ORIGINAL ARTICLE

First-order optimization algorithms: state of the art, classification, and performance: a practitioner's guide

Ruslan Shaiakhmetov · Danilo Pianini · Angelo Filaseta · Gabriele D'Angelo · Valter Venusti

Received: 4 September 2025 / Accepted: 4 February 2026 / Published online: 26 March 2026
© The Author(s) 2026

Abstract

Driven by the rising interest in machine learning techniques, mathematical continuous optimization algorithms have made great progress. Among them, first-order optimization algorithms, which rely on the first derivative (gradient) to find a function's minimum or maximum, have gained popularity due to their efficiency and scalability. As a result, a vast number of optimization algorithms have been developed, each applying different techniques and offering diverse guarantees. This variety, while beneficial, makes selecting the most appropriate algorithm a challenging yet crucial task–choosing the wrong one may lead to sub-par accuracy or performance. This paper explores the state of the art in continuous first-order optimization algorithms, offering guidance for selecting the most suitable method. We classify 23 algorithms, detailing their dependency relationships, theoretical foundations, and optimization strategies. The analysis includes a performance evaluation using implementations in the PyTorch framework. Convergence, quantified by the area under the training-loss curve, is assessed with two benchmarks: the Rosenbrock function as a standard test and ResNet-18 training on the CIFAR-10 dataset as a practical test. We evaluate performance using an integral metric and analyze robustness to hyperparameter variations, including learning rate sensitivity. Additionally, we introduce a classification of algorithm convergence behaviors. These experiments provide insights into algorithm performance across varying problem complexities and highlight their stability under hyperparameter changes. Practitioners and researchers can use this work as a guide to identify the set of most likely good candidates as first-order optimization algorithms for their use case.

Keywords First-order optimization · Performance evaluation · Machine learning · Convergence analysis · Robustness analysis

1 Introduction

Mathematical optimization represents the way of selecting the most favorable choice from a given set of alternatives, all of which are evaluated against a specified criterion. The categorization of optimization problems primarily hinges on two fundamental aspects: the nature of the elements involved, which can be either continuous

 https://doi.org/10.1007/s00521-026-12014-1

 https://doi.org/10.1007/s00521-026-12014-1

or discrete, and the existence of constraints. In the context of this paper, our primary focus is directed toward continuous optimization problems that are devoid of constraints.

Continuous unconstrained optimization holds a pivotal role across various domains, encompassing fields as diverse as including biology [1], economics [2], and machine learning [3]. Remarkably, the advancement of machine learning has played a driving role in the evolution of optimization algorithms [4]. This stems from the necessity to tackle large-scale optimization problems when training neural networks.

Optimization algorithms fall into two main categories: first-order (also referred to as *gradient-based*) and second-order methods. The former uses a first-order approximation of a function's curvature, while the latter leverage second-order curvature information, like the Hessian matrix or its approximation [5]. The availability of higher-order information usually reduces the number of steps needed to converge, but it require more computational resources per each step. The speed at which an algorithm converges to a solution is termed "convergence rate", a key categorization metric for optimization algorithms. First-order methods show linear or superlinear convergence, whereas second-order methods achieve quadratic convergence [6]. However, as already stated, second-order methods require higher computational costs for each search step, making them more suitable for low-dimensional problems. Conversely, first-order methods are preferred for high-dimensional (over 100-dimensional) scenarios [7]. Typically, all modern neural network applications can be seen as Large-scale global optimization (LSGO) problems [3, 8, 9]; thus, in this work, we keep our focus centered on first-order methods.

Additionally, we intentionally exclude Nature-inspired Algorithms (NIAs) and other derivative-free, population-based metaheuristics such as genetic algorithms, particle swarm optimization, differential evolution, chaotic Puma optimization [10], artificial bee colony [11] and firefly [12] algorithms, or arithmetic optimization [13]: these algorithms treat the objective as a black box and explore the parameter space via *stochastic sampling* rather than *following gradient* information.

They therefore target non-differentiable or highly multi-modal (typically low–moderate-dimensional) objectives – where gradient methods are inapplicable – whereas our scope is large-scale smooth problems with accessible gradients (typical in modern machine learning), where first-order methods are markedly more sample- and compute-efficient; a fair comparison would require different benchmarks, budgets, and stopping criteria and is out of scope.

Stochastic Gradient Descent (SGD) [14] was the primary go-to optimization algorithm for neural networks. However, it featured linear convergence [3, 15], which makes its application to large scale problems challenging [16, 17]. To improve its performance, some versions of SGD rely on "brute force" techniques, such as parallelization [18] and distribution [19], which make better use of the available hardware, but do not increase efficiency (here intended as the count of steps to be performed to converge, times the time required to compute a single step). To enhance the convergence rate, a multitude of algorithmic innovations emerged [5], including Nesterov momentum [20], warm-up heuristics [3], forgetting factors [21, 22], and decoupled weight decay [23–25].

The literature has witnessed a substantial proliferation of techniques, particularly since the latter half of the 2010 s. Recent proposals have been built upon combinations of earlier methods, thereby paving the way for a new generation of hybrid optimization algorithms. Consequent to the growth in number of the available options, the selection of the most suitable optimization algorithm for a specific task has become a challenging endeavor.

Problem statement

The existing literature primarily focuses on the theoretical aspects of optimization algorithms [4, 26]. However, real-world applications present several practical challenges [27–30] that are often overlooked.

Classical theoretical analysis typically relies on convergence rate [31] or generalization performance [32] as key performance metrics. In practice, however, convergence behavior can vary significantly depending on the specific conditions of the optimization problem. A notable example is the Rosenbrock function [33, 34], a standard benchmark for optimization algorithms. The existing literature using the Rosenbrock function for performance evaluation usually limits analysis to simple numerical metrics, such as the step count or the final accuracy [33, 35]. This approach ignores qualitative behaviors: for instance, algorithms may struggle from high-frequency "zig-zagging" [36] or demonstrate "looping" typical of momentum methods [37]. In fact, many algorithms struggle to

converge asymptotically, often getting trapped in oscillations near the solution, exhibiting instability, or progressing extremely slowly from the initial point.

A well-designed metric for assessing optimization performance should therefore account for both convergence rate and the quality of the final solution reached before the algorithm terminates.

While recent works have analyzed specific dynamic phenomena, such as the "Edge of Stability" (where algorithms oscillate rather than settle [38]), or the specific instability patterns of adaptive methods [39], to our knowledge, no work provides a systematic classification of these convergence behaviors within a unified experimental setup. Consequently, current benchmarks are not able to distinguish between algorithms that converge smoothly and those that oscillate precariously near the solution.

In real-world scenarios, optimization algorithms are typically compared against a limited set of alternatives, with testbeds varying across studies. Recently, training a ResNet-18 neural network on the CIFAR-10 dataset [17] has become one of the most widely used benchmarks. However, this approach is computationally expensive for hyperparameter optimization, often forcing researchers to rely on default hyperparameter settings, which can significantly impact the convergence properties of certain algorithms. Additionally, it represents only a specific type of problem complexity, offering little flexibility to adjust difficulty levels.

Alternatively, despite its synthetic nature, the Rosenbrock function provides a more adaptable benchmark. Its complexity can be easily modified, and it remains computationally efficient to evaluate, making it a valuable alternative for studying convergence behavior under controlled conditions.

Relevance to machine learning First-order, derivative-based optimizers are the workhorses of modern deep learning: they scale to billions of parameters with modest memory, tolerate noisy mini-batch gradients, and map well to data-parallel hardware. These properties explain their dominance across computer vision, NLP, and recommendation workloads, and motivate our comparative study focused on ML-relevant behavior.

Contribution This paper makes the following contributions.

- We introduce a *unified notation and algorithmic template* for a broad class of first-order optimisation methods, highlighting their structural relationships and pinpointing the minimal changes that differentiate them.
- We perform a *controlled empirical comparison* of these methods on both a synthetic testbed (Rosenbrock) and a realistic deep-learning workload (ResNet-18 on CIFAR-10).
- We derive and discuss a *taxonomy of convergence behaviours* (for example, fast but unstable, slow but robust, or highly sensitive to learning-rate choices), and relate these patterns to specific design choices in the algorithms.
- We summarise our findings as *concrete practitioner guidelines* for choosing and tuning optimisers under common constraints, such as limited tuning budget, noisy gradients, or prioritisation of stability versus speed.*Structure of the paper* The remainder of this paper is structured as follows: Sect. 2 classifies the approaches under consideration and shows their dependency relations; Sect. 3 provides theoretical background and compares the existing algorithms; Sect. 4 measures the performance of the algorithms under scrutiny; finally, Sect. 5 concludes the paper and outlines recommendations for researchers and practitioners.

2 The landscape

In this section, we provide a bird-eye view of the main optimization algorithms available in the literature. We classify them according to the kind of information about the problem they leverage to speed up computation and we provide a short reconstruction of their history, building a map that will help the reader navigate the upcoming detailed discussion. To navigate through the many acronyms and names, we provide a reference summary in Table 1. To navigate through the notation used in this paper, we provide a reference summary in Table 2.

Figure 1 is provided to visualize the relationships and inspirations between these ideas. The common ancestor of all the algorithms considered in this work is plain Gradient Descent (GD). To tackle the complexity induced

Table 1 Acronym table

Acronym	Definition
AdaBound [40]	Bounded adaptive movement estimation
AdaGrad [41]	Adaptive gradient
Adam [42]	Adaptive movement estimation
AdaMax [42]	Adaptive maximum movement estimation
AdaMod [43]	Adaptive and momental bound
AdamP [44]	Projection-based adaptive movement estimation
AdamW [45]	Weight decay adaptive movement estimation
AggMo [46]	Aggregative momentum
ALR	Adaptive learning rate
AMSgrad [47]	Adaptive maximum squared gradients
ASGD	Averaged stochastic gradient descent
DiffGrad [48]	Difference gradient
DFC [48]	Difference gradient (DiffGrad) friction coefficient
GAN [49]	Generative adversarial network
GD	Gradient descent
LAMB [50]	Layer-wise adaptive moments
LARS [51]	Layer-wise adaptive rate scaling
LR	Learning rate
LSGO	Large-scale global optimization
MadGrad [52]	Momentumized, adaptive, dual averaged gradient
NAdam [53]	Nesterov adaptive movement estimation
NAG [54]	Nesterov accelerated gradient
NovoGrad [55]	Gradient normalization and decoupled weight decay
PI	Proportional-integral
PID [56]	Proportional-integral-derivative
QHAdam [57]	Quasi-hyperbolic adaptive movement estimation
QHM [57]	Quasi-hyperbolic moment
RAdam [58]	Rectified adaptive movement estimation
RMSprop [59]	Root mean square propagation
Rprop [60]	Resilient BackPropagation
SGD	Stochastic gradient descent
SGDP	Projection based stochastic gradient descent
SGDW [61]	Rectified stochastic gradient descent
SWATS [62]	Switches from Adam to SGD
SNV [63]	Synthesized Nesterov variant
TPE	Tree-structured Parzen estimator
HPO	Hyperparameter optimization

by LSGO problems, it was first evolved into SGD, which, using a randomly selected subset of data, improves the convergence rate but loses accuracy. Adaptive Gradient (AdaGrad) improved over SGD through the adoption of Adaptive Learning Rate (ALR), namely a technique that changes the step size during optimization (intuitively, a smaller step size is used to refine the search when the solution is close, while a larger step is adopted to quickly explore the solution space when far away from the solution). ALR increases the convergence rate compared to plain SGD, but it suffers from outdated updates [59]: old or irrelevant information could be used to adjust a parameter during optimization, as no mechanism to discard it is in place. Root Mean Square Propagation (RMSprop) evolves ALR by incorporating a forgetting factor, a technique inspired by Resilient BackPropagation (Rprop) [60] where outdated information is gradually marked as less relevant and progressively forgot through a process of exponential decay, de facto building exponential moving estimations of the mean (first moment). These estimations, however, are biased towards zero, especially at the beginning of the optimization process (when part of the information is defaulted to zero) or when the forgetting factors are small, as those small values enter the average (first statistical moment) [42]. Adam [42] introduces a bias correction mechanism based on uncentered variance (second statistical moment). Further techniques devoted to correcting this bias led to a

Table 2 Notation summary

Symbol	Description
$t \in (0, T)$	Iteration index
T	Number of iterations required to converge
$w^{\langle t \rangle}$	Parameters at iteration *t*
f	Objective function (function to be optimized)
$\nabla f(w^{\langle t \rangle})$	Gradient of *f* at the point of $w^{\langle t \rangle}$
$(w^{\langle t \rangle})^2$	Parameters at iteration *t* in the power of 2
μ_g	Global learning rate
$g^{\langle t \rangle}$	Value of the objective function gradient at iteration *t*
m	First statistical moment
μ	Second statistical moment
$\bar{m},\ \bar{\mu}$	Bias-corrected first and second statistical moments
β	Forgetting factor
$\beta_1,\ \beta_2$	Forgetting factor for the first and second statistical moment
λ	Weight decay parameter
$\Delta w^{\langle t \rangle}$	Finite difference of the parameters in Eq. 4 of Rprop
$\eta^+,\ \eta^-$	Learning rate of the positive and negative gradients in Rprop
ξ	DiffGrad friction coefficient
$\Delta g^{\langle t \rangle}$	Finite difference of the gradient in Eq. 12 of DiffGrad
ϑ	Sign monitor of Yogi
g_{NAG}	Nesterov accelerated gradient [54]
ρ	Moment of saturation for RAdam
K_D	Derivative term of the PID controller
$\mu_l^{\langle t \rangle},\ \mu_u^{\langle t \rangle}$	Lower and upper bounds of the AdaBound learning rate
$\nu_1,\ \nu_2$	Hyperbolic weight of the first and second statistical moment in QHAdam
p	Projection of the moment vector in AdamP
q	Adaptive projection-based moment in AdamP
σ	LARS coefficient
ϕ	Norm function of parameter vector in LAMB
γ_k	Estimation of optimal learning rate for SGD in SWATS
s	Bound of adaptive statistical moment in AdaMod
η	Actual learning rate for bound of adaptive statistical moment in AdaMod
β_3	Forgetting factor for bound of adaptive statistical moment in AdaMod

family of algorithms including Adaptive Maximum Movement Estimation (AdaMax) [42], Adaptive Maximum Squared Gradients (AMSgrad) [47], and Difference Gradient (DiffGrad) [48]: they all add bias correction on top of RMSprop.

Adaptive movement estimation (Adam) family algorithms are suffering from generating excessively large learning rates (LRs), hindering learning [40]. To tackle this, the bounded LR was introduced in bounded adaptive movement estimation (AdaBound), adaptive and momental bound (AdaMod).

The equivalence of L_2 regularization and weight decay regularization breaks down in adaptive gradient algorithms like Adam [45]. To tackle this issue, decoupled weight decay was introduced in weight decay adaptive movement estimation (AdamW) [45], gradient normalization and decoupled weight decay (NovoGrad) [55].

