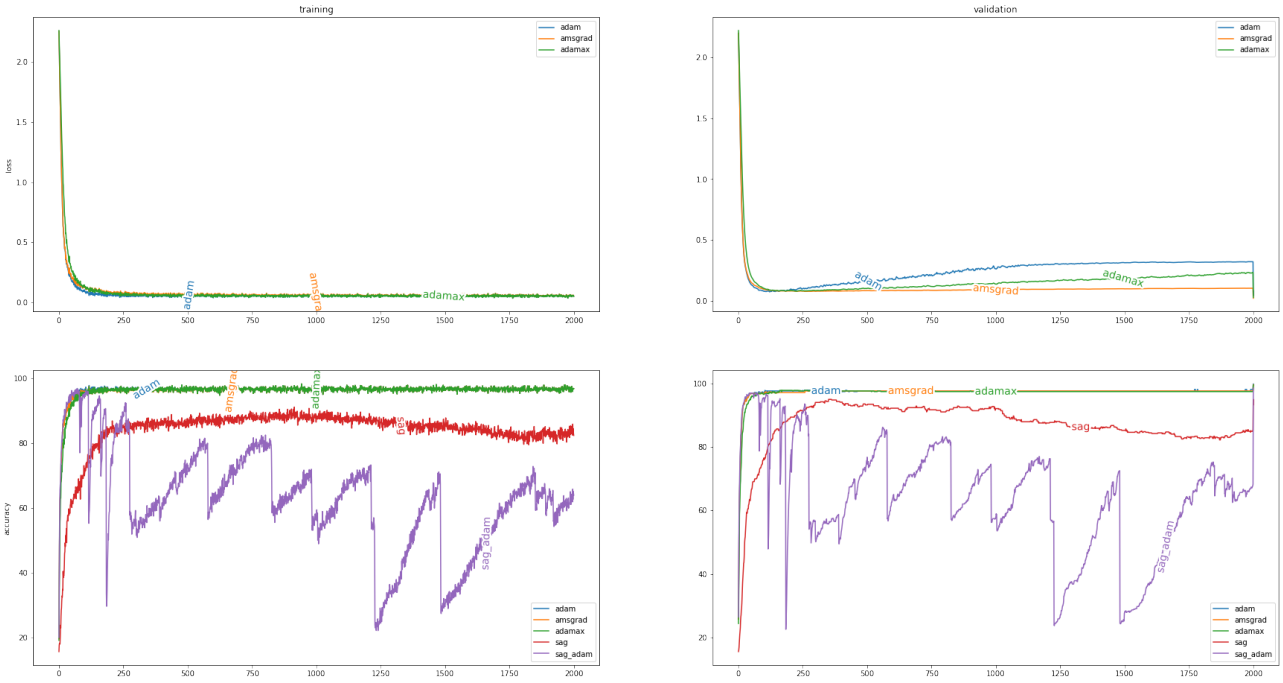


Figure 27: adam, amsgrad, adamax, sag, sag_adam (digits)

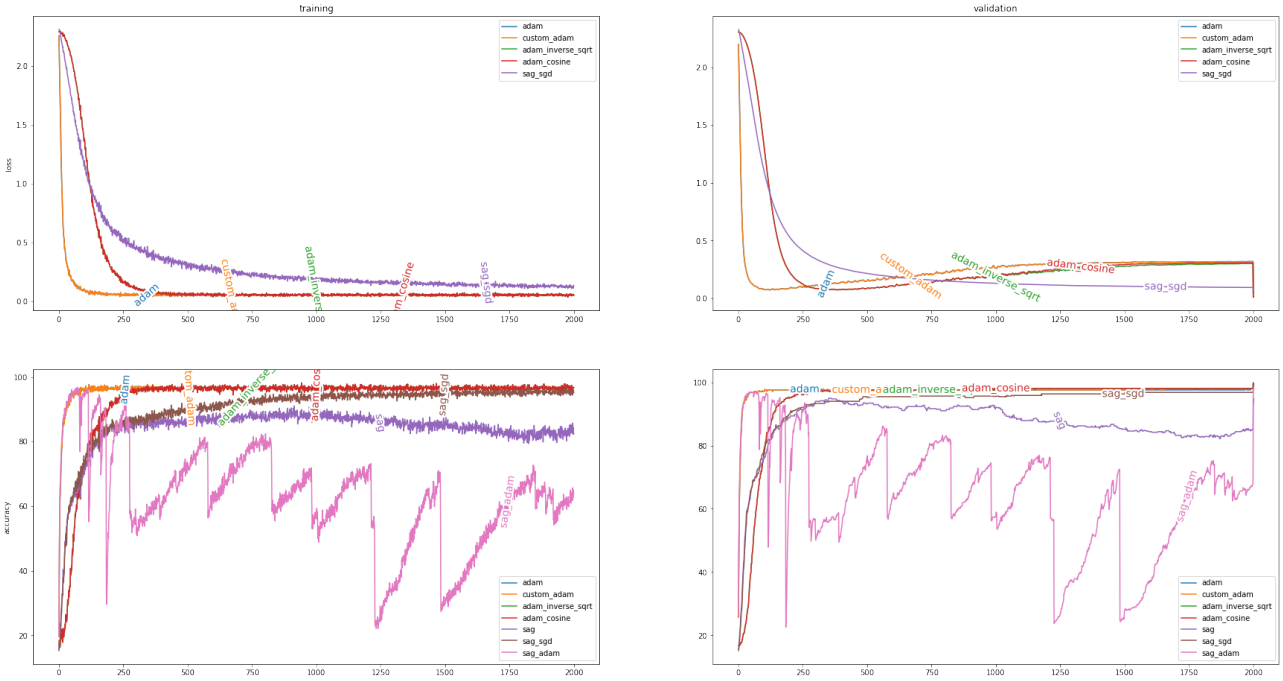


Figure 28: adam, custom_adam, adam_inverse_sqrt, adam_cosine, sag, sag_sgd, sag_adam (digits)