The usage of mini-batch is presented in Yogi [64], layer-wise adaptive rate scaling (LARS) [51], and Layer-wise Adaptive Moments (LAMB) [50]. Aggregating of several sets of hyperparameters presented in aggregative momentum (AggMo) [46]. Quasi-hyperbolic moment presented in quasi-hyperbolic adaptive movement estimation (QHAdam) [57], quasi-hyperbolic moment (QHM). Scale-invariance parameters are presented in projection-based adaptive movement estimation (AdamP) [44]. Aggregation of previously presented algorithms form new algorithms like Switches from Adam to SGD (SWATS) [62] or momentumized adaptive dual-averaged gradient (MadGrad) [52].

 https://doi.org/10.1007/s00521-026-12014-1

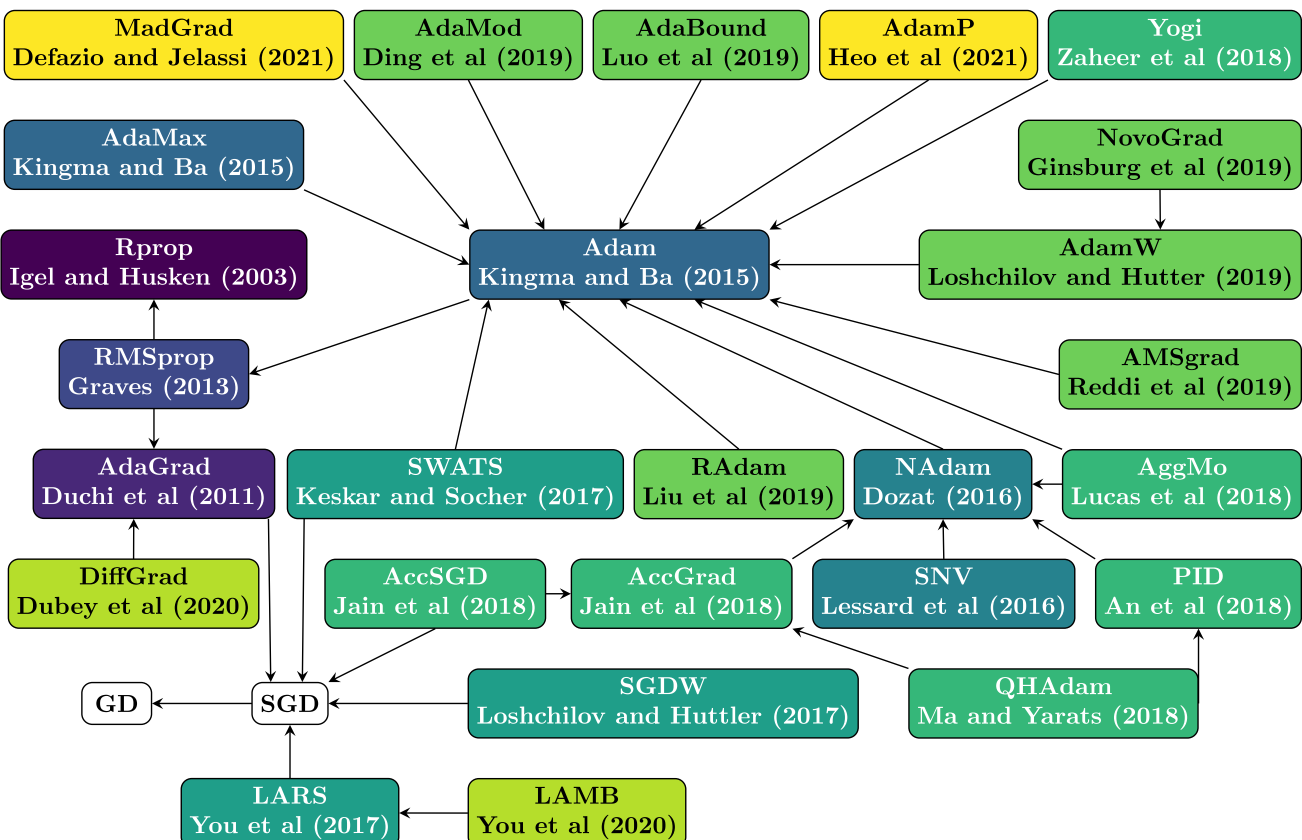

Fig. 1 Relationships among first-order optimization algorithms. Arrows represent "derives from" relations. Node color reflects publication year: darker shades indicate older algorithms

Techniques rooted in SGD may suffer from slow convergence (due, e.g., to oscillations around the solution) in cases where the loss features a particularly complex structure. A technique that may compensate for this issue is the adoption of an additional hyperparameter representing the *momentum* of the gradient search, essentially as a moving exponentially-weighted average of the past gradients, to be used to shift the gradient computation point. The idea was introduced by Nesterov as early as 1983 with Nesterov accelerated gradient (NAG) [65], but it was rarely applied to machine learning problems until recently [66], leading to a new family of algorithms that includes synthesized Nesterov variant (SNV) [63], AccGrad [67], Nesterov adaptive movement estimation (NAdam) [53].

Introducing *momentum* in the gradient search may lead to issues at startup: at the beginning of the optimization process, a value must be set for the *momentum*, but it can hardly get estimated correctly. To tackle this issue, rectified adaptive movement estimation (RAdam) and rectified stochastic gradient descent (SGDW) introduced a warmup restart technique [58], consisting in the execution of some optimization steps with a technique not using momentum, data collection, and subsequent switch to a momentum-equipped algorithm.

Further improvements over momentum-based algorithms come from the fact that, observed from a control theory stance, they are akin to Proportional-Integral (PI) controllers. Introducing a derivative component to the controller, thus building a proportional-integral-derivative (PID) controller, can help reducing oscillations, as it introduces a form of anticipation (feed-forward control) [56].

In the following discussion, we will deepen the analysis of the algorithms, showing how each builds on the previous ones to improve performance.

3 Methods for first-order optimization

In this section, we show how the previously mentioned algorithms build on each other, detailing how they implemented the ideas introduced in the previous section. We start from the SGD, which was the go-to algorithm directly derived from GD.

3.1 Notation

The notation used by the original authors of the algorithms summarized in this work is not uniform. As part of the contribution of this paper, we provide a notation, summarized in Table 2, which captures the commonalities between the algorithms. The notation will be consistently used in the remainder of this paper.

Note on omitting ϵ terms. In many cases, the equations presented in this paper and derived from the original presentation of the algorithms' authors, present divisions in which the denominator contains an ϵ term, where ϵ is a very small positive number. To make the equations more readable and the paper easy to follow, we omit such notation, although its presence is implied. For instance, in the RMSprop algorithm, the term $\frac{\mu_g}{\sqrt{\mu_i^{\langle t \rangle}+\epsilon}}\nabla f(w^{\langle t \rangle})$ will appear in this paper as $\frac{\mu_g}{\sqrt{\mu_i^{\langle t \rangle}}}\nabla f(w^{\langle t \rangle})$, omitting ϵ.

3.2 Basic approach: stochastic gradient descent

One of the simplest optimization algorithms is gradient descent, represented by the following equation:

$$w^{\langle t \rangle} = w^{\langle t-1 \rangle} - \mu_g \nabla f(w^{\langle t-1 \rangle}) \tag{1}$$

The approach uses μ_g as a constant LR for all parameters. The idea is to find the function's steepest increase, determined by the gradient, and proceed in the opposite direction to find the function's minimum value through iterative steps. However, plain gradient descent is nowadays rarely used in performance-critical settings, as it converges linearly [68].

SGD accelerates gradient computation by using a random subset of data instead of the entire dataset. This reduces execution time, especially for large-scale tasks and at the expense of accuracy, but the algorithm still converges linearly.

3.3 Adaptive learning rate: AdaGrad

One way to speed up the computation is to exploit interdependence between parameters. This idea is applied in AdaGrad [41], which introduces adaptive (changing) LR instead of a single constant as μ_g in Eq. 1.

The LR can be represented as a function of the optimization parameters, such as the current value of the gradient or the current iterations count. AdaGrad adapts the LR based on parameter interdependence, increasing it for less interdependent parameters and decreasing it for more interdependent ones. Appropriately designed adaptive LRs can improve the convergence rate [69]. In AdaGrad, adaptivity is achieved by changing the speed at which the search space is explored based on the usage history of gradients. It accumulates squared gradients using a matrix μ as follows:

$$\mu = \sum_{t=1}^{T} (\nabla f(w^{\langle t \rangle}))^2 \tag{2}$$

Each value in this matrix contributes to the adaptive LR through the following equation:

$$w^{\langle t \rangle} = w^{\langle t-1 \rangle} - \frac{\mu_g}{\sqrt{\mu}} \nabla f(w^{\langle t-1 \rangle}) \tag{3}$$

 https://doi.org/10.1007/s00521-026-12014-1

This mechanism enhances convergence, especially in sparse problems with few interdependent variables. Algorithm 1 provides the complete AdaGrad implementation.

```
Input: μ_g: LR; w⟨0⟩: initial parameters; f: loss function
while t = 1 to T do
    g⟨t⟩ ← ∇f(w⟨t−1⟩)
    μ⟨t⟩ ← μ⟨t−1⟩ + (g⟨t⟩)^2
    w⟨t⟩ ← w⟨t−1⟩ − μ_g g⟨t⟩ / sqrt(μ⟨t⟩)
end while
```

Algorithm 1 AdaGrad

A potential drawback of this approach is an influence of outdated updates. Due to cumulative nature of the algorithm, updates perform well at the initial stages, but they keep influencing subsequent steps, even though they are no longer relevant.

3.4 Forgetting factor: Rprop, RMSprop

A possible mitigation to the problem of outdated updates is the introduction of a *forgetting factor*. It is a cut off mechanism that limits the influence of outdated data by detecting sign changes in the partial derivatives of the objective function. More precisely, if the partial derivative with respect to a dimension maintains the same sign over consecutive steps, the step size increases of a value proportional to a parameter η^+; if the sign changes, the step size decreases of a value proportional to a parameter η^- as per the following equation:

$$\Delta w^{\langle t+1\rangle} = \begin{cases} \eta^+ \Delta w_i^{\langle t\rangle} & if \quad \frac{\partial f}{\partial w_i}^{\langle t-1\rangle} \frac{\partial f}{\partial w_i}^{\langle t\rangle} > 0 \\ \eta^- \Delta w_i^{\langle t\rangle} & if \quad \frac{\partial f}{\partial w_i}^{\langle t-1\rangle} \frac{\partial f}{\partial w_i}^{\langle t\rangle} < 0 \\ \Delta w_i^{\langle t\rangle} \end{cases} \tag{4}$$

where $0 < \eta^- < 1 < \eta^+$. An algorithm applying this strategy is Rprop [60].

A similar strategy is proposed in RMSprop [59], leveraging of two consecutive gradient values. The LR is updated iteratively using the following equation:

$$\mu_i^{\langle t\rangle} = \beta \mu_i^{\langle t-1\rangle} + (1-\beta)\left(\frac{\partial f(w^{\langle t-1\rangle})}{\partial w_i}\right)^2 \quad \forall i \tag{5}$$

where $\beta \in (0, 1)$ is the forgetting factor. The larger β, the smaller is the influence of past values; as the algorithm progresses, the influence of initial stages diminishes. With this approach, parameter updates are computed as:

$$w_i^{\langle t\rangle} = w_i^{\langle t-1\rangle} - \frac{\mu_g}{\sqrt{\mu_i^{\langle t\rangle}}} \nabla f(w^{\langle t-1\rangle}) \tag{6}$$

where μ_g is the global component of LR and $\frac{\mu_g}{\sqrt{\mu_i^{\langle t\rangle}}}$ is the adaptive component. Algorithm 2 presents the complete implementation.

```
Input: μ_g: LR; β: forgetting factor; w⟨0⟩: initial parameters; f: loss function
while t = 1 to T do
    g⟨t⟩ ← ∇f(w⟨t−1⟩)
    μ⟨t⟩ ← βμ⟨t−1⟩ + (1 − β)(g⟨t⟩)^2
    w⟨t⟩ ← w⟨t−1⟩ − μ_g g⟨t⟩ / sqrt(μ⟨t⟩)
end while
```

Algorithm 2 RMSprop. Blue text shows differences with respect to AdaGrad

3.5 Statistical moments: Adam, AdaMax, AMSgrad, DiffGrad, Yogi

Carefully looking at Eq. 5 reveals that the LR update strategy is based on a **raw** (or **uncentered**) second statistical moment. In principle, however, other moments (mean, skewness, kurtosis, etc.) could be used to adapt the

LR, incorporating more information on the evolution of the parameter space exploration. This idea is the basis for the family of algorithms that we discuss in this section. Since the family is extensive, we summarize the key modifications introduced by each algorithm with respect to the original Adam, which we discuss first, in Table 3.

3.5.1 Adam and AdaMax

The progenitor algorithm is Adam [42], which calculates the *raw statistical moment* in a way akin to the one shown for RMSprop in Eq. 5:

$$\mu_i^{\langle t\rangle} = \beta_2 \mu_i^{\langle t-1\rangle} + (1-\beta_2)\left(\nabla f(w_i^{\langle t-1\rangle})\right)^2 \quad \forall i \tag{7}$$

Here, β_2 serves as the forgetting factor for the second moment; in addition, Adam also computes the first moment, which is the mean of the gradients:

$$m_i^{\langle t\rangle} = \beta_1 m_i^{\langle t-1\rangle} + (1-\beta_1)\nabla f(w_i^{\langle t-1\rangle}) \quad \forall i \tag{8}$$

In this equation, β_1 corresponds to the forgetting factor for gradient averaging. Typical values for these factors are $\beta_1 \approx 0.9$ and $\beta_2 \approx 0.999$ [70]. Additionally, bias correction is applied to the moments (which we denotate with an overbar), accounting for the influence of the forgetting factors over time:

$$\bar{\mu}_i^{\langle t\rangle} = \frac{\mu_i^{\langle t\rangle}}{1-\beta_2^t} \qquad \bar{m}_i^{\langle t\rangle} = \frac{m_i^{\langle t\rangle}}{1-\beta_1^t} \quad \forall i \tag{9}$$

Finally, the parameter updates are performed as follows:

$$w_i^{\langle t\rangle} = w_i^{\langle t-1\rangle} - \frac{\mu_g \bar{m}_i^{\langle t\rangle}}{\sqrt{\bar{\mu}_i^{\langle t\rangle}}} \quad \forall i \tag{10}$$

The complete implementation of Adam is depicted in Algorithm 3.

Table 3 Adam family variants: key differences with respect to Adam, summarizing sections from Sect. 3.5 to Sect. 3.13

Algorithm	Key modification	Intended effect
Adam	EMA of first and second moments with bias correction (Eqs. 7 and 10 and algorithm 3)	Stable, fast early progress under noisy gradients
AdamW	Decouple weight decay from the adaptive gradient update (Eq. (27) and algorithm 12)	Better generalisation vs. coupled L2; clearer regularisation control
AdaBound	Clip the per-parameter learning rate to dynamic bounds that converge to a target step size (Algorithm 10)	Avoids LR explosion; asymptotically approaches SGD-like behaviour
AdaMax	Replace second-moment EMA with an exponential moving maximum of past gradients	Tighter control of scaling
AdaMod	Track an EMA of the actual learning rate and cap it with a secondary EMA-based bound (Algorithm 11)	Caps aggressive steps adaptively; stabilizes early training
AMSGrad	Use a nondecreasing second-moment tracker in the denominator (Algorithm 4)	Prevents step-size growth; better nonconvex convergence
DiffGrad	Scale the step by a sigmoidal factor of the change in gradient magnitude (Eq. 12 and algorithm 5)	Smaller step sizes as gradients stabilize
Yogi	Sign-controlled updating of the second-moment estimate (Algorithm 6)	Reduces oscillations; improves robustness
NAdam	Add Nesterov momentum to Adam (Eqs. 15–17 and algorithm 7)	Smoother trajectories; mitigates overshoot
RAdam	When variance is well estimated, use it to rectify the adaptive step (Eq. 18 and algorithm 8)	Safe early-phase steps; reduces manual warmup
QHAdam	Mixing of raw gradient and EMA in both numerator and denominator (Eqs. 31 and 32 and algorithm 14)	Tunable speed–stability trade-off
AdamP	Remove the radial component when the gradient is nearly orthogonal to the weight vector (Eqs. 33 and 34, and algorithm 15)	Better norm growth and scale-invariant behaviour

Input: μ_g: LR; β_1, β_2: forgetting factors; $w^{\langle 0\rangle}$: initial parameters; f: loss function
while $t = 1$ to T **do**
$g^{\langle t\rangle} \leftarrow \nabla f(w^{\langle t-1\rangle})$
$\mu^{\langle t\rangle} \leftarrow \beta_2 \mu^{\langle t-1\rangle} + (1-\beta_2)(g^{\langle t\rangle})^2$
$m^{\langle t\rangle} \leftarrow \beta_1 m^{\langle t-1\rangle} + (1-\beta_1) g^{\langle t\rangle}$
$\bar{\mu}^{\langle t\rangle} \leftarrow \frac{\mu^{\langle t\rangle}}{1-\beta_2^t}$
$\bar{m}^{\langle t\rangle} \leftarrow \frac{m^{\langle t\rangle}}{1-\beta_1^t}$
$w^{\langle t\rangle} \leftarrow w^{\langle t-1\rangle} - \frac{\mu_g \bar{m}^{\langle t\rangle}}{\sqrt{\bar{\mu}_i^{\langle t\rangle}}}$
end while

Algorithm 3 Adam. Blue text shows differences with respect to RMSprop

AdaMax [42] is a variant of Adam which employs the infinite norm of the gradient, selecting the largest gradient component for all parameters:

$$\mu_i^{\langle t\rangle} = \beta_2 \mu_i^{\langle t-1\rangle} + (1-\beta_2^p) \left| \frac{\partial f}{\partial w_i} \right|_p \quad p \to \infty \quad \forall i \tag{11}$$

3.5.2 AMSgrad

In various applications, such as learning with large output spaces, it has been observed that algorithms of the Adam family may struggle to converge to an optimal solution or critical point in nonconvex settings (e.g., neural networks) [47]. This limitation arises from the Exponential Moving Average (EMA) employed by these algorithms. To address this issue, an additional variant called AMSgrad [47] has been proposed, which incorporates "long-term memory" of past gradients. This modification not only resolves convergence issues but often improves empirical performance [47].

Input: μ_g: LR; β_1, β_2: forgetting factors; $w^{\langle 0\rangle}$: initial parameters; f: loss function
$\bar{m}^{\langle 0\rangle} \leftarrow 0$
while $t = 1$ to T **do**
$g^{\langle t\rangle} \leftarrow \nabla f(w^{\langle t-1\rangle})$
$\mu^{\langle t\rangle} \leftarrow \beta_2 \mu^{\langle t-1\rangle} + (1-\beta_2)\left(g^{\langle t\rangle}\right)^2$
$m^{\langle t\rangle} \leftarrow \beta_1 m^{\langle t-1\rangle} + (1-\beta_1) g^{\langle t\rangle}$
$\bar{\mu}^{\langle t\rangle} \leftarrow \frac{\mu^{\langle t\rangle}}{1-\beta_2^t}$
$\bar{m}^{\langle t\rangle} \leftarrow max(\bar{m}^{\langle t-1\rangle}, m^{\langle t\rangle})$
$w^{\langle t\rangle} \leftarrow w^{\langle t-1\rangle} - \frac{\mu_g \bar{m}^{\langle t\rangle}}{\sqrt{\bar{\mu}^{\langle t\rangle}}}$
end while

Algorithm 4 AMSgrad. Blue text shows differences with respect to Adam

3.5.3 DiffGrad

Further improvements to the convergence rate could be achieved by considering how the gradient is changing locally. Although methods like AdaGrad, RMSprop, and Adam partly capture local gradient changes, they are not specifically meant to do so. Opposedly, the DiffGrad optimizer [48] explicitly focuses on the difference between present and immediate past gradients to capture local changes with greater detail. DiffGrad adjusts the step size for each parameter based on the speed of gradient changes, taking larger steps for faster changes and smaller steps for slower changes. The DiffGrad optimizer utilizes the finite difference of gradient $\Delta g^{\langle t\rangle}$:

$$\Delta g^{\langle t\rangle} = g^{\langle t\rangle} - g^{\langle t-1\rangle} \tag{12}$$

to calculate a DiffGrad Friction Coefficient (DFC) ξ using a sigmoid function:

$$\xi = \frac{1}{1 + e^{-|\Delta g^{\langle t \rangle}|}} \quad (13)$$

The DFC imposes more friction for slower gradient changes and less friction for larger changes during parameter updates.

$$w^{\langle t \rangle} = w^{\langle t-1 \rangle} - \frac{\mu_g \xi \bar{m}^{\langle t \rangle}}{\sqrt{\bar{\mu}^{\langle t \rangle}}} \quad (14)$$

The complete implementation of DiffGrad is presented in Algorithm 5.

Input: μ_g: LR; β_1, β_2: forgetting factors; $w^{\langle 0 \rangle}$: initial parameters; f: loss function
while $t = 1$ to T **do**
$g^{\langle t \rangle} \leftarrow \nabla f(w^{\langle t-1 \rangle})$
$\mu^{\langle t \rangle} \leftarrow \beta_2 \mu^{\langle t-1 \rangle} + (1 - \beta_2)(g^{\langle t \rangle})^2$
$m^{\langle t \rangle} \leftarrow \beta_1 m^{\langle t-1 \rangle} + (1 - \beta_1) g^{\langle t \rangle}$
$\bar{\mu}^{\langle t \rangle} \leftarrow \frac{\mu^{\langle t \rangle}}{1-\beta_2^t}$
$\bar{m}^{\langle t \rangle} \leftarrow \frac{m^{\langle t \rangle}}{1-\beta_1^t}$
$\xi \leftarrow \frac{1}{1+exp(-|g^{\langle t-1 \rangle} - g^{\langle t \rangle}|)}$
$w^{\langle t \rangle} \leftarrow w^{\langle t-1 \rangle} - \frac{\mu_g \xi \bar{m}^{\langle t \rangle}}{\sqrt{\bar{\mu}^{\langle t \rangle}}}$
end while

Algorithm 5 DiffGrad. Blue text shows differences with respect to Adam

3.5.4 Yogi

Adaptive gradient methods like RMSprop, Adam, and AdaDelta [71] are widely used in deep learning for optimizing non-convex problems. However, recent studies have shown that these methods can struggle to converge even in simple convex optimization scenarios. As pointed out in [72], the multiplicative nature of statistical moments causes past gradients to be forgotten quickly, especially in settings in which gradients are usually zero. Yogi [64] was introduced to address this issue. The idea behind Yogi is to monitor signs changes of $\mu_i^{\langle t \rangle} - (g^{\langle t \rangle})^2$ and use them to prevent oscillations around the solution. Empirical evaluations on challenging machine learning tasks have shown that Yogi can outperform other methods in the Adam family while offering similar theoretical guarantees for convergence. The complete implementation of Yogi is presented in Algorithm 6.

Input: μ_g: LR; β_1, β_2: forgetting factors; $w^{\langle 0 \rangle}$: initial parameters; f: loss function
while $t = 1$ to T **do**
$g^{\langle t \rangle} \leftarrow \nabla f(w^{\langle t-1 \rangle})$
$\vartheta \leftarrow \text{sign}\left(\mu^{\langle t-1 \rangle} - \left(g^{\langle t \rangle}\right)^2\right)$
$\mu_i^{\langle t \rangle} \leftarrow \mu^{\langle t-1 \rangle} - \vartheta\,(1 - \beta_2)\left(g^{\langle t \rangle}\right)^2$
$m^{\langle t \rangle} \leftarrow \beta_1 m^{\langle t-1 \rangle} + (1 - \beta_1) g^{\langle t \rangle}$
$w^{\langle t \rangle} \leftarrow w^{\langle t-1 \rangle} - \frac{\mu_g m_i^{\langle t \rangle}}{\sqrt{\mu_i^{\langle t \rangle}}}$
end while

Algorithm 6 Yogi. Blue text shows differences with respect to Adam

3.6 Nesterov momentum: NAG, NAdam

Approaches based on momentum rely on the idea that the direction of the past exploration can be used to improve convergence. Thus, instead of computing the gradient at the current point, NAG [54, 73] computes it at a shifted

 Springer

point based on the direction the previous steps were following, thus incorporating a sort of momentum. Intuitively, it is similar to a ball rolling down an irregular slope: thanks to momentum, the ball can overcome small local minima and reach the bottom. Nesterov momentum was developed by Yrii Nesterov in 1983 [65], and although the original algorithm did not perform well in practice, the idea has paved the way for improvements over classic SGD and Adam. The computation of the momentum starts from the Nesterov Accelerated Gradient (g_{NAG}), computed as:

$$g_{NAG} = \nabla f\left(w^{\langle t-1\rangle} + \beta_1 m^{\langle t-1\rangle}\right) \tag{15}$$

Then, the momentum at the current iteration $m^{\langle t\rangle}$ is updated on g_{NAG}:

$$m_i^{\langle t\rangle} = \beta_1 m_i^{\langle t-1\rangle} + \mu_g g_{NAG} \tag{16}$$

and subsequently used to update the parameters:

$$w_i^{\langle t\rangle} = w_i^{\langle t-1\rangle} - m_i^{\langle t\rangle} \tag{17}$$

NAdam [53] is based on this idea, using a decaying mean of the previous gradients as momentum. In NAdam, variables β_1 and μ_g are changed over time in a way that minimizes oscillations around the optimal solution. The complete NAdam algorithm is presented in Algorithm 7. There, we use a formulation of the algorithm that computes the regular gradient and applies momentum transformations afterward to minimize the set of differences with respect to Adam. This form has been shown to be mathematically equivalent to the direct use of the Nesterov gradient [53].

Input: $[\mu_g^{\langle 0\rangle}, \mu_g^{\langle 1\rangle}, ..., \mu_g^{\langle T\rangle}]$: LR; $[\beta_1^{\langle 0\rangle}, \beta_1^{\langle 1\rangle}, ..., \beta_1^{\langle T\rangle}]$, β_2: forgetting factors; $w^{\langle 0\rangle}$: initial parameters; f: loss function

while $t = 1$ to T **do**

$g^{\langle t\rangle} \leftarrow \nabla f(w^{\langle t-1\rangle})$

$\mu^{\langle t\rangle} \leftarrow \beta_2 \mu^{\langle t-1\rangle} + (1-\beta_2)\left(g^{\langle t\rangle}\right)^2$

$m^{\langle t\rangle} \leftarrow \beta_1 m^{\langle t-1\rangle} + (1-\beta_1) g^{\langle t\rangle}$

$\bar{\mu}^{\langle t\rangle} \leftarrow \frac{\mu^{\langle t\rangle}}{1-\beta_2^t}$

$\bar{m}^{\langle t\rangle} \leftarrow \frac{\beta_1^{\langle t\rangle} m^{\langle t\rangle}}{1-\prod_{k=1}^{t+1}\beta_1^k} + \frac{(1-\beta_1^{\langle t\rangle}) g^{\langle t\rangle}}{1-\prod_{k=1}^{t}\beta_1^k}$

$w^{\langle t\rangle} \leftarrow w^{\langle t-1\rangle} - \frac{\mu_g^{\langle t\rangle} \bar{m}_i^{\langle t\rangle}}{\sqrt{\bar{\mu}_i^{\langle t\rangle}}}$

end while

Algorithm 7 NAdam. Blue text shows differences with respect to Adam

3.7 Warm-up heuristic: RAdam, SGDR

Moment-based algorithms dynamically alter the learning rate with a multiplicative factor that depends on the previous iterations of the optimization process (see Eqs. 10 and 16). However, at the beginning of the optimization process, the variance of the moment is problematically large. This issue can be mitigated using a heuristic called *warm-up strategy*. The core of the idea is to begin the optimization process without taking the moment or momentum into account, while still updating it:

$$w^{\langle t\rangle} = w^{\langle t-1\rangle} - \mu_g^{\langle t\rangle} \bar{m}_i^{\langle t\rangle} \tag{18}$$

In a few optimization steps, the moment converges to a stable value (*moment saturation*), and the algorithm can proceed with the learning rate adaptation in place.

RAdam [58] is an adaptation of Adam that implements this idea. It introduces a heuristic to detect moment saturation, based on a parameter ρ initially calculated as:

$$\rho_\infty = \frac{2}{1-\beta_2} - 1 \tag{19}$$

And subsequently updated with:

$$\rho_t = \rho_\infty - \frac{2t\beta_2^t}{1-\beta_2^t} \tag{20}$$

For this parameter, there is emprical evidence [58] that saturation is reached when $\rho > 4$, thus, in RAdam, moments are used to update the LR only when ρ is below this threshold, as shown in Algorithm 8.

```
Input: μ_g: LR; β_1, β_2: forgetting factors; w^⟨0⟩: initial parameters; f: loss function
ρ_∞ ← 2/(1−β_2) − 1
while t = 1 to T do
    g^⟨t⟩ ← ∇f(w^⟨t−1⟩)
    μ^⟨t⟩ ← β_2 μ^⟨t−1⟩ + (1 − β_2) (g^⟨t⟩)^2
    m^⟨t⟩ ← β_1 m^⟨t−1⟩ + (1 − β_1) g^⟨t⟩
    μ̄_i^⟨t⟩ ← μ^⟨t⟩ / (1 − β_2^t)
    m̄_i^⟨t⟩ ← m^⟨t⟩ / (1 − β_1^t)
    ρ_t ← ρ_∞ − 2tβ_2^t / (1 − β_2^t)
    if ρ_t > 4.0 then
        w^⟨t⟩ ← w^⟨t−1⟩ − μ_g m̄_i^⟨t⟩ / sqrt(μ̄_i^⟨t⟩)
    else
        w^⟨t⟩ ← w^⟨t−1⟩ − μ_g m̄_i^⟨t⟩
    end if
end while
```

Algorithm 8 RAdam. Blue text shows differences with respect to Adam

SGDW applies a similar approach to improve the plain SGD, using a combination of warm-up strategies and restart mechanisms [61].

3.8 Addition of a derivative term: PID

From a control theory perspective, SGD is mathematically equivalent to a proportional controller, as it uses the gradient of the loss function to update the parameters. Instead, Adam can be interpreted, using the same lens, as a proportional-integral controller, as it incorporates the accumulation of historical gradients. As in control systems, an integral-proportional controller introduces a form of inertia, which can lead to overshooting and oscillations around the optimal point. These issues can be mitigated by adding a derivative term to the Adam algorithm, thus transforming it into a Proportional-Integral-Derivative controller [56]. This modification adds a damping effect to the optimization dynamics, reducing oscillations. The derivative term is calculated as the difference between the current and previous gradients:

$$\mu^{\langle t\rangle} = \beta\mu^{\langle t-1\rangle} + (1-\beta)\left(g^{\langle t\rangle} - g^{\langle t-1\rangle}\right) \tag{21}$$

While the integral and proportional terms are calculated as in Adam:

$$m^{\langle t\rangle} = \beta m^{\langle t-1\rangle} + \mu_g g^{\langle t\rangle} \tag{22}$$

The PID controller combines the proportional, integral, and differential components for parameters update:

 https://doi.org/10.1007/s00521-026-12014-1

$$w^{\langle t\rangle} = w^{\langle t-1\rangle} - m^{\langle t\rangle} + K_D\mu^{\langle t\rangle} \tag{23}$$

We summarize the complete PID algorithm in Algorithm 9.

Input: μ_g: LR; β: integral factor; K_D: derivative factor; $w^{\langle 0\rangle}$: initial parameters; f: loss function
while $t = 1$ to T **do**
$g^{\langle t\rangle} \leftarrow \nabla f(w^{\langle t-1\rangle})$
$m^{\langle t\rangle} \leftarrow \beta m^{\langle t-1\rangle} + \mu_g g^{\langle t\rangle}$
$\mu^{\langle t\rangle} \leftarrow \beta\mu^{\langle t-1\rangle} + (1-\beta)\left(g^{\langle t\rangle} - g^{\langle t-1\rangle}\right)$
$w^{\langle t\rangle} \leftarrow w^{\langle t-1\rangle} - m^{\langle t\rangle} + K_D\mu^{\langle t\rangle}$
end while

Algorithm 9 PID. Blue text shows differences with respect to Adam

3.9 Learning rate bound: AdaBound, AdaMod

In initial optimization phase algorithms featuring adaptive LR can lead to “gradient explosion”—a rapid growth of the LR [74]. In these cases, limiting such growth (technically, reaching a *LR saturation*) can help achieve better performance. This idea is at the basis of AdaBound, a variant of Adam that progressively restricts the LR adaptation rate, gradually transforming Adam into a classic SGD.