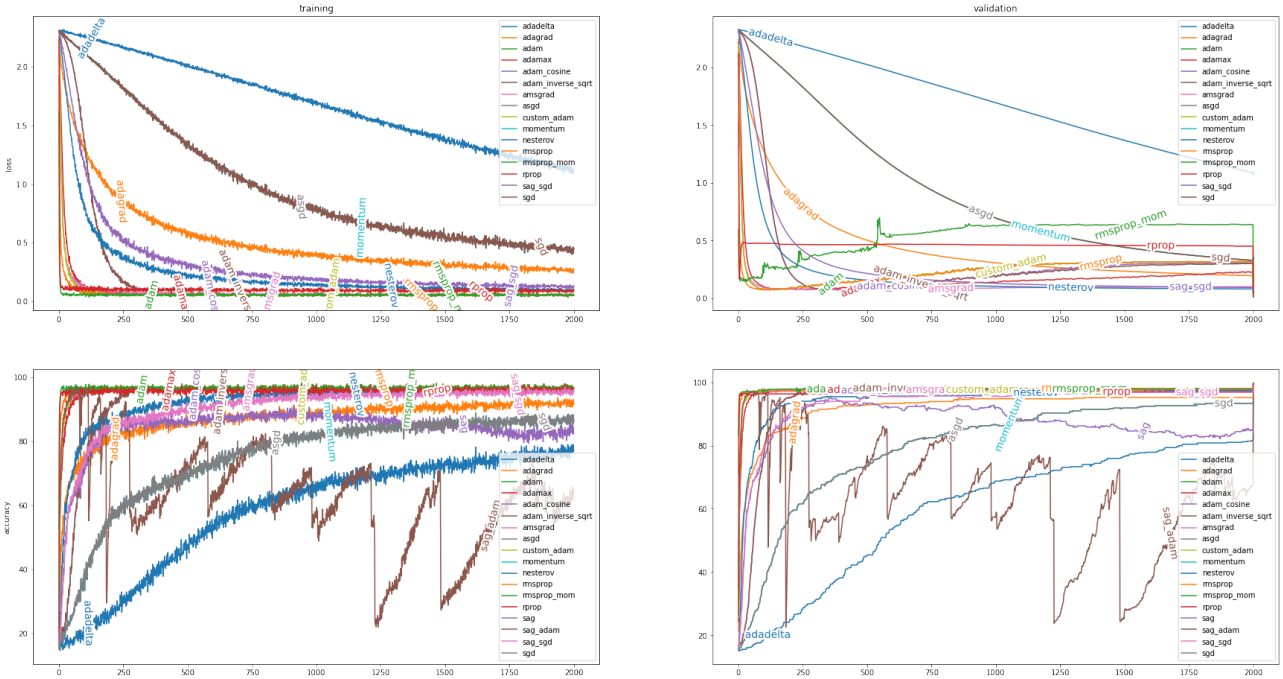


Figure 29: Summary (digits)

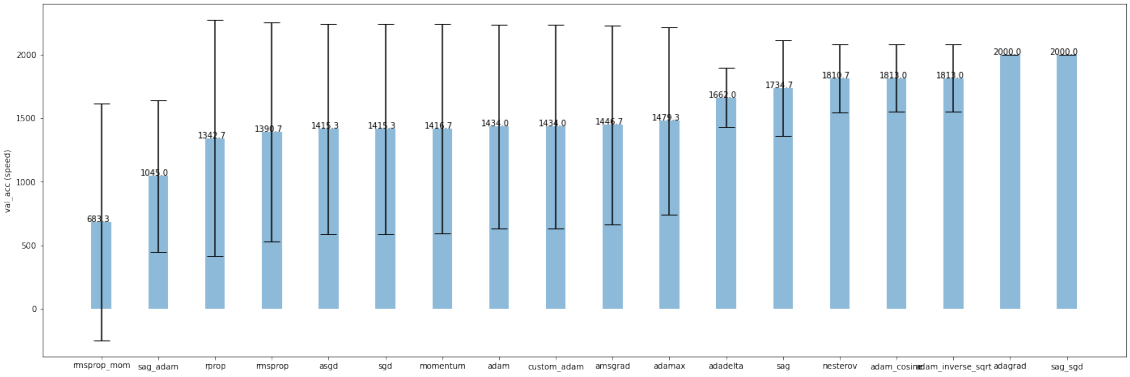


Figure 30: Comparative visualization of convergence speeds (digits)

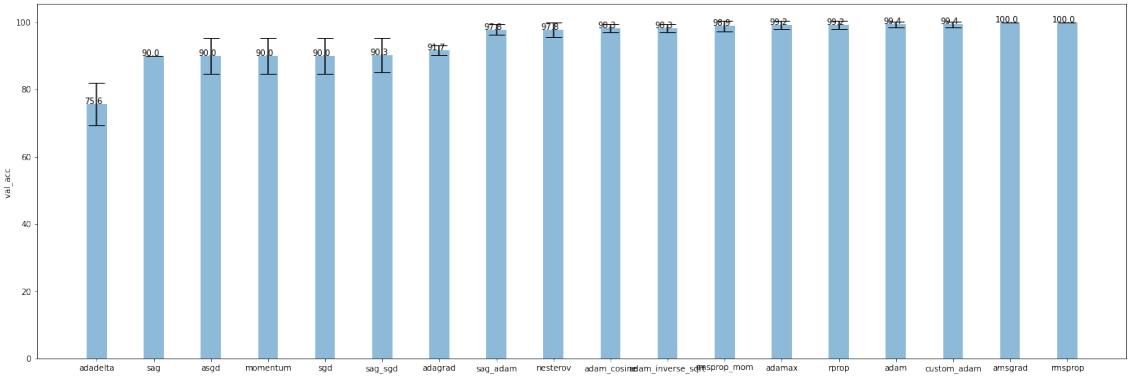


Figure 31: Performances at steady states (digits)

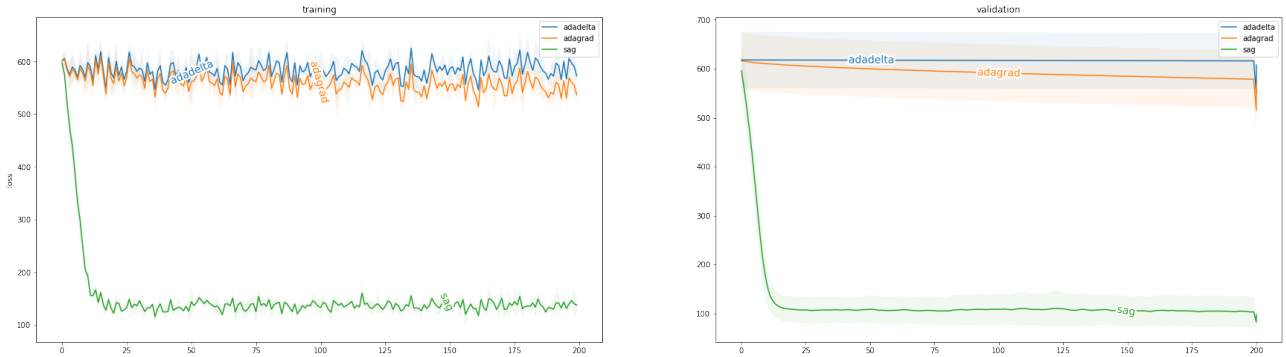


Figure 32: adadelta, adagrad, sag (boston)