The algorithm uses parameter-specific Learning Rates based on the moment $\mu^{\langle t\rangle}$, which, at every iteration t, are clipped to stay within the interval $\left[\mu_l^{\langle t\rangle}, \mu_u^{\langle t\rangle}\right]$, where $\mu_l^{\langle t\rangle}$ and $\mu_u^{\langle t\rangle}$ are monotonic functions asymptotically converging to a non-negative finite value α_* (typically selected heuristically), starting respectively from 0 and ∞; in other words, $\mu_l^{\langle 0\rangle} = 0$ and $\mu_u^{\langle 0\rangle} = \infty$. Thus, with the progression of the search, the LRs get constrained into progressively smaller intervals, eventually converging to a fixed value α_*. Algorithm 10 shows the differences with Adam.

Input: $[\mu_g^{\langle 0\rangle}, \mu_g^{\langle 1\rangle}, ..., \mu_g^{\langle T\rangle}]$: LR; $[\beta_1^{\langle 0\rangle}, \beta_1^{\langle 1\rangle}, ..., \beta_1^{\langle T\rangle}]$, β_2: forgetting factors; $[\mu_l^{\langle 0\rangle}, \mu_l^{\langle 1\rangle}, ..., \mu_l^{\langle T\rangle}]$, $[\mu_u^{\langle 0\rangle}, \mu_u^{\langle 1\rangle}, ..., \mu_u^{\langle T\rangle}]$: bounds; $w^{\langle 0\rangle}$: initial parameters; f: loss function
while $t = 1$ to T **do**
$g^{\langle t\rangle} \leftarrow \nabla f(w^{\langle t-1\rangle})$
$\mu^{\langle t\rangle} \leftarrow \beta_2\mu^{\langle t-1\rangle} + (1-\beta_2)\left(g^{\langle t\rangle}\right)^2$
$m^{\langle t\rangle} \leftarrow \beta_1^{\langle t\rangle} m^{\langle t-1\rangle} + (1-\beta_1^{\langle t\rangle}) g^{\langle t\rangle}$
$\bar{\mu}^{\langle t\rangle} \leftarrow Clip(\mu_g^{\langle t\rangle}/\sqrt{\mu^{\langle t\rangle}}, \mu_l^{\langle t\rangle}, \mu_u^{\langle t\rangle})/\sqrt{t}$
$w^{\langle t\rangle} \leftarrow w^{\langle t-1\rangle} - \bar{\mu}^{\langle t\rangle} m^{\langle t\rangle}$
end while

Algorithm 10 AdaBound. Blue text shows differences with respect to Adam

AdaMod [43] uses a similar approach to limit the maximum LR, but based on the first statistical moment. First, the algorithm computes the actual LR η, similar to Adam’s Eq. 10:

$$\eta_i^{\langle t\rangle} = \frac{\mu_g^{\langle t\rangle}}{\sqrt{\bar{\mu}_i^{\langle t\rangle}}} \tag{24}$$

Then, the actual LR accumulates into the statistical moment $s^{\langle t\rangle}$ of the LR with the forgetting factor β_3:

$$s^{\langle t\rangle} = \beta_3 s^{\langle t-1\rangle} + (1-\beta_3)\eta^{\langle t\rangle} \tag{25}$$

The resulting moment $s^{\langle t\rangle}$ clips the actual LR $\eta^{\langle t\rangle}$, which is then used to update the parameters:

$$w^{\langle t\rangle} = w^{\langle t-1\rangle} - min(s^{\langle t\rangle}, \eta^{\langle t\rangle})\bar{\mu}_i^{\langle t\rangle} \tag{26}$$

Springer

Thus, in AdaMod, the upper bound of the LR is computed adaptively, and its evolution is controlled by a single constant hyper-parameter β_3. The overall AdaMod algorithm is summarized in Algorithm 11.

Input: $[\mu_g^{\langle 0\rangle}, \mu_g^{\langle 1\rangle}, ..., \mu_g^{\langle T\rangle}]$: LR; β_1, β_2, β_3: forgetting factors; $w^{\langle 0\rangle}$: initial parameters; f: loss function

while $t = 1$ to T **do**
$g^{\langle t\rangle} \leftarrow \nabla f(w^{\langle t-1\rangle})$
$\mu^{\langle t\rangle} \leftarrow \beta_2 \mu^{\langle t-1\rangle} + (1-\beta_2)\left(g^{\langle t\rangle}\right)^2$
$m^{\langle t\rangle} \leftarrow \beta_1^{\langle t\rangle} m^{\langle t-1\rangle} + (1-\beta_1) g^{\langle t\rangle}$
$\bar{\mu}_i^{\langle t\rangle} \leftarrow \frac{\mu^{\langle t\rangle}}{1-\beta_2^t}$
$\bar{m}_i^{\langle t\rangle} \leftarrow \frac{m^{\langle t\rangle}}{1-\beta_1^t}$
$\eta^{\langle t\rangle} \leftarrow \mu_g^{\langle t\rangle} / \sqrt{\bar{\mu}^{\langle t\rangle}}$
$s^{\langle t\rangle} \leftarrow \beta_3^{\langle t\rangle} s^{\langle t-1\rangle} + (1-\beta_3)\eta^{\langle t\rangle}$
$\bar{\eta}^{\langle t\rangle} \leftarrow min(s^{\langle t\rangle}, \eta^{\langle t\rangle})$
$w^{\langle t\rangle} \leftarrow w^{\langle t-1\rangle} - \bar{\eta}^{\langle t\rangle} \bar{\mu}_i^{\langle t\rangle}$
end while

Algorithm 11 AdaMod. Blue text shows differences with respect to Adam

3.10 Decoupled weight decay: AdamW

In plain Stochastic Gradient Descent,, "weight decay" and "L_2 regularization" are mathematically equivalent [45]. However, for adaptive optimization algorithms, like Adam and its descendants, the two concepts become different, and such difference may have led to poorer generalization performance in Adam compared to SGD [75, 76]. We notice that, on this topic, there is often confusion in the literature, with the two terms often used interchangeably [77, 78]. The existence of a family of successful algorithms (including Adam, RMSprop, and AdaGrad) using *coupled* weight decay may have contributed to this confusion.

In *coupled* weight decay, a single parameter (μ_g) is used to control the convergence rate. In contrast, *decoupled weight decay* [79] uses two different parameters (λ and μ_g):

$$w_i^{\langle t\rangle} = (1-\mu_g\lambda) w_i^{\langle t-1\rangle} - \frac{\mu_g \bar{m}_i^{\langle t\rangle}}{\sqrt{\bar{\mu}_i^{\langle t\rangle}}} \tag{27}$$

Where λ *is the weight decay factor* - an additional hyperparameter that controls the decay rate. AdamW [45], which we summarized in Algorithm 12, implements this idea. The authors show that treating "weight decay" and "L_2 regularization" as separate concepts improves the generalization performance with respect to plain Adam [45].

 https://doi.org/10.1007/s00521-026-12014-1

Input: μ_g: LR; β_1, β_2: forgetting factors; $w^{\langle 0\rangle}$: initial parameters; f: loss function; λ: weight decay factor
while $t = 1$ to T **do**
  for $\forall i$ **do**
    $g^{\langle t\rangle} \leftarrow \nabla f(w^{\langle t-1\rangle})$
    $\mu^{\langle t\rangle} \leftarrow \beta_2 \mu^{\langle t-1\rangle} + (1-\beta_2)\left(g^{\langle t\rangle}\right)^2$
    $m^{\langle t\rangle} \leftarrow \beta_1 m^{\langle t-1\rangle} + (1-\beta_1) g^{\langle t\rangle}$
    $\bar{\mu}_i^{\langle t\rangle} \leftarrow \frac{\mu^{\langle t\rangle}}{1-\beta_2^t}$
    $\bar{m}_i^{\langle t\rangle} \leftarrow \frac{m^{\langle t\rangle}}{1-\beta_1^t}$
    $w_i^{\langle t\rangle} \leftarrow w_i^{\langle t-1\rangle} - \frac{\mu_g \bar{m}_i^{\langle t\rangle}}{\sqrt{\bar{\mu}_i^{\langle t\rangle}}} - \mu_g \lambda w^{\langle t-1\rangle}$
  end for
end while

Algorithm 12 AdamW. Blue text shows differences with respect to Adam

3.11 Multiple velocity vector: AggMo

Moment-based optimizers like Adam require a forgetting factor (β) to balance between recent and past gradient information. Achieving optimal performance is, in many cases, subject to an appropriate selection of β: in other words, hyperparameter tuning is often necessary. AggMo [46], whose implementation is depicted in Algorithm 13, is an approach that works around this limitation by using multiple forgetting factors at once:

$$\boldsymbol{\beta} = [\beta_1, \beta_2, ..., \beta_K]^T \in \mathbb{R}^K \tag{28}$$

Based on this vector of forgetting factors, the algorithm computes a *multiple velocity vector* by element-wise multiplication of the previous velocity vector and the vector of forgetting factors:

$$\boldsymbol{m}^{\langle t\rangle} = \begin{bmatrix} m_1^{\langle t\rangle} \\ m_2^{\langle t\rangle} \\ \ldots \\ m_K^{\langle t\rangle} \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \ldots \\ \beta_K \end{bmatrix} \odot \begin{bmatrix} m_1^{\langle t-1\rangle} \\ m_2^{\langle t-1\rangle} \\ \ldots \\ m_K^{\langle t-1\rangle} \end{bmatrix} - \nabla f(w^{\langle t-1\rangle}) \tag{29}$$

Then, the mean of such multiple velocity vector is used to update the parameters:

$$w^{\langle t\rangle} = w^{\langle t-1\rangle} - \frac{\mu_g}{K} \sum_{i=1}^{K} m_i^{\langle t\rangle} \tag{30}$$

On average, AggMo achieves better performance than instances of Adam in which the forgetting factor is picked randomly [46].

Input: μ_g: LR; $[\beta_0, \beta_1, ..., \beta_K]$: forgetting factors; $w^{\langle 0\rangle}$: initial parameters; f: loss function
while $t = 1$ to T **do**
  $g^{\langle t\rangle} \leftarrow \nabla f(w^{\langle t-1\rangle})$
  for $\forall i$ **do**
    $m_i^{\langle t\rangle} \leftarrow \beta^{\langle t\rangle} m_i^{\langle t-1\rangle} - g^{\langle t\rangle}$
  end for
  $w^{\langle t\rangle} \leftarrow w^{\langle t-1\rangle} - \frac{\mu_g}{K} \sum_{i=1}^{K} m_i^{\langle t\rangle}$
end while

Algorithm 13 AggMo. Blue text shows differences with respect to Adam

3.12 Quasi-hyperbolic moment: QHAdam

Lower variance is beneficial for the stability of the optimization process, but it may cause slower convergence rate. Unfortunately, the usage of a single forgetting factor β can lead to moment's variance decrease with increasing values of β. The influence of variance reduction onto the convergence rate can be mitigated by introducing a new hyperparameter ν that weights the moment. The goal is to balance the influence of the moment and the gradient itself, while leaving the moment accumulation unchanged. The introduction of such weight is beneficial in term of finer control over the relationship between the variance of the moment and the convergence rate.

Other Adam's family algorithms were able to control only the variance of the moment using μ_g, while the introduction of ν can be used as an additional degree of freedom to control the convergence rate:

$$w^{\langle t\rangle} = w^{\langle t-1\rangle} - \mu_g((1-\nu)g^{\langle t\rangle} + \nu\bar{m}_i^{\langle t\rangle}) \tag{31}$$

where $\nu \in \mathcal{R}$ and $0 < \nu < 1$.

QHAdam [57] proposes a new parameters update rule for Adam algorithm, shown in Eq. 31. The usage of the new hyperparameter ν for weighting the moment is highlighted in blue, and is referred to as Quasi-hyperbolic Moment.

In QHAdam, the Quasi-hyperbolic Momen is used for both the first and the second moment, using two different hyperparameters (ν_1 and ν_2 respectively).

The resulting update rule is the same proposed in Adam, firstly introduced in Eq. 10, but Quasi-hyperbolic Moment is used for both moments instead of the pure moments:

$$w^{\langle t\rangle} \leftarrow w^{\langle t-1\rangle} - \mu_g \frac{(1-\nu_1)g^{\langle t\rangle} + \nu_1\bar{m}_i^{\langle t\rangle}}{\sqrt{(1-\nu_2)(g^{\langle t\rangle})^2 + \nu_2\bar{\mu}^{\langle t\rangle}}} \tag{32}$$

The final algorithm QHAdam is shown Algorithm 14.

Input: μ_g: LR; ν_1, ν_2: immediate discount factors; β_1, β_2: forgetting factors; $w^{\langle 0\rangle}$: initial parameters; f: loss function

while $t = 1$ to T **do**

$g^{\langle t\rangle} \leftarrow \nabla f(w^{\langle t-1\rangle})$

$\mu^{\langle t\rangle} \leftarrow \beta_2\mu^{\langle t-1\rangle} + (1-\beta_2)(g^{\langle t\rangle})^2$

$m^{\langle t\rangle} \leftarrow \beta_1 m^{\langle t-1\rangle} + (1-\beta_1)g^{\langle t\rangle}$

$\bar{\mu}_i^{\langle t\rangle} \leftarrow \frac{\mu^{\langle t\rangle}}{1-\beta_2^t}$

$\bar{m}_i^{\langle t\rangle} \leftarrow \frac{m^{\langle t\rangle}}{1-\beta_1^t}$

$w^{\langle t\rangle} \leftarrow w^{\langle t-1\rangle} - \mu_g \frac{(1-\nu_1)g^{\langle t\rangle} + \nu_1\bar{m}_i^{\langle t\rangle}}{\sqrt{(1-\nu_2)(g^{\langle t\rangle})^2 + \nu_2\bar{\mu}^{\langle t\rangle}}}$

end while

Algorithm 14 QHAdam. Blue text shows differences with respect to Adam

3.13 Projection based: AdamP

Generally, momentum-based algorithms show better converge rate compared to algorithms of previous generations [44]. However, including the momentum as part of an optimization algorithm can lead to performance issues caused by the excessive growth of the LR, as discussed in Sect. 3.9 with AdaBound and AdaMod algorithms. AdaBound mitigates the problem by bounding the LR while AdamP [44] tries to solve the same problem differently. The idea is to project geometrically the moment vector m onto the parameter vector w. More specifically, this projection eliminates the component of the moment vector that is perpendicular to the parameter vector (the radial component). By removing this radial component, we maintain the direction of the moment vector unchanged, while ensuring that any adjustments made align with the parameter space, enhancing efficiency and stability in the process. This approach also helps to address the previously discussed issue of learning rate

 https://doi.org/10.1007/s00521-026-12014-1

explosion, thus preventing the moment from growing uncontrollably in directions orthogonal to the parameter vector. The excessive growth of the moment has been empirically found [44] to trigger a specific condition:

$$\frac{|(w^{\langle t-1\rangle})^T \cdot g^{\langle t\rangle}|}{||w^{\langle t-1\rangle}||\ ||g^{\langle t\rangle}||} < \frac{0.1}{\sqrt{dim(w^{\langle t-1\rangle})}} \tag{33}$$

When the condition is triggered, the moment is replaced with the projection:

$$q^{\langle t\rangle} = \begin{cases} p^{\langle t\rangle} - (w^{\langle t-1\rangle} p^{\langle t\rangle}) w^{\langle t-1\rangle} & \text{if Eq. (33) holds} \\ p^{\langle t\rangle} & \text{otherwise} \end{cases} \tag{34}$$

where p is a projection of the moment vector m onto the parameter vector w.

The complete implementation of AdamP is shown in Algorithm 15.

Input: μ_g: LR; β_1, β_2: forgetting factors; $w^{\langle 0\rangle}$: initial parameters; f: loss function
while $t = 1$ to T **do**
  $g^{\langle t\rangle} \leftarrow \nabla f(w^{\langle t-1\rangle})$
  $\mu^{\langle t\rangle} \leftarrow \beta_2 \mu^{\langle t-1\rangle} + (1-\beta_2)\left(g^{\langle t\rangle}\right)^2$
  $m^{\langle t\rangle} \leftarrow \beta_1 m^{\langle t-1\rangle} + (1-\beta_1) g^{\langle t\rangle}$
  $p^{\langle t\rangle} \leftarrow m^{\langle t\rangle} / \sqrt{\mu^{\langle t\rangle}}$
  $q^{\langle t\rangle} \leftarrow$ Equation (34)
  $w^{\langle t\rangle} \leftarrow w^{\langle t-1\rangle} - \mu_g q^{\langle t\rangle}$
end while

Algorithm 15 AdamP. Blue text shows differences with respect to Adam

3.14 Layer-wise adaptivity: NovoGrad, LARS, LAMB

A common usage of algorithms belonging to the Adam's family is weight optimization in Neural Networks (NNs) [42, 58], where artificial neurons are arranged in collections named "layers", with each layer serving a specific role in the learning process. Although "layer" is a term specific to NNs, it can be generalized to indicate groups of parameters with shared characteristics, such as interdependencies or other common features. In problems where parameters are organized in layers, the problem structure can be exploited to improve the optimization process: parameters belonging to the same layer can be grouped and normalized/optimized together, independently of others.

This is the idea behind NovoGrad [55]: instead of calculating a single gradient (and related moments), NovoGrad processes each layer's components separately. Parameters are first grouped by layer, then, each layer is normalized independently of the others. Without group-wise normalization, some layers could get optimized too quickly with respect to others, as the dominant component of one or few layers can assume much larger values than any component of other layers. Instead, with per-layer normalization, all layers get optimized at a similar pace, as their dominant component gets weighted equally to the dominant component of each other layer. Figure 2 provides a graphical representation of layer-based normalization.

In Algorithm 16 we compare NovoGrad with Adam.

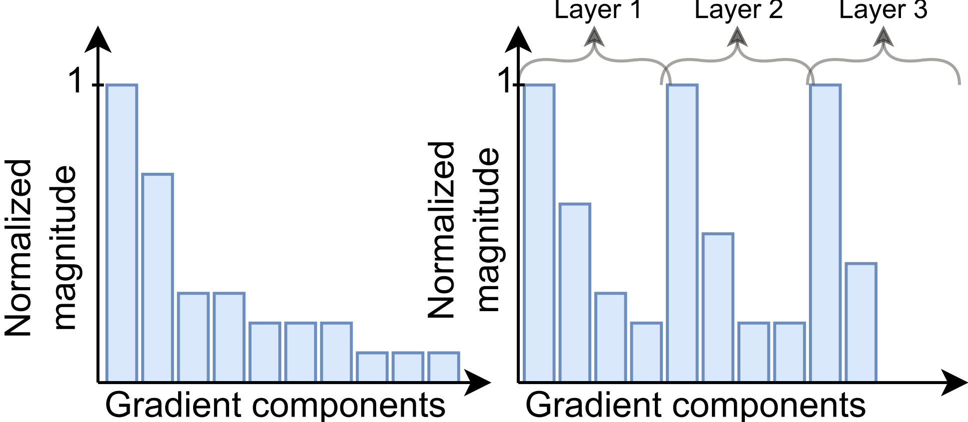

Fig. 2 Schematic illustration of the per-layer normalization (on the right) in NovoGrad with respect to the standard normalization (on the left)

Input: μ_g: LR; β_1, β_2: forgetting factors; $w^{\langle 0\rangle}$: initial parameters; f: loss function; λ: weight decay

while $t = 1$ to T **do**
  for each layer i **do**
    $g_i^{\langle t\rangle} \leftarrow \nabla f(w_i^{\langle t-1\rangle})$
    $\mu_i^{\langle t\rangle} \leftarrow \beta_2 \mu_i^{\langle t-1\rangle} + (1-\beta_2)||g_i^{\langle t\rangle}||^2$
    $m_i^{\langle t\rangle} \leftarrow \beta_1 m_i^{\langle t-1\rangle} + \frac{g^{\langle t\rangle}}{\sqrt{\mu^{\langle t-1\rangle}}} + \lambda w_i^{\langle t\rangle}$
    $w_i^{\langle t\rangle} \leftarrow w_i^{\langle t-1\rangle} - \mu_g m_i^{\langle t\rangle}$
  end for
end while

Algorithm 16 NovoGrad. Blue text shows differences with respect to Adam. Layer indices are denoted with *i*

Overall, NovoGrad shows similar or better performance than SGD, Adam, and AdamW, retaining robustness to weight initialization and LR changes [55].

The idea of organizing parameters into layers is also leveraged by LAMB [50] and LARS [51]. LAMB uses the same update rule of AdamW (cf. Algorithm 12), but applies it to each layer independently (similarly to NovoGrad), thereby introducing normalization to limit the growth of the LR. To do so, LAMB incorporates a hyperparameter which we denote with ϕ, representing a function that returns the norm of the parameters vector: $\phi(||w^{\langle t\rangle}||)$. The resulting update rule aims to keep the norm of the update rule proportional to the norm of the parameters. This change typically leads to faster convergence in deep neural networks [50]. LAMB's implementation is shown in Algorithm 17.

Input: μ_g: LR; β_1, β_2: forgetting factors; $w^{\langle 0\rangle}$: initial parameters; f: loss function; ϕ: scaling function; λ: weight decay;
while $t = 1$ to T **do**
$g^{\langle t\rangle} \leftarrow \nabla f(w^{\langle t-1\rangle})$
$\mu^{\langle t\rangle} \leftarrow \beta_2 \mu^{\langle t-1\rangle} + (1-\beta_2)(g^{\langle t\rangle})^2$
$m^{\langle t\rangle} \leftarrow \beta_1 m^{\langle t-1\rangle} + (1-\beta_1) g^{\langle t\rangle}$
$\bar{\mu}^{\langle t\rangle} \leftarrow \frac{\mu^{\langle t\rangle}}{1-\beta_2^t}$
$\bar{m}^{\langle t\rangle} \leftarrow \frac{m^{\langle t\rangle}}{1-\beta_1^t}$
$r \leftarrow \frac{\bar{m}^{\langle t\rangle}}{\sqrt{\bar{\mu}^{\langle t\rangle}}}$
for each layer i **do**
$w_i^{\langle t\rangle} \leftarrow w_i^{\langle t-1\rangle} - \mu_g \phi(||w_i^{\langle t-1\rangle}||) \frac{(r_i + \lambda w_i^{\langle t-1\rangle})}{||r_i + \lambda w_i^{\langle t-1\rangle}||}$
end for
end while

Algorithm 17 LAMB. Blue text shows differences with respect to Adam. Layer indices are denoted with *i*

When optimizing large NNs, most approaches use large learning rates to speed up convergence. However, large LRs can lead to divergence, especially in the initial phase of the optimization process (cf. Sect. 3.7). Therefore, LARS was designed to optimize weight updates in large NNs, addressing the problem by computing LRs independently for each layer, adjusting it to the specific layer's characteristics. The LR μ_i for layer *i* is calculated at every step based on the ratio between parameters and gradients norms, as shown in Eq. 35.

$$\mu_i = \sigma \cdot {}^{||w_i||}/_{||g_i||} \tag{35}$$

For large gradient values, the LR is reduced to prevent gradient explosion. Conversely, for small gradients, the LR is increased to accelerate the optimization process in flat regions of the cost function landscape. Additionally, the algorithm introduces a coefficient σ ($0 < \sigma < 1$) which provides an estimate of the probability the LR will change during one update, acting as a weight for the LR update. While NovoGrad and LAMB are part of Adam's family, LARS uses SGD as its base algorithm, resulting in Algorithm 18.

Input: $\mu_g^{\langle 0\rangle}$: LR; β: forgetting factor; σ: LARS coefficient; $w^{\langle 0\rangle}$: initial parameters; f: loss function; λ: moment;
while $t = 1$ to T **do**
$g^{\langle t\rangle} \leftarrow \nabla f(w^{\langle t-1\rangle})$
$\mu_g^{\langle t\rangle} \leftarrow \mu_g^{\langle t-1\rangle} \cdot (1 - \frac{t}{T})^2$
for each layer i **do**
$\mu_i^{\langle t\rangle} \leftarrow \sigma \cdot \frac{||w_i||}{||g_i|| + \beta ||w_i||}$
$m_i^{\langle t\rangle} \leftarrow \lambda m_i^{\langle t-1\rangle} + \mu_g^{\langle t\rangle} \cdot \mu_i^{\langle t\rangle} (g^{\langle t\rangle} + \beta w_i^{\langle t-1\rangle})$
$w_i^{\langle t\rangle} \leftarrow w_i^{\langle t-1\rangle} - m_i^{\langle t\rangle}$
end for
end while

Algorithm 18 LARS. Layer indices are denoted with *i*

3.15 Hybrid methods: SWATS, MADGRAD

Hybrid methods are rooted in the idea of combining the strengths of other methods, depending on the context. In particular, in our case, adaptive optimization methods such as Adam, AdaGrad, and RMSprop tend to perform well in the initial portion of the optimization process, but SGD can outperform them at later stages: despite their superior average training outcomes, they have been found to generalize poorly compared to SGD [62]. Consequently, SWATS adopts a hybrid strategy, starting with an adaptive method (Adam) and then switching to SGD

at an appropriate stage. This approach combines the strengths of both methods: Adam quickly converges towards a solution at the early stages, while SGD refines the solution more effectively in the later stages of optimization. Initially, the algorithm runs Adam while providing an estimate of the optimal learning rate of SGD γ_k. When the learning rate of Adam becomes similar enough to γ_k (their difference drops below a threshold ϵ), then γ_k is used as the LR for SGD (see Eq. 36) and the algorithm proceeds with the latter.

$$\left|\bar{\mu}^{adam} - \gamma_k\right| < \epsilon \Rightarrow \mu^{sgd} \leftarrow \gamma_k \tag{36}$$

MadGrad [52], summarized in Algorithm 19, is also introduced as a combination of some of the algorithms introduced earlier. In particular, it is built upon the uncommon dual averaging form of AdaGrad, incorporating multiple techniques we have discussed for other algorithms, including dynamic updates to the LR, momentum, forgetting factors, and weight decay; while retaining provable convergence under convexity assumptions. The goal of the algorithm is to improve over the performance of Adam and its variants in the context of deep learning optimization, and specifically in image classification, in which, in some cases, they generalize poorly.

Input: μ_g: LR; $[\beta^{\langle 0\rangle}, \beta^{\langle 1\rangle}, ..., \beta^{\langle t\rangle}]$: forgetting factor; λ: weight decay; $w^{\langle 0\rangle}$: initial parameters; f: loss function

while $t = 1$ to T **do**

$g^{\langle t\rangle} \leftarrow \nabla f(w^{\langle t-1\rangle})$

$\lambda^{\langle t\rangle} \leftarrow \mu_g \sqrt{t+1}$

$m^{\langle t\rangle} \leftarrow m^{\langle t-1\rangle} + \lambda^{\langle t\rangle} g^{\langle t\rangle}$

$\mu_i^{\langle t\rangle} \leftarrow \mu_i^{\langle t-1\rangle} + \lambda^{\langle t\rangle} (g^{\langle t\rangle})^2$

$w^{\langle t\rangle} \leftarrow (1-\beta^{\langle t\rangle}) w^{\langle t-1\rangle} + \beta^{\langle t\rangle} \left(w^{\langle 0\rangle} - \frac{m^{\langle t\rangle}}{\sqrt[3]{\mu_i^{\langle t\rangle}}} \right)$

end while

Algorithm 19 MadGrad

4 Performance assessment

In this section, we profile the performance of the previously introduced algorithms. We do so in two phases: first, we evaluate their performance on a synthetic problem with low dimensionality and low sparsity, for which the optimal solution is known, and whose difficulty can be adjusted by changing the value of a single parameter; then, we assess their performance on a real-world problem with high dimensionality and high sparsity.

We use the synthetic problem to investigate the search dynamics of each algorithm in detail, analyzing and classifying their convergence behavior (cf. Sect. 4.5) and their robustness to hyperparameter values (cf. Sect. 4.6). We then use the real-world problem to evaluate the algorithms' performance in a more realistic scenario, through the common ResNet18 model and the CIFAR-10 dataset [80].

We use the algorithms' implementation from one of the most well-known libraries for optimization: `torch-optimizer`[1] for Pytorch [81]. The code for both experiments has been released as open-source[2] with a permissive license (MIT) and permanently archived on Zenodo [82].

In this study, we evaluate the algorithms' performance assuming that one step of the optimization process has a similar computational cost for all algorithms. Thus, we focus on convergence by step rather than convergence by effective computation time. This kind of measurement helps with reproducibility, as results can be compared across different hardware configurations, and it is realistic when the computation of one realization of the problem is substantially more expensive than the computation of one step of the optimizer.

[1] https://github.com/jettify/pytorch-optimizer

[2] https://github.com/ruslanissimo/Artefact-2025-first-order-algorithms

 https://doi.org/10.1007/s00521-026-12014-1

4.1 AUC, a convergence quality metric

For both evaluations, we need a metric that can tell the story of an entire optimization process to evaluate the algorithms' dynamics. We thus introduce the Area Under the training-loss Curve (AUC), computed as the sum of all cost function values over time. Given a loss function $\mathcal{J} : \mathbb{N}^+_{\not{0}} \to \mathbb{R}$, AUC over T iterations (epochs) is defined as:

$$\mathrm{AUC} = \sum_{t=0}^{T} \mathcal{J}(t) \tag{37}$$

The AUC provides, in a single number, a measure of how good is the final value or attractor obtained by the optimization process, and of how quickly an optimization algorithm converges:

- if an algorithm ends with a low stable value, the AUC will be stable or growing very little, because less cost is added over time.
- If an algorithm converges quickly, the AUC will decrease more rapidly, thus accumulating less error along the process, and ending with a lower value.

This metric is particularly useful when comparing algorithms with different convergence dynamics, as it can let algorithms oscillate or temporarily increase the cost to "jump" to a better solution.

4.2 Synthetic problem

In this section, we evaluate the performance of the algorithms on the Rosenbrock function [83] (Sect. 4.2.1), performing behavioral analysis (cf. Sect. 4.5) and evaluating robustness to hyperparameter changes (cf. Sect. 4.6).

4.2.1 Problem definition

We use the the search for the minimum of the Rosenbrock function [83] as reference benchmark:

$$f(x, y) = (a - x)^2 + b(y - x^2)^2 \tag{38}$$

The Rosenbrock function is a challenging and commonly used benchmark for optimization algorithms [33], as it features a global minimum at $(x, y) = (a, a^2)$, where $f(a, a^2) = 0$, at the end of a narrow, parabolic-shaped flat valley. Parameters a and b influence the shape and steepness of the function. Changes to a shift the global minimum along the x-axis, while larger values of b change the shape of the valley, making it narrower and steeper as shown in Fig. 3, thus increasing the complexity of the optimization problem. In our experiments, we vary b logarithmically from 10^0 to 10^3 to evaluate the algorithms' performance scaling with increasingly difficult problems.

4.2.2 Evaluation protocol

Hyperparameter optimization

Different algorithms in this review feature hyperparameters that can influence performance significantly. They vary across algorithms (cf. Fig. 4), and their optimal numeric value may be very diverse across algorithms. Thus, hyperparameter need to be optimized as well. Doing so is a meta-optimization problem, typically solved by means of methods such as grid search, random search, Bayesian optimization, and genetic algorithms [84, 85].

In our experiment, to measure near-peak performance for all algorithms under test, we optimize the hyperparameters of each algorithm for every value of b in the Rosenbrock function. We do so using Bayesian Optimization based on a Tree-structured Parzen Estimator (TPE) [86], an approach used successfully in the past to optimize hyperparameters in machine learning algorithms [87]. We do so through the procedure illustrated in Fig. 5, running 500 iterations of hyperparameter optimization, each consisting of 100 realizations of 1000 iterations of

Fig. 3 Rosenbrock function for logarithmically distributed values of b

the Rosenbrock function minimization problem from a random initial point $(x, y) \in ([-2, 2], [-2, 2])$ uniformly distributed within the range. For each parameter, we optimize within a very permissive range that includes all the values suggested in the literature, and expands well past them when mathematically sound.