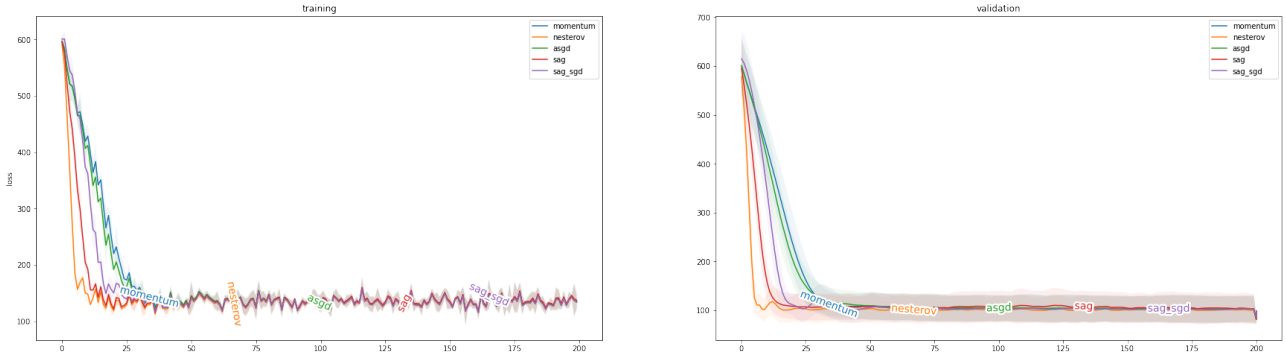


Figure 33: momentum, nesterov, asgd, sag, sag, sgd (boston)

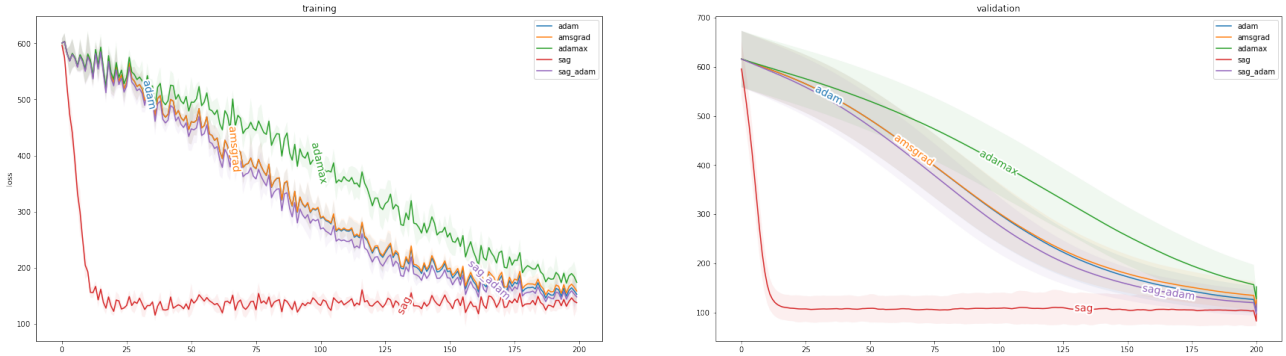


Figure 34: adam, amsgrad, adamax, sag, sag_adam (boston)

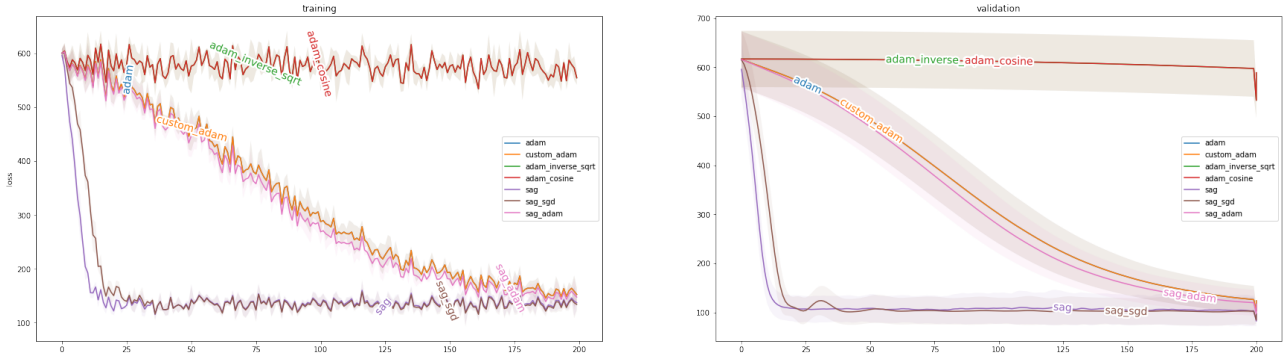


Figure 35: adam, custom_adam, adam_inverse_sqrt, adam_cosine, sag, sag_sgd, sag_adam (boston)

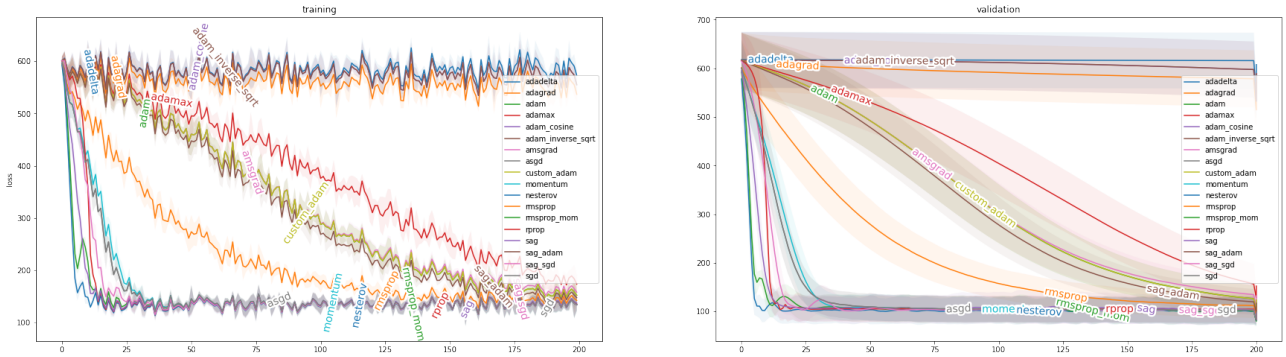


Figure 36: Summary (boston)