Table 4 summarizes each hyperparameter, its tested range, and the algorithms to which it applies.

Performance evaluation

After hyperparameter optimization, we run a batch of 1000 realizations of the Rosenbrock function minimization problem from different random initial conditions, using iterations (epochs) count as stopping criterion, stopping at 1000 such iterations.

4.3 Convergence

First, we show how the algorithms under test converge on the Rosenbrock function minimization problem, using a reference values of $b = 1$ and $b = 100$ (cf. Equation 38). Figure 6 shows the best-ever minimum value of the loss function value (left column) and the value of the loss function at each iteration (right column). We can observe that retaining only the minimum value achieved so far hides much of the optimization dynamics, whose evolution over iterations provides insights into the algorithms' behavior, which we will explore in more detail in Sect. 4.5. After an initial phase in which all but AdaMod quickly reduce the loss function value, different algorithms show different convergence dynamics. Some reach very low values quickly, and then stabilize there

 https://doi.org/10.1007/s00521-026-12014-1

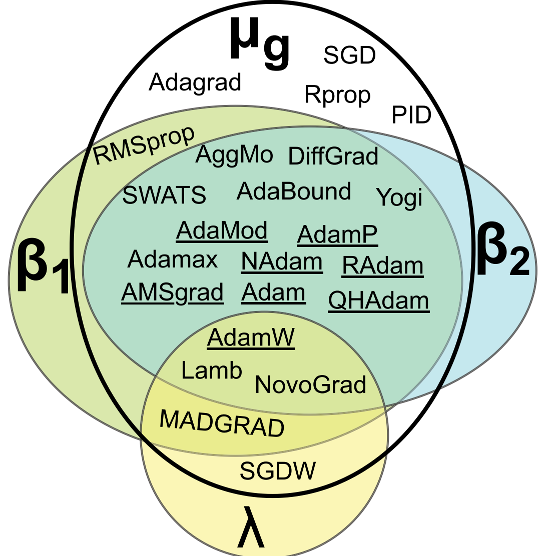

Fig. 4 Venn diagram of the main algorithm hyperparameters from Table 4. Each circle represents a specific hyperparameter and includes all the algorithms that use it. Underlined names identify members of the Adam family

(Rprop, AdaBound, QHAdam, AdaGrad, SGD, SGDW). Yogi also converges to close-to-optimal values, but it does so more slowly than the previously mentioned algorithms, and with an oscillatory behavior. Most other algorithms find their best values within 200 iterations, but then oscillate around them without further improvements. An interesting exception is AdaMod, which is the slowest to converge for $b = 1$, and whose dynamics are characterized by a decrease in performance (increase in the loss function value) that leads the algorithm away from the best-ever minimum found so far.

This behavior is also present (but less marked) in AMSgrad.

4.4 Sensitivity to complexity

We now discuss how the different algorithms perform when increasing the complexity of the Rosenbrock function minimization problem by increasing the value of b in Eq. 38. Data presented in Fig. 7 show that all algorithms are sensitive to increases in the problem complexity. AdaGrad and RMSprop stand out as the most consistent performers, occupying the top two positions in all cases but with $b = 1$ (in which they are in the top three), and with performance decreasing of about 8 times at each complexity step, with no sudden deterioration. AdaBound performs best for $b = 1$ value, but performance decrease significantly for higher values of b, (almost 11 times when raising b from 1 to 10, and again similarly when raising it from 100 to 1000). Conversely, AMSgrad is initially the worst performer ($b = 1$), but it proves to be resilient to initial increases in the problem complexity, showing very good performance for $b = 10$, and then decreasing in performance by about 8–9 times for $b = 100$ and $b = 1000$, a behavior comparable to other good performers. RAdam shows the opposite behavior, starting with better-than-average performance for simpler problems ($b = 1$ and $b = 10$) and then decreasing dramatically in performance for $b = 100$ (a whooping 56 times) and $b = 1000$ (over 13 times), becoming by far the worst performer. The remaining algorithms exhibit AUC increments ranging from 6 to 10 times, with minor fluctuations in their rankings.

4.5 Behavior analysis and classification

While the AUC provides a generalized evaluation of the algorithms' performance, it does not offer direct insights into the optimization process itself to understand *why* algorithms perform in some way and *how* the optimization search pans out. Each method has strengths and weaknesses which, properly understood, can lead to a more conscious selection of a specific algorithm for the problem at hand. To characterize the behavior of the algorithms, we

Fig. 5 Hyperparameter optimization procedure. We run 500 iterations (epochs) of hyperparameter optimization, each of which consisting of 100 realizations of our problem (Rosenbrock function minimization) starting from different random initial conditions and executing for 1000 iterations

look at the dynamics of the loss function over time, and build a new convergence taxonomy based on four main categories of behavior, summarized in Fig. 9:

- *Asymptotic convergence*, in which the loss function decreases monotonically towards numerical zero;
- *Initial-condition dependent*, in which the convergence rate depends on the initial conditions;
- *Stable oscillations*, in which the loss function oscillates around the optimal value;
- *Mixed behavior*, in which different realizations of the optimization problem show different behaviors.

Table 4 Hyperparameters, their range, and applicable algorithms

Hyperparameters	Range, scale	Algorithm
Learning rate (μ_g)	$[10^{-12}, 1]$, log	All algorithms
Forgetting factor (β_1)	[0, 1], uni	See Fig. 4
Forgetting factor (β_2)	[0.9, 0.9999], log	See Fig. 4
Weight decay (λ)	$[10^{-12}, 1]$, log	See Fig. 4
Forgetting factor (β_3)	[0.9, 0.9999], log	AggMo, AdaMod
Forgetting factor (μ^-)	[0.5, 1.0], uni	Rprop
Forgetting factor (μ^+)	[1.0, 1.5], uni	Rprop
Final LR (μ_{fin})	$[10^{-12}, 1]$, log	AdaBound
Decreasing factor (γ)	$[10^{-12}, 1]$, log	AdaBound
Integral weight	[0, 10], uni	PID
Derivative weight	[0, 10], uni	PID
Discount factor (ν_1)	[0, 1], uni	QHAdam
Discount factor (ν_2)	[0, 1], uni	QHAdam

log - logarithmic scale, uni - linear scale

Then, for each degree of complexity of the problem, we observe how the algorithms behave, and we classify them according to our behavior taxonomy in Fig. 8. To provide insights on the way the optimization process unfolds, we also show the X-Y trajectories followed by selected optimizers starting with different initial conditions (phase portraits) in Fig. 10.

4.5.1 Asymptotic convergence

An algorithm is asymptotically convergent if, after a short transition process, its loss decreases monotonically. This is the behavior that AdaBound, SGD, PID, Rprop, SGDW, and AggMo show, regardless of the complexity of the problem. Monotonically convergent algorithms can terminate when they reach numerical zero[3] (e.g., AdaBound, PID); or they get stuck in an infinite loop, oscillating closely around the optimal point with an extremely short period because of a relatively large LR is inducing a numerical error (Rprop, SGDW, AggMo); or they take too long to converge, as convergence rate is too low (SGD).

We depict the phase potrait of AdaBound in Fig. 10. For $b = 1$, it aggressively leaps over the valley and displays random damped oscillations around the global minimum. With $b = 100$, it closely follows the valley shape without overshooting and then slows down, stabilizing. For $b = 1000$, it struggles to reach the valley, oscillating between slopes and showing a certain difficulty in adapting to the valley's curvature, especially for trajectories starting far from the valley. However, independently of the initial conditions, all trajectories ultimately converge asymptotically.

4.5.2 Initial condition dependency

In some cases, although the algorithm shows asymptotic convergence, its convergence rate varies substantially based on the initial conditions. Algorithms with adaptive learning rate, such as AdaGrad, may accumulate large gradient values when the initial point is far from the valley, and small gradient values when the initial point is close to the valley (cf. Eq. 2). Then, due to Eq. 3, the adaptive learning rate respectively decreases or increases, as μ is at the denominator. The absence of a forgetting factor makes the initial steps of the optimization process particularly important for the overall convergence. Algorithms that show this behavior include AdaGrad, AMSgrad, RMSprop, and Yogi. They do so only for simpler problems, as, with increased complexity, their behavior becomes mixed.

[3] A numeric value that can be treated as zero for practical purposes due to limitations in numerical precision, typically due to floating-point arithmetic (e.g., IEEE 754).

Fig. 6 Comparative figure of the convergence curves for the different algorithms on the Rosenbrock function minimization problem with $b = 1$ (top) and $b = 100$ (bottom), showing (left) the best-ever minimum value of the loss function value and (right) the value of the loss function at each iteration

4.5.3 Stable oscillations

Some algorithms may get trapped in stable oscillations around the valley after the transition phase. Adam, AdamP, AdamW, DiffGrad, LAMB, and NovoGrad exhibit this behavior, regardless of the complexity of the problem at hand. Momentum-based algorithms (such as the Adam family), can induce stable oscillations by forming a self-sustained oscillatory system driven by interactions between the accumulated momentum and the gradient value. As the accumulated momentum grows, it increases the LR, causing the algorithm to descend rapidly along the slope of the valley, overshooting the bottom, and climbing the opposite slope. On the opposite slope, the

 https://doi.org/10.1007/s00521-026-12014-1

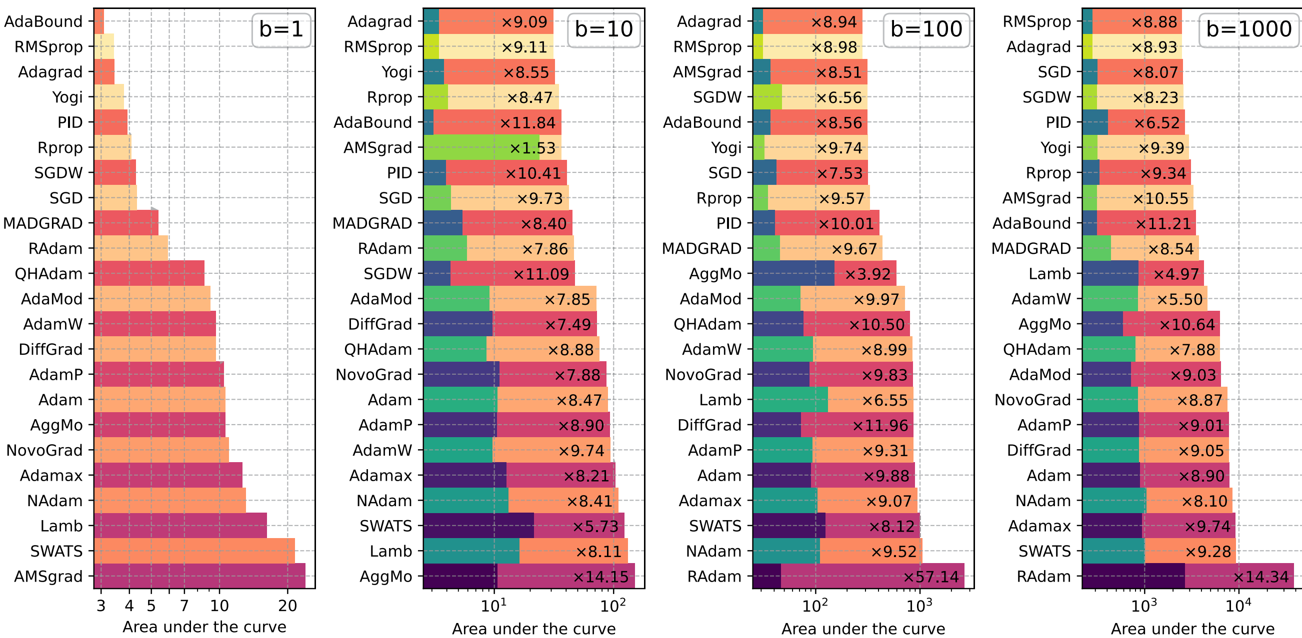

Fig. 7 Rosenbrock function minimization performance measured as convergence via AUC over 1000 iterations, following hyperparameter tuning, for increasing (left to right) values of b. Lower values represent faster convergence and better performance. The cold-colored component of the bar shows the algorithm performance on the immediately simpler problem (the chart at the left of the current one), to underline how performance change with increased complexity

gradient's change in sign reduces the momentum and the LR. Eventually, the momentum starts to accumulate in the opposite direction, repeating the oscillatory process, building a phenomenon akin to a pendulum's motion. In general, oscillations become more pronounced and chaotic as the complexity of the problem increases. The phase portraits of Adam in Fig. 10 show oscillations that become more pronounced as the problem complexity increases.

4.5.4 Mixed behavior

In some cases, algorithms exhibit asymptotic convergence with periodic explosions in the loss function. They converge asymptotically along the valley after locating it but remain on one of its slopes near the valley bottom. This behavior is induced by the parabolic shape of the valley in the X-Y direction (a non-linear interdependency between two parameters under optimization) introducing a constant tracking error. Closer to the optimum, the parabolic shape becomes more linear, eventually allowing the valley to be crossed. Once this happens, the accumulated momentum drives the algorithm further in the same direction, causing the observed explosive peaks.

The sole algorithm exhibiting this behavior consistently across the board is MadGrad; however, all the algorithms showing initial condition dependency for simpler problems (AdaGrad, AMSgrad, RMSprop, and Yogi) switch to mixed behavior for more complex problems. Additionally, RAdam, NAdam, and SWATS steer towards mixed behavior when the problem complexity increases.

In Fig. 10, we can observe that MadGrad's combination of momentum and adaptive gradient decreases the learning rate, leading the algorithm quickly close to the valley. However, immediately later it may transition into oscillations perpendicular to the valley direction (explosions), clearly visible in the phase portraits. In some cases, these oscillations eventually stabilize, leading to asymptotic convergence, especially for simpler problems. Due to the fact that asymptotic convergence is achieved only occasionally, MadGrad behavior is an exemplar of mixed behavior.

Fig. 8 Behavior classification for different problem complexity (columns). Colors identify different behaviors: orange for asymptotical convergence, pink for initial condition dependency, green for mixed behavior, and purple for stable oscillations

4.5.5 Behavior changes with complexity

AdaMax, QHAdam, NAdam, and RAdam show multiple behavior shifts as the problem complexity increases. This suggests that some algorithms will belong to multiple behavior classes within our taxonomy, depending on the complexity of the problem being solved. For example, RAdam shows stable oscillations, typical of Adam and derivatives, for $b = 1$, but then shifts to asymptotic convergence for $b = 10$, and finally showing mixed behavior for higher complexities. We hypothesize that this behavior change is due to the algorithm's adaptive learning rate mechanism with a constant parameter p_t inducing high sensitivity to the problem complexity: although the system has been proven to work well for machine learning tasks, it is not as effective for general-purpose optimization tasks.

In contrast, QHAdam initially exhibits mixed behavior for $b = 1$, then transitions to stable oscillations at $b = 10$, and eventually to asymptotic convergence for highly complex problems. Changes in behavior of QHAdam are clearly visible in its phase portraits (Fig. 10). This shift is linked to the quasi-hyperbolic momentum that combines SGD and Adam's momentum with continuous hyperparameters ν_1 and ν_2. The value of ν_2 drops from 0.95 for $b = 1$ and $b = 10$ to 0.75 for $b = 100$, reducing the influence of momentum and enhancing stability, which leads to the observed behavior change: QHAdam seems to be capable to effectively adapt to the problem complexity.

4.6 Hyperparameter robustness

In Sect. 4.5, we learned that hyperparameters can be critical in shaping the behavior and performance of optimization algorithms, see for instance the discussion about the behavior of QHAdam in Sect. 4.5.5. In practical applications, however, optimization problems are often so complex that hyperparameter tuning becomes prohibitively time-consuming. Consequently, fixed "reasonable" hyperparameters are frequently used without further adjustment. This makes evaluating algorithm performance under non-optimized hyperparameters essential. To address this, we fix each algorithm's hyperparameters to the values optimized for $b = 1000$ (the most complex case shown in Fig. 12 (left)) and evaluate the increase in AUC for simpler problems ($b = 1, 10, 100$) compared to their performance with specifically-optimized hyperparameters.

Our robustness-to-hyperparameters analysis uses the same Bayesian Optimization procedure described in Sect. 4.2.2. Consequently, we optimize all hyperparameters of each algorithm in a single joint procedure, thereby quantifying how performance varies across the learning-rate range and, where present, momentum- and decay-related parameters. In this analysis, we do not isolate the effect of each hyperparameter individually; performing exhaustive one-dimensional sweeps of each hyperparameter on a fixed grid while keeping all others constant. A full factorial, multi-dimensional sensitivity study for every algorithm in our portfolio would be computationally expensive and strongly algorithm-specific, and is therefore beyond the scope of this survey article; such detailed sensitivity analyses are more naturally carried out in the original papers that introduce each optimizer or in dedicated follow-up studies.

Figure 11 shows the results. The performance of AdaGrad, RMSprop, Yogi, Rprop, QHAdam, DiffGrad, Adam, AdamP, AdaMax, NovoGrad, SWATS, and NAdam show high robustness to hyperparameter changes, with changes in the AUC of no more than 12%. Conversely, RAdam, PID, SGDW, SGD, AdaBound, AdaMod, MadGrad, AggMo, and LAMB exhibit high sensitivity to hyperparameters, with significant performance differences (up to 300 times) between unoptimized and specifically-optimized hyperparameters.

Figure 12 shows the ratio between the smallest and largest global learning rate obtained after hyperparameter optimization for each problem, to provide an idea of how large is the "reasonable" range of values for each algorithm. Larger values indicate lower stability of the global learning rate, and thus they are a general indicator of the algorithm's sensitivity to hyperparameters. In fact, algorithms that have the largest performance boost with hyperparameter optimization in Fig. 11 are also those with the largest ratio in Fig. 12.

By these results, we can conclude that AdaGrad, RMSprop, Yogi, and Rprop are particularly robust to hyperparameter changes, and as such they should be preferred in practical applications where hyperparameter optimization is not feasible.

4.7 Practical machine learning application

The optimization of the Rosenbrock function is a synthetic test often used as a benchmark for evaluating optimization algorithms due to its challenging shape, but real-world problems may present different challenges. In particular, the Rosenbrock function is a low-dimensional non-sparse problem, while machine learning training tasks (in which these optimization algorithms are often used), typically require to optimize highly-multidimensional and very sparse settings, introducing a different type of complexity.

To better assess the performance of optimization algorithms on practical tasks, we used the widely used ResNet18 model and the CIFAR-10 dataset [80]. ResNet18, a convolutional neural network, is particularly well-suited for image classification, while CIFAR-10 consists of 60000 32x32 color images across 10 classes: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The objective is to “learn” the characteristics of each class from the dataset and accurately classify new images. The percentage of images correctly classified represent the recognition *accuracy* of the model. This learning process is inherently an optimization task, where the algorithm adjusts the neural network’s weights to minimize the error between predicted probabilities and actual labels.

4.7.1 Evaluation protocol

To tackle this optimization task efficiently, the dataset was divided into 128 batches, and the optimization algorithm is applied to each batch, performing 100 iterations (stopping criterion) on the ResNet18 model. For every step of the optimization process, we measure memory usage and execution time, of each algorithm and then average them over all batches. The experiment was conducted using a 2019 MacBook Pro equipped with a 1.4GHz quad-core Intel® Core i5®, Turbo Boost up to 3.9GHz and 8GB 2133 MHz LPDDR3. For this experiment we did not perform hyperparameter optimization, as the task was deemed too computationally intensive, instead, we relied on the recommended default hyperparameters.

4.7.2 Results

The results are summarized in Fig. 13. We observe that most algorithms achieved a similar recognition accuracy, close to 83%, with a group of algorithms including AdamP, LAMB, AggMo, NovoGrad, AdaGrad, PID, SGD, and Rprop, producing worse results. If we measure the AUC of the loss function, we learn that algorithms achieving good recognition accuracy also show good integral metrics, an indication that they converge to a good solution relatively quickly. There is one notable exception: MadGrad. MadGrad is among the top performers accuracy-wise, but its AUC is slightly below-median, indicating convergence to the vicinity of the solution with a rate relatively lower other top performers.

In terms of memory usage, the algorithms ranged from approximately 300 to 600 MB. The Adam family (Adam, NAdam, Adamax, and AdamP) demonstrated the highest memory usage, consuming between 500 and 600 MB, while simpler methods like Adagrad and AdaMod required only around 300 MB.

Execution time varied dramatically, making from 7 ms to 130 ms per optimization step. Adagrad and QHAdam were the fastest, requiring 7 ms and 7.2 ms per step, respectively. In contrast, AMSGrad, Adamax, MadGrad, and Yogi were the slowest, taking more than 100 ms per step.

4.8 A practitioner’s guide to choosing an optimizer

Premise Across standard ML workloads, many optimizers reach similar final accuracy given enough epochs (Fig. 13); differences matter most for (i) time-to-quality, (ii) stability under noise/scale, and (iii) robustness when hyperparameter tuning is limited (Figs. 7 and 11). The checklist below is a concise, empirically anchored rule-of-thumb from our study, visually summarized in Fig. 14.

Step 0: Are gradients available? If the objective is differentiable (typical for deep learning), use first-order, derivative-based methods; they dominate in scalability and hardware efficiency. If not, gradient-free methods are out of scope here.

Step 1: Pick the base family by regime

 https://doi.org/10.1007/s00521-026-12014-1

- *Fast early progress under mini-batch noise (most DL training)*: **AdamW**. Rationale: better generalization than Adam when regularization is desired, since weight decay is decoupled from the adaptive step (Sect. 3.10); stable, low-friction descent (Fig. 7).
- *Very large batches / pretraining at scale*: **LAMB** (or **LARS** for SGD-based pipelines). Rationale: layer-wise trust ratios keep update norms well-conditioned across layers, enabling high throughput (Algorithm 17 and 18); expect wins at scale even if small-batch CIFAR does not highlight them (Fig. 13).
- *Sparse or multi-scale features (e.g., embeddings)*: **AdaGrad** , or **AdamW** if you also want fast early progress. Rationale: coordinate-wise adaptation; AdaGrad is notably robust without tuning (Fig. 11).*Step 2: Adjust for stability and tuning budget*

- *Little or no hyperparameter search*: prefer **AdaGrad** , **RMSprop**, **Yogi**, or **Rprop** (top robustness in Fig. 11).
- *Early oscillations or LR blow-ups*: add warmup/rectification (**RAdam**), or cap adaptation (**AdaBound**/**AdaMod**), or derivative damping (**PID**); see behavior taxonomy in Figs. 8 and 9.
- *Tuning the speed–stability trade-off without changing the family*: **QHAdam** gives a smoother control surface and adapts across difficulty levels (Figs. 8 and 10).*Step 3: Late-stage generalization* If you need the last few points on the test set: either (i) *switch* from an adaptive method to **SGD**+ **NAG** for the tail phase, or (ii) use **SWATS** to do the switch automatically (Sect. 3.7; Algorithm 19 and Fig. 13). This matches the common " AdamW for speed, SGD for finish" practice.

Sanity checks (from our results)

1. Many methods tie on final accuracy in 100 epochs (Fig. 13); choice mainly affects stability and time-to-quality.
2. Robustness matters if you cannot tune (Fig. 11): AdaGrad /RMSprop/Yogi/Rprop degrade the least off optimum hyperparameters.
3. If you see long-lived oscillations, you are likely in an Adam-family regime where momentum and curvature interact (Fig. 9); warmup, bounds, or a late SGD phase fix it in practice.

5 Conclusion

This paper explores modern first-order optimization algorithms, analyzing 23 distinct methods and their underlying principles. The algorithms were iteratively considered and compared within a unified notation framework. We conducted a comprehensive evaluation of their performance, using the Rosenbrock function as a synthetic benchmark to investigate the details of their internal behavior with raising complexity. We found four main categories of behavior: asymptotically convergent, oscillative, sensitive to initial conditions, and mixed. We measured the robustness of the algorithms to hyperparameter changes. Hyperparameter optimization was conducted for all algorithms at each complexity level to ensure optimal performance. Finally, we evaluated the algorithms' performance on a practical machine learning task, finding that most algorithms achieved similar accuracy within 100 epochs.

In the future, we plan to profile the performance of the algorithms considering the computational cost of each iteration, thus moving from a convergence-by-step to a convergence-by-computation-time evaluation. Additionally, as novel methods are devised, they should be included in the analysis to provide a comprehensive overview of the field. Finally, the proposed review could be extended to include more practical machine learning tasks in different domains (speech recognition, Large Language Models, Generative Adversarial Networks, etc.) to measure the algorithms' performance and robustness in a broader context.

Fig. 9 Loss trajectories on Rosenbrock illustrating four convergence patterns, top to bottom: *Asymptotic convergence*, in which values progressively converge to numerical zero; *Initial-condition dependent*, in which the convergence rate depends on the initial conditions; *Oscillatory*, in which the loss function oscillates around the optimal value; and *Mixed* in which different realizations of the optimization problem show different behaviors

https://doi.org/10.1007/s00521-026-12014-1

Fig. 10 Phase portraits of AdaBound, Yogi, QHAdam, MadGrad, and Adam for varying Rosenbrock parameter b. The chart illustrates convergence classes: Asymptotic (AdaBound), Dependent (Yogi), Oscillation (Adam), and Mixed (MadGrad). It highlights MadGrad's explosive tendency towards mixed behavior and QHAdam's adaptability

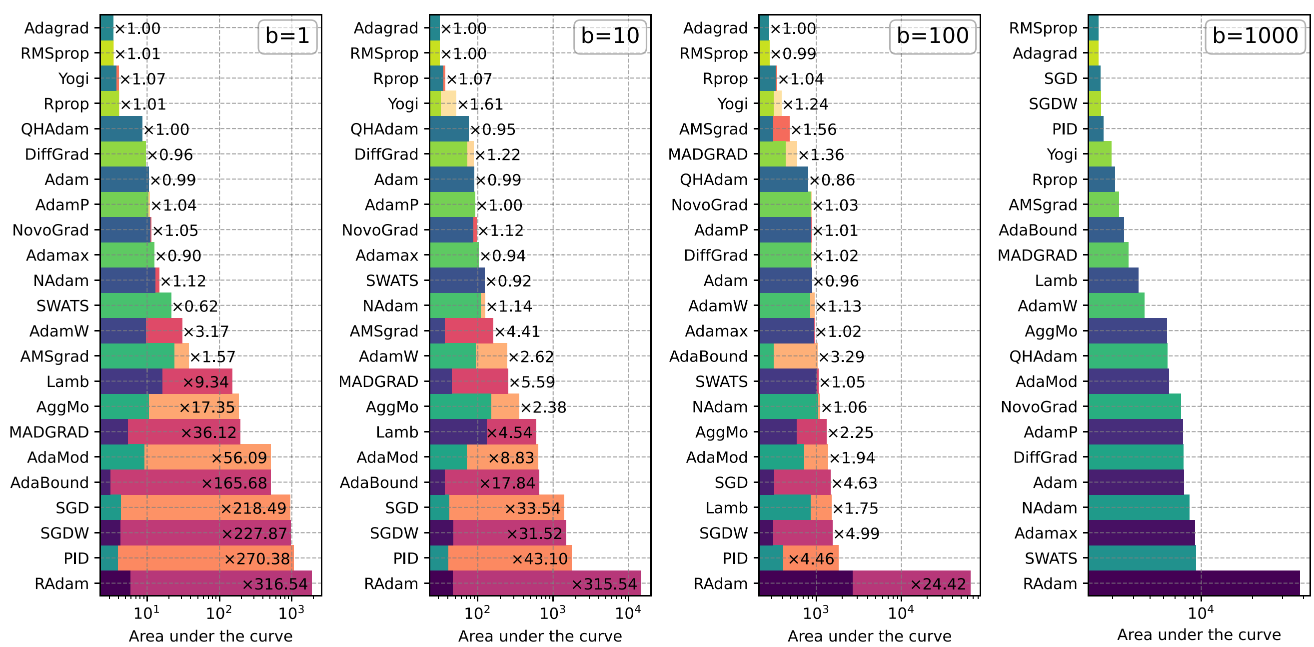

Fig. 11 Robustness to hyperparameter changes, by comparing the performance between instances with hyperparameters optimized on the specific problem, and instances with fixed hyperparameters (using the values optimized for $b = 1000$). Performance are measured using AUC (cf. Sect. 4.1) The bar length depicts the unoptimized version performance, while the cold-colored part of each bar shows performance *after hyperparameter optimization*. The note on the bar indicates the improvement obtained by hyperparameter optimization. Charts range from simpler (left) to more complex (right) problems

5.1 Future research directions

Based on our analysis (performance in Fig. 7, behavior taxonomy in Figs. 8 and 9, robustness in Fig. 11, and ML results in Fig. 13), we see several concrete avenues:

1. *Time- and budget-normalized evaluation* Move beyond step-based AUC to report time-to-target, energy-to-target, and memory-footprint, under fixed *tuning budgets*. Standardize benchmarks with both *equalized compute* and *equalized HPO* settings, so results are not artifacts of tuning freedom.
2. *Theory for oscillatory and mixed regimes* Our taxonomy shows stable oscillations (e.g., Adam/ AdamW/NovoGrad) and mixed behaviors (MadGrad, RAdam, QHAdam shifting with complexity). We need dynamical-systems/control-theory analyses (Lyapunov/anti-windup models) for QHM, warmup/rectification, and PID-style terms to predict and eliminate limit cycles.
3. *Self-tuning, closed-loop optimizers* Design controllers that *measure* variance, curvature proxies, and oscillation indicators online, then adjust LR, momentum, decay, and bounds automatically. Generalize SWATS-style switching beyond Adam→SGD to multi-way, per-layer switching with detection logic.
4. *Curvature-aware first-order preconditioning* Bridge to second-order benefits at first-order cost using cheap diagonal/low-rank statistics (e.g., Hutchinson trace, Fisher diagonals), sharpness-aware steps compatible with AdamW, and per-layer trust regions (LAMB/LARS-like) that adapt to curvature changes.
5. *Robust-to-tuning algorithms with guarantees* Fig. 11 shows large gaps when hyperparameter optimization is limited. Target methods with provable bounded regret vs. tuned optima; emphasize *safe defaults* that degrade gracefully across data scales, batch sizes, and losses (non-smooth, heavy-tailed, label noise).
6. *Numerics for low precision and huge models* Formalize stability for FP8/INT8 training and quantized optimizer states (e.g., "8-bit Adam"), including bias corrections that keep the QHM/moment dynamics intact. Study projection/bounding (AdamP, AdaBound/AdaMod) under quantization.

 https://doi.org/10.1007/s00521-026-12014-1

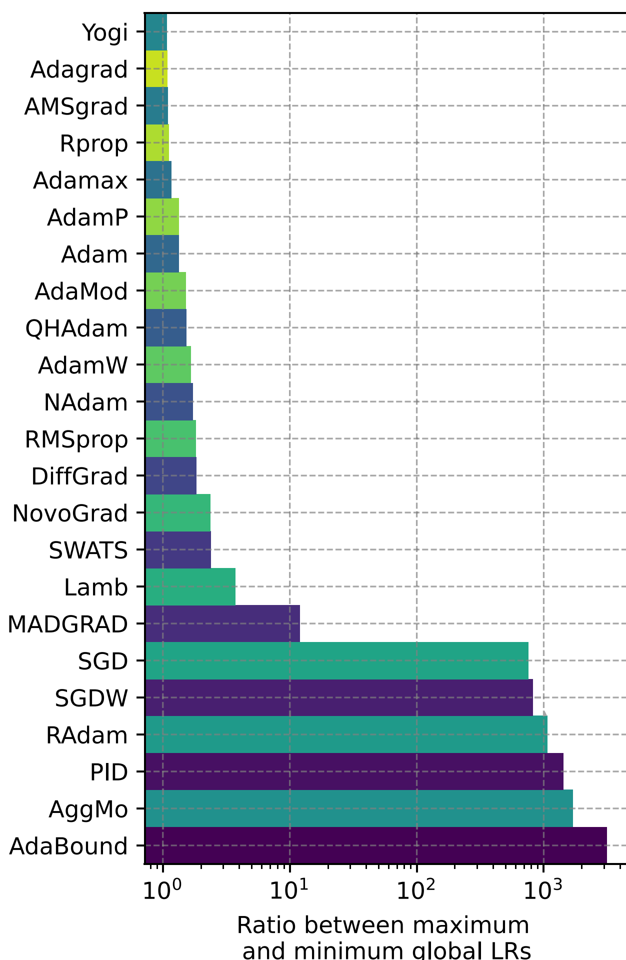

Fig. 12 Ratio between maximum and minimum optimized learning rates. Larger values indicate higher sensitivity to hyperparameter optimization

7. *Richer benchmark suite with controllable difficulty* Rosenbrock gave clear diagnostics; extend with families that expose non-smoothness, plateaus, anisotropy, and constraints. For machine learning, cover transformers/LLMs, diffusion, and Generative Adversarial Network (GAN) stability with fixed compute and fixed hyperparameter optimization tracks and publicly released logs/seeds.
8. *Regularization beyond decoupled weight decay* Generalize AdamW to learn per-parameter/per-group decay on-the-fly (data-dependent shrinkage), and study its interaction with layer-wise scaling (LAMB) and projection (AdamP) in scale-invariant architectures.
9. *Geometry- and constraint-aware updates* Systematize projection-based ideas (AdamP) into trust-region or manifold-aware first-order methods for scale-invariant or norm-constrained models, with detection rules that trigger geometry changes only when needed.
10. *Optimizer selection and meta-learning* Train lightweight meta-policies that, after a short probe (estimating gradient variance, curvature, anisotropy), choose among AdamW/SGD+ NAG/LAMB/QHAdam/Yogi and set schedules. Release an open corpus of optimizer traces to enable this.
11. *Handling non-stationarity and continual learning* Develop optimizers that adapt when the objective drifts (curriculum/domain shift), with momentum “reset/gating” and state partitioning to reduce catastrophic interference while keeping fast adaptation.
12. *Robustness and safety* Make first-order methods resilient to outliers/adversaries (norm clipping, noise shaping, robust loss coupling) with theory and standardized robustness tests, not just clean accuracy/AUC.