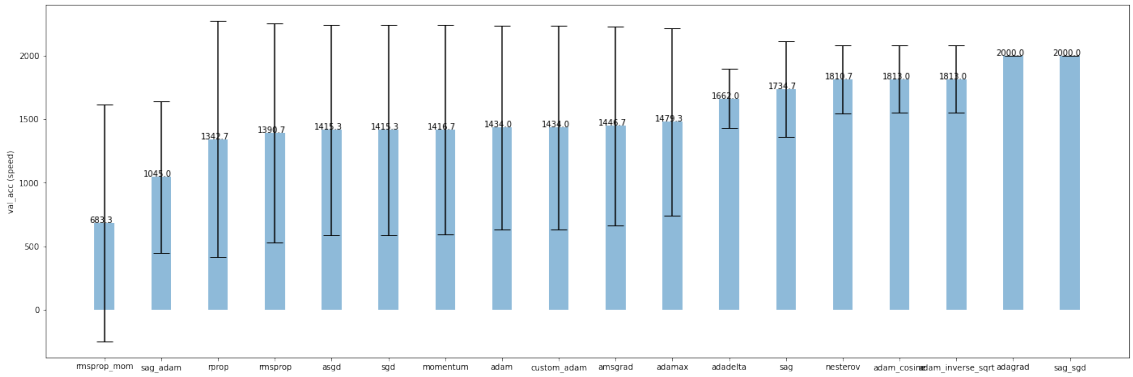


Figure 37: Comparative visualization of convergence speeds (boston)

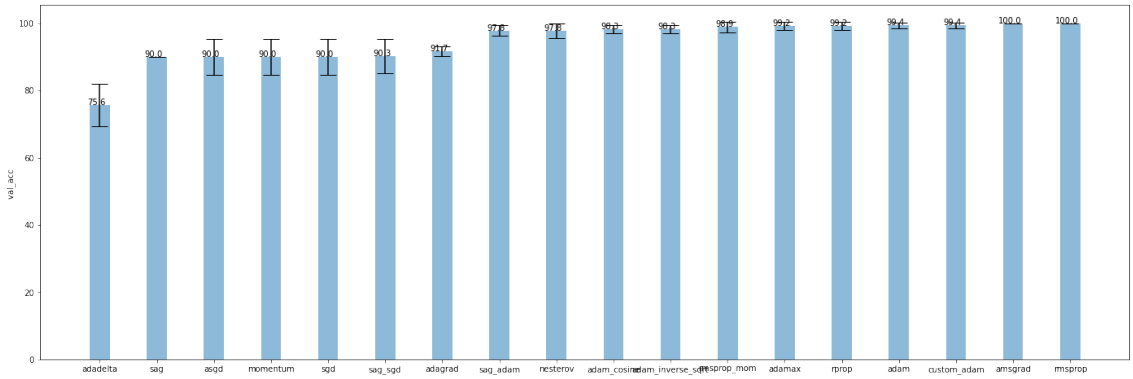


Figure 38: Performances at steady states (boston)

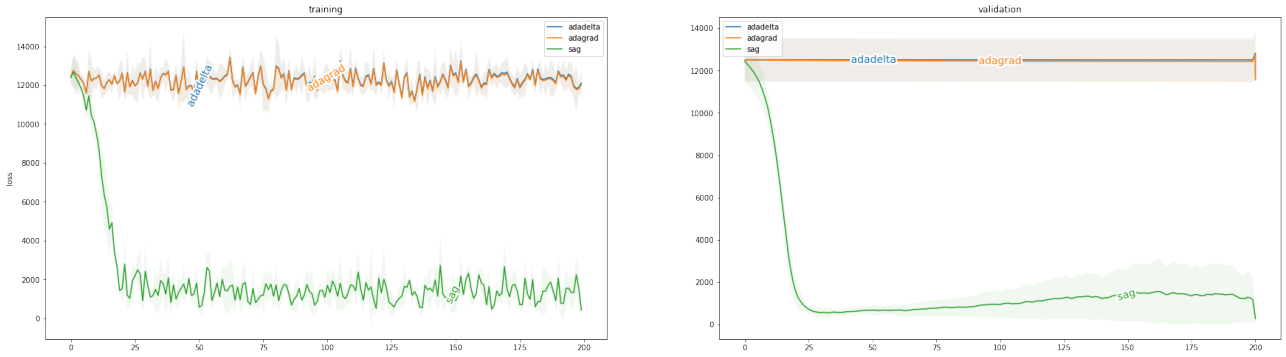


Figure 39: adadelta, adagrad, sag (linnerud)

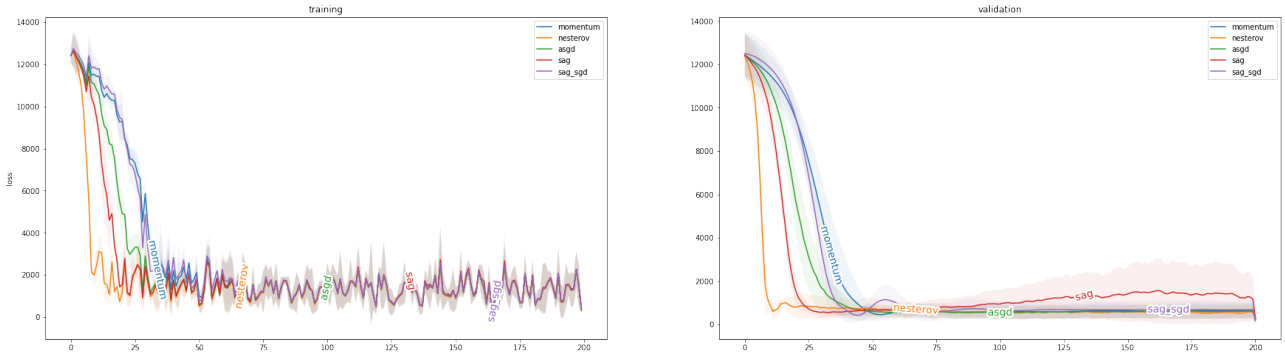


Figure 40: momentum, nesterov, asgd, sag, sag, sgd (linnerud)

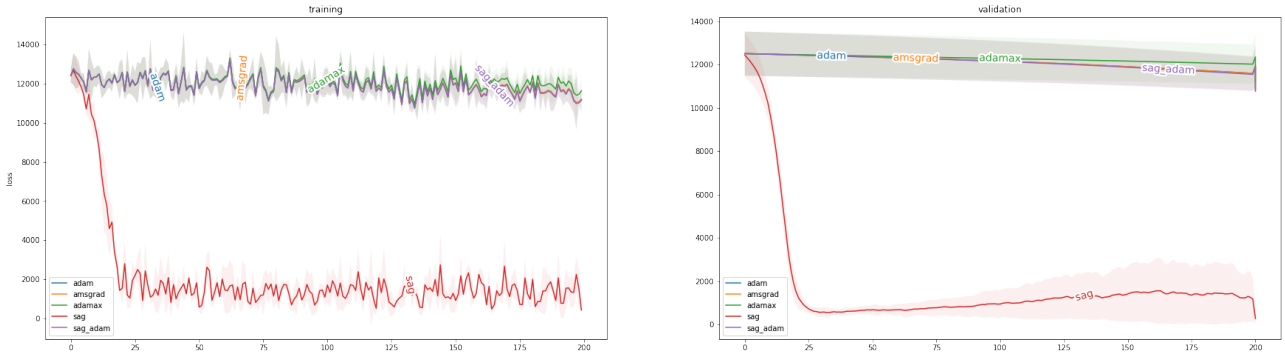


Figure 41: adam, amsgrad, adamax, sag, sag_adam (linnerud)

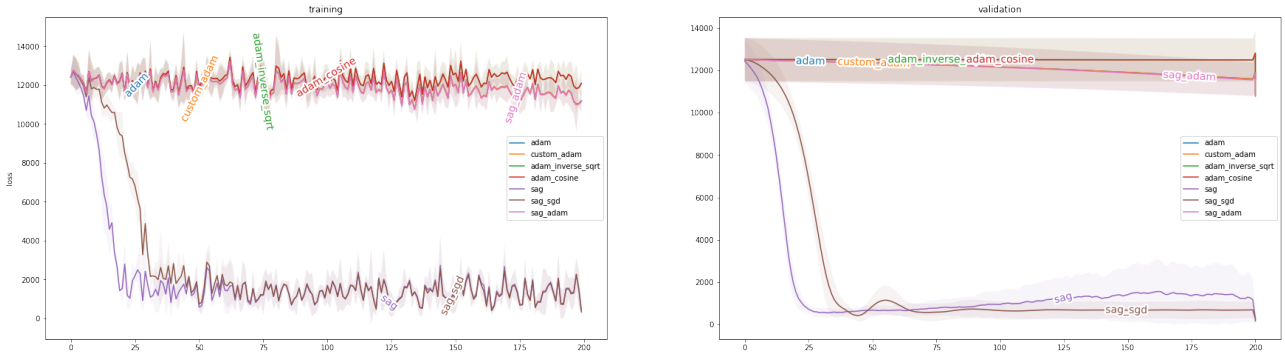


Figure 42: adam, custom_adam, adam_inverse_sqrt, adam_cosine, sag, sag_sgd, sag_adam (linnerud)

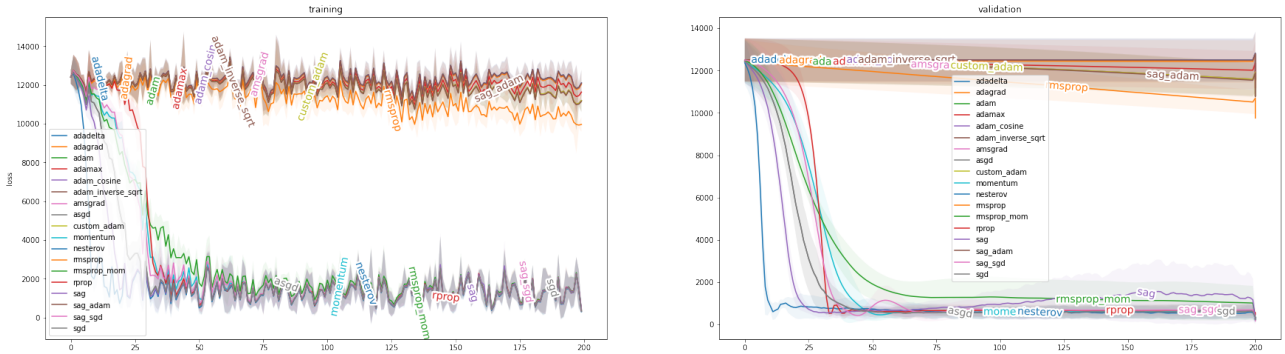


Figure 43: Summary (linnerud)

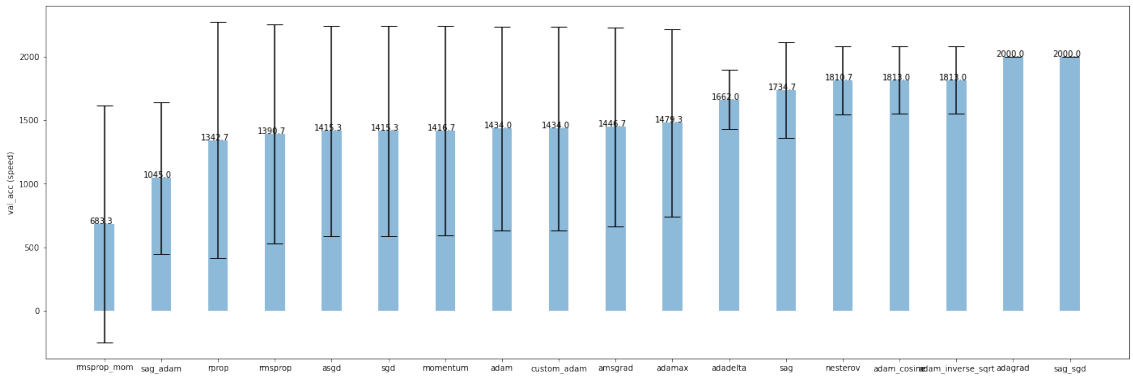


Figure 44: Comparative visualization of convergence speeds (linnerud)

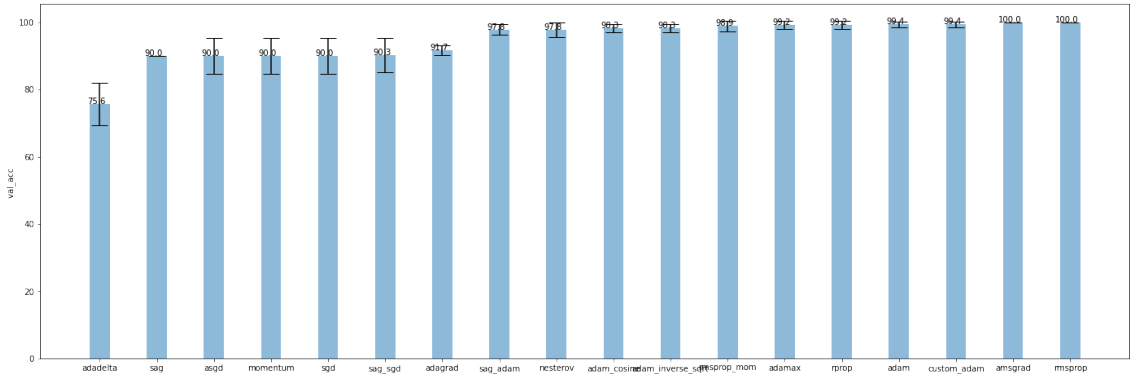


Figure 45: Performances at steady states (linnerud)

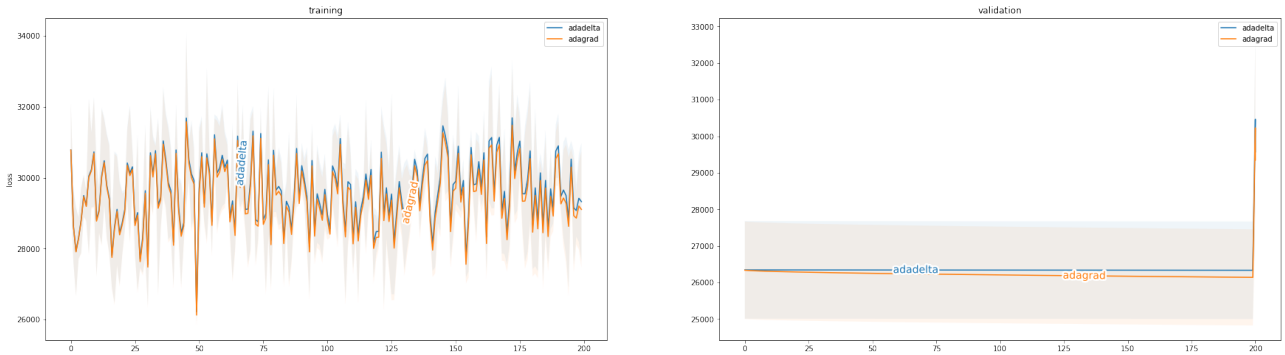


Figure 46: adadelta, adagrad, sag (diabete)

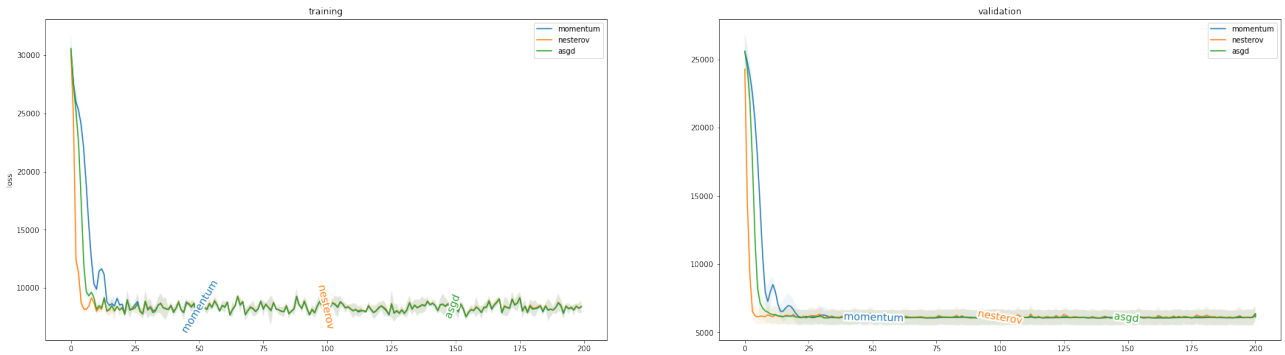


Figure 47: momentum, nesterov, asgd, sag, sag, sgd (diabete)

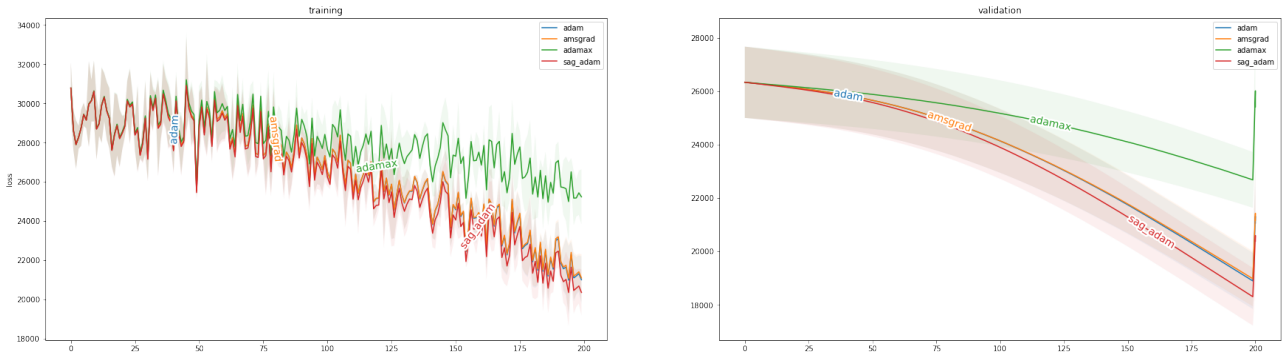


Figure 48: adam, amsgrad, adamax, sag, sag_adam (diabete)

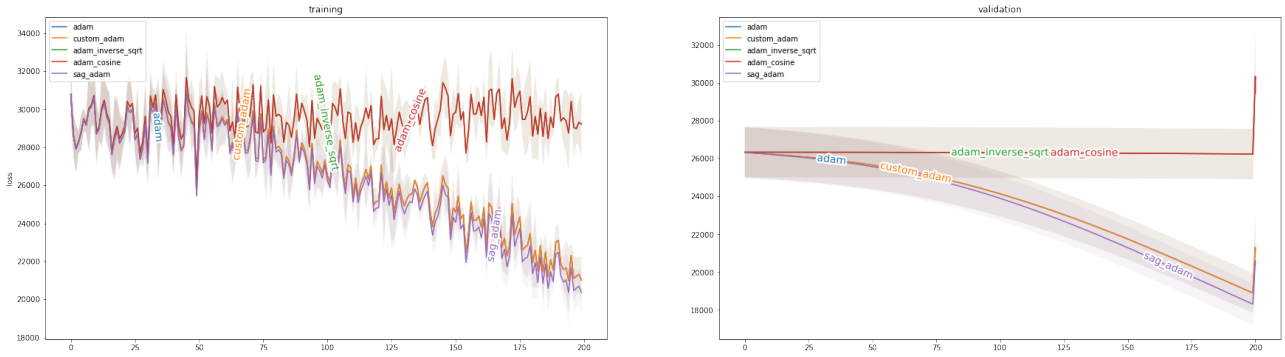


Figure 49: adam, custom_adam, adam_inverse_sqrt, adam_cosine, sag, sag_sgd, sag_adam (diabete)

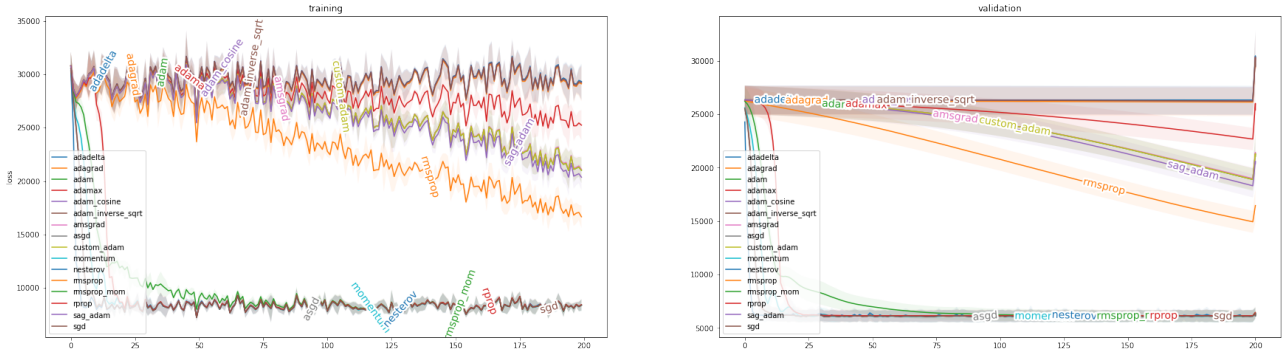


Figure 50: Summary (diabete)

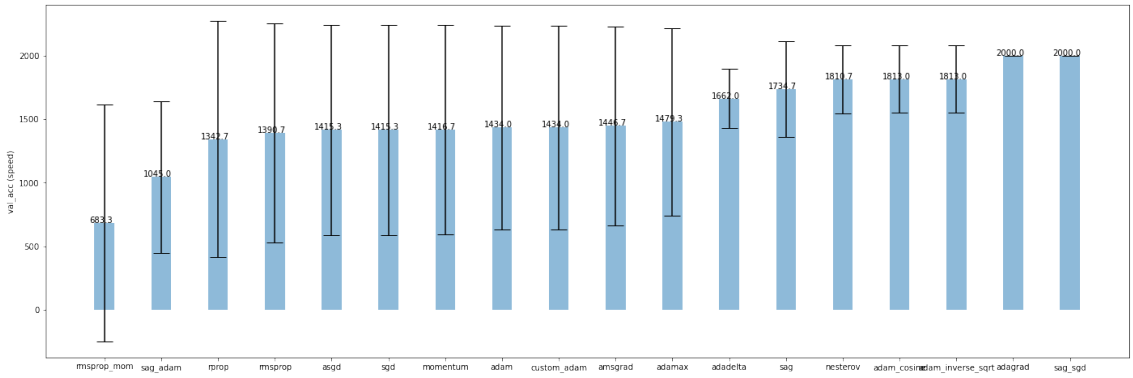


Figure 51: Comparative visualization of convergence speeds (diabete)

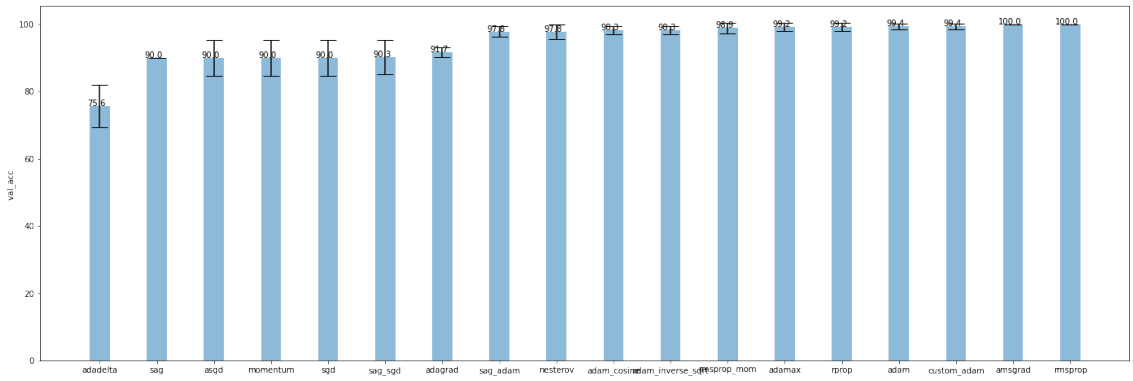


Figure 52: Performances at steady states (diabete)

7.2.2 TorchVision dataset

We extracted the datasets presented in table 3 from pytorch (Paszke et al., 2019). The reader is kindly invited to refer to the official pytorch website [11] for more information about these data (sources, ...).

Dataset	(# channels, height, width)	# classes	size	train size (80%)	val size (20%)
mnist	(1, 28, 28)	10	70000	60000	10000
fashion mnist	(1, 28, 28)	10	7000	60000	10000
cifar10	(3, 32, 32)	10	60000	50000	10000
cifar100	(3, 32, 32)	100	60000	50000	10000

Table 3: TorchVision datasets

We trained a classifier having two main successive parts : , and

- A first part consisting of two layers of convolutions neural networks :
  - (0): Conv2d(# channels, 10, kernel_size=(5, 5), stride=(1, 1))
  - (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  - (2): Conv2d(10, 10, kernel_size=(5, 5), stride=(1, 1))
  - (3): Dropout2d(p=0.1, inplace=False)
  - (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
- A second part consisting of a a two layers feed forward neural network :
  - (0): Linear(in_features=160, out_features=50, bias=True)
  - (1): Dropout(p=0.1, inplace=False)
  - (2): Linear(in_features=50, out_features=10, bias=True)
  - (3): Dropout(p=0.1, inplace=False)

The results are presented in the following figures :

- wine: in progression
- iris: in progression
- boston: in progression
- linnerud: in progression
- diabete: in progression

8 Summary and Discussion

In this work, we compared the performance of SAG and several other optimization algorithms for continuous objectives such as SGD with momentum, Nesterov Accelerated SGD, Averaged SGD, RMSProp (with and without momemtum), resilient backpropagation algorithm (Rprop), Adadelta, Adagrad, Adam, AMSGrad, Adamax, Adam with special learning rate decay procedure (inverse square root of the update number, cyclical schedule that follows the cosine function). SAG, although with a simple iteration, outperforms the majority of these algorithms. We have proposed two combinations of SAG. One with the momentum algorithm, which allows control of the importance of each gradient term in the mean used by SAG depending on the iteration during which it is used, and another with Adam where the importance of the square of the norm of the gradient is also controlled. These two variants allowed us to improve the speed empirically while obtaining better performances.

Limitations The memory cost used by SAG is very high compared to other algorithms, which makes it impractical for large scale use.

Perspectives What we presented as an improvement is only an empirical illustration of the performance of SAG. It would be interesting to evaluate theoretically the expected convergence rate of all these algorithms. We leave this for future work.

[11] `https://pytorch.org/vision/stable/datasets.html`

Acknowledgement

The authors thank Fabian Bastin who made this work possible, and for discussion at the early stage of this project during the stochastic programming (IFT6512) course at UdeM (Université de Montréal). We also thank Compute Canada for computational resources.

References

Léon Bottou. Online algorithms and stochastic approximations. In David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998. URL `http://leon.bottou.org/papers/bottou-98x`. revised, oct 2012.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Augustin-Louis Cauchy. Analyse mathématique. – méthode générale pour la résolution des systèmes d'équations simultanées. volume 25, pages 536–538, 1847.

Yann N. Dauphin, Razvan Pascanu, Çaglar Gülçehre, KyungHyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 2933–2941, 2014. URL `https://proceedings.neurips.cc/paper/2014/hash/17e23e50bedc63b4095e3d8204ce063b-Abstract.html`.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12(null):2121–2159, July 2011a. ISSN 1532-4435.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011b. URL `http://jmlr.org/papers/v12/duchi11a.html`.

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

G. Hinton. Neural networks for machine learning. coursera, video lectures, 2012.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference On Learning Representations*, 2014.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

Schalk Kok and Carl Sandrock. Locating and characterizing the stationary points of the extended rosenbrock function. *Evol. Comput.*, 17(3):437–453, sep 2009. ISSN 1063-6560. doi: 10.1162/evco.2009.17.3.437. URL `https://doi.org/10.1162/evco.2009.17.3.437`.

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

Andrew L. Maas. Rectifier nonlinearities improve neural network acoustic models. 2013.

H. B. McMahan and M. Streeter. Adaptive bound optimization for online convex optimization. *Annual Conference Computational Learning Theory*, 2010.

Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep double descent: Where bigger models and more data hurt. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL `https://openreview.net/forum?id=B1g5sA4twr`.

A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on Optimization*, 19(4):1574–1609, 2009. doi: 10.1137/070704277. URL `https://doi.org/10.1137/070704277`.

Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$. 1983.

Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*. Springer, 2004.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

B. T. Polyak and A. B. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM J. Control Optim.*, 30(4):838–855, jul 1992. ISSN 0363-0129. doi: 10.1137/0330046. URL https://doi.org/10.1137/0330046.

B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964. ISSN 0041-5553. doi: https://doi.org/10.1016/0041-5553(64)90137-5. URL https://www.sciencedirect.com/science/article/pii/0041555364901375.

Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *International Conference On Learning Representations*, 2018.

M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: the rprop algorithm. In *IEEE International Conference on Neural Networks*, pages 586–591 vol.1, 1993. doi: 10.1109/ICNN.1993.298623.

Mark W. Schmidt, Nicolas Le Roux, and F. Bach. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, 2013. doi: 10.1007/s10107-016-1030-6.

Sargur N. Srihari. Challenges in neural network optimization. 2020. URL https://cedar.buffalo.edu/~srihari/CSE676/.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. URL http://proceedings.mlr.press/v28/sutskever13.html.

Richard S. Sutton. Two problems with backpropagation and other steepest-descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum, 1986.

Vimal Thilak, Etai Littwin, Shuangfei Zhai, Omid Saremi, Roni Paiss, and Joshua Susskind. The slingshot mechanism: An empirical study of adaptive optimizers and the grokking phenomenon, 2022. URL https://arxiv.org/abs/2206.04817.

Paul Tseng. An incremental gradient(-projection) method with momentum term and adaptive stepsize rule. *SIAM Journal on Optimization*, 8(2):506–531, 1998. doi: 10.1137/S1052623495294797. URL https://doi.org/10.1137/S1052623495294797.

Rachel Ward, Xiaoxia Wu, and Leon Bottou. Adagrad stepsizes: Sharp convergence over nonconvex landscapes. *The Journal of Machine Learning Research*, 21(1):9047–9076, 2020.

Matthew D. Zeiler. Adadelta: An adaptive learning rate method, 2012.