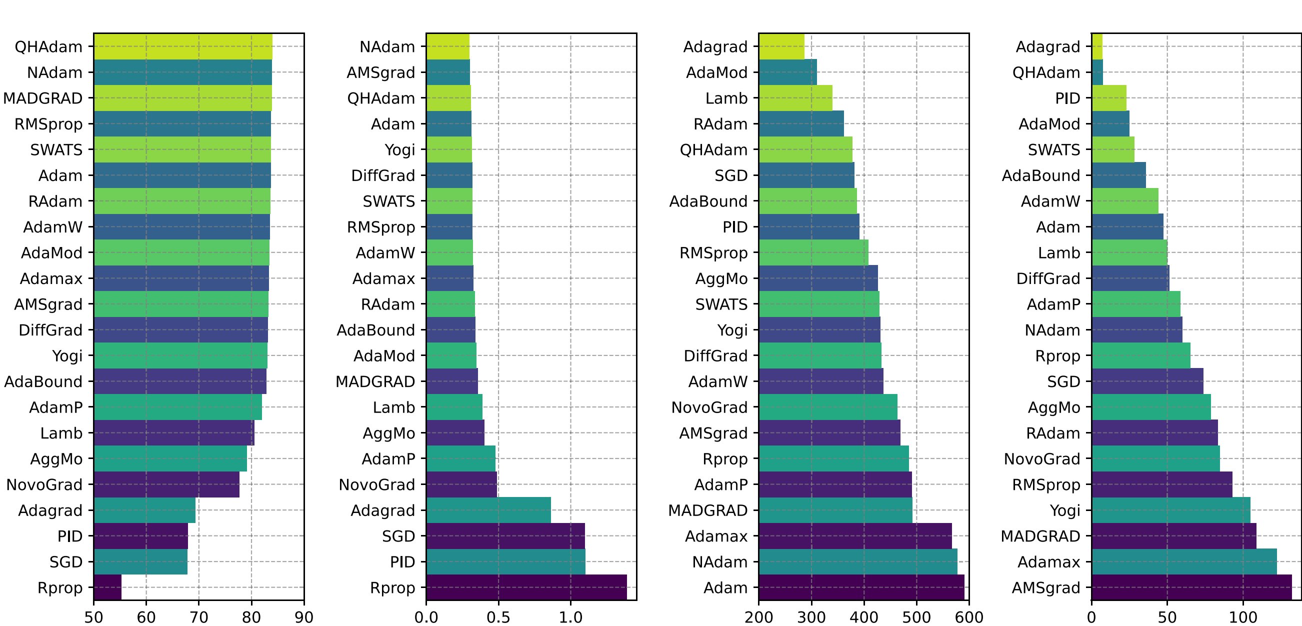

Fig. 13 Performance of the algorithms under evaluation training ResNet18 on the CIFAR-10 dataset for 100 epochs. Metrics shown are recognition accuracy (left, higher is better), convergence via AUC (center-left, lower is better), memory used per batch (center-right, lower is better), and execution time per optimization step (right, lower is better)

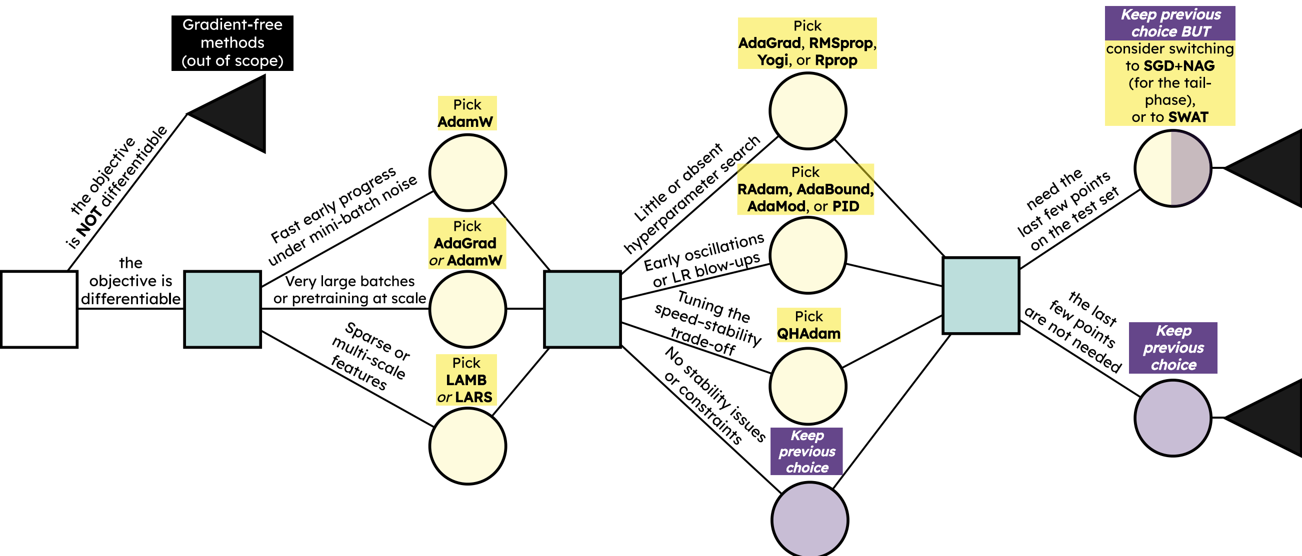

Fig. 14 Decision Tree for choosing first-order optimization algorithms (to be used left-to-right). Squares represents a decision point, circles outcomes, and triangles represent end-points. The color of outcomes indicates whether the outcome differs from a previous one (yellow) or is the same (purple)

13. *Fair comparisons to NIAs and hybrids* When differentiability breaks (augmentations, discrete modules), define *budget-fair*, problem-matched comparisons to NIAs, and study hybrids that use NIAs for initialization/ structure search and switch to SGD/ AdamW for fine-tuning.
14. *Detailed sensitivity analysis to hyperparameters* In this work, robustness-to-hyperparameters is assessed by jointly variating LR and (where applicable) other hyperparameters. As a complementary direction, a

dedicated follow-up study could perform systematic one-dimensional sweeps of individual hyperparameters while keeping all other hyperparameters fixed, thereby providing clearer response curves and isolating the marginal effect of each parameter on convergence speed and final performance.

Author contributions **Ruslan Shaiakhmetov**: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data Curation, Writing - Original Draft, Writing - Review & Editing, Visualization. **Danilo Pianini**: Conceptualization, Methodology, Validation, Investigation, Resources, Writing - Original Draft, Writing - Review & Editing, Visualization, Supervision, Project administration, Funding acquisition. **Angelo Filaseta**: Writing - Original Draft, Writing - Review & Editing, Visualization. **Gabriele D'Angelo**: Writing - Original Draft, Writing - Review & Editing, Supervision. **Valter Venusti**: Resources, Supervision.

Funding Open access funding provided by Alma Mater Studiorum - Università di Bologna within the CRUI-CARE Agreement. The efforts of Ruslan Shaiakhmetov work is supported by the project "Ottimizzazione computazione di un modello di simulazione dinamica del veicolo" funded by the European Union - NextGenerationEU and Dallara Automobili S.p.A. through the Italian "National Recovery and Resilience Plan" (PNRR) Mission 4, Component 2, Investment 3.3 (DM 352/2022) - CUP J33C22001400009. The efforts of Angelo Filaseta have been supported by "WOOD4.0 - Woodworking Machines for Industry 4.0" funded by Emilia-Romagna through the regional project call 2022, art. 6 L.R. N. 14/2014 (DGR 1098/2022) - CUP E69J22007520009

Code availability The code used in this study has been released as open-source at https://github.com/ruslanissimo/Artefact-2025-first-order-algorithms with a permissive license (MIT) and permanently archived on Zenodo [82].

Declarations

Conflict of interest The authors have no conflict of interest.

Ethical approval This paper does not contain any studies with human participants or animals performed by any of the authors.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Zomorrodi AR, Suthers PF, Ranganathan S, Maranas CD (2012) Mathematical optimization applications in metabolic networks. Metab Eng 14(6):672–686. https://doi.org/10.1016/j.ymben.2012.09.005
2. Intriligator MD (2002) Mathematical optimization and economic theory. Soc Ind Appl Math Phila PA. https://doi.org/10.1137/1.9780898719215
3. Nocedal J, Wright SJ (2006) Numerical optimization, 2e edn. Springer, New York, NY, USA
4. Sun S, Cao Z, Zhu H, Zhao J (2020) A survey of optimization methods from a machine learning perspective. IEEE Trans Cybern 50(8):3668–3681. https://doi.org/10.1109/TCYB.2019.2950779
5. Goodfellow I, Bengio Y, Courville A (2016) Deep Learning. MIT Press, Cambridge, MA
6. Geczy P, Akaho S, Usui S (2006) Efficient first order superlinear algorithms. In: 9th international symposium on artificial intelligence and mathematics, ISAIM 2006
7. Battiti R (1992) First- and second-order methods for learning: between steepest descent and newton's method. Neural Comput 4(2):141–166. https://doi.org/10.1162/NECO.1992.4.2.141
8. Bottou L, Curtis FE, Nocedal J (2018) Optimization methods for large-scale machine learning. SIAM Rev 60(2):223–311. https://doi.org/10.1137/16M1080173
9. Tan HH, Lim KH (2019) Review of second-order optimization techniques in artificial neural networks backpropagation. IOP Conf Ser Mater Sci Eng 495:012003. https://doi.org/10.1088/1757-899x/495/1/012003

10. Kmich M, Ghouate NE, Bencharqui A, Karmouni H, Sayyouri M, Askar SS, Abouhawwash M (2025) Chaotic puma optimizer algorithm for controlling wheeled mobile robots. Eng Sci Technol Int J 63:101982. https://doi.org/10.1016/j.jestch.2025.101982
11. Bencherqui A, Karmouni H, Daoui A, Alfidi M, Qjidaa H, Sayyouri M (2020) Optimization of Jacobi moments parameters using artificial bee colony algorithm for 3d image analysis. In: 2020 fourth international conference on intelligent computing in data sciences (ICDS), pp. 1–7. IEEE, Piscataway, NJ. https://doi.org/10.1109/icds50568.2020.9268736
12. Bencherqui A, Tahiri MA, Karmouni H, Daoui A, Alfidi M, Jamil MO, Qjidaa H, Sayyouri M (2022) Optimization of Meixner moments by the firefly algorithm for image analysis. Springer, Cham, pp 439–448. https://doi.org/10.1007/978-3-031-01942-5_44
13. Bencherqui A, Tamimi M, Tahiri MA, Karmouni H, Alfidi M, Jamil MO, Qjidaa H, Sayyouri M (2023) Optimal color image watermarking based on dwt-svd using an arithmetic optimization algorithm. Springer, Cham, pp 441–450. https://doi.org/10.1007/978-3-031-29860-8_45
14. Amari S (1993) Backpropagation and stochastic gradient descent method. Neurocomputing 5(3):185–196. https://doi.org/10.1016/0925-2312(93)90006-O
15. Zhang T (2004) Solving large scale linear prediction problems using stochastic gradient descent algorithms. In: Brodley CE (ed.) machine learning, proceedings of the twenty-first international conference (ICML 2004), Banff, Alberta, Canada, July 4–8, 2004. ACM international conference proceeding series, vol. 69. ACM, New York, NY. https://doi.org/10.1145/1015330.1015332. https://doi.org/10.1145/1015330.1015332
16. Bottou L (2010) Large-scale machine learning with stochastic gradient descent. In: Lechevallier Y, Saporta G (eds.) 19th international conference on computational statistics, COMPSTAT 2010, Paris, France, August 22–27, 2010 - Keynote, invited and contributed papers, pp. 177–186. Physica-Verlag, Heidelberg. https://doi.org/10.1007/978-3-7908-2604-3_16. https://doi.org/10.1007/978-3-7908-2604-3_16
17. Golmant N, Vemuri N, Yao Z, Feinberg V, Gholami A, Rothauge K, Mahoney MW, Gonzalez J (2018) On the computational inefficiency of large batch sizes for stochastic gradient descent. CoRR arxiv:1811.12941
18. Chaudhari P, Baldassi C, Zecchina R, Soatto S, Talwalkar A (2017) Parle: parallelizing stochastic gradient descent. CoRR arxiv:1707.00424
19. Gemulla R, Nijkamp E, Haas PJ, Sismanis Y (2011) Large-scale matrix factorization with distributed stochastic gradient descent. In: Apté C, Ghosh J, Smyth P (eds.) proceedings of the 17th ACM SIGKDD international conference on knowledge discovery and data mining, San Diego, CA, USA, August 21–24, 2011, pp. 69–77. ACM, New York, NY, USA. https://doi.org/10.1145/2020408.2020426. https://doi.org/10.1145/2020408.2020426
20. Srinivasan V, Sankar AR, Balasubramanian VN (2018) ADINE: an adaptive momentum method for stochastic gradient descent. In: Ranu, S., Ganguly, N., Ramakrishnan, R., Sarawagi, S., Roy, S. (eds.) proceedings of the ACM India joint international conference on data science and management of data, COMAD/CODS 2018, Goa, India, January 11–13, 2018, pp. 249–256. ACM, New York, NY, USA. https://doi.org/10.1145/3152494.3152515
21. Ferrarotti L, Bemporad A (2019) Synthesis of optimal feedback controllers from data via stochastic gradient descent. In: 17th European control conference, ECC 2019, Naples, Italy, June 25–28, 2019, pp. 2486–2491. IEEE, Piscataway, NJ, USA. https://doi.org/10.23919/ECC.2019.8796130
22. Yang L, Wang K, Xu C, Zhu C, Sun Y (2016) An incremental learning classification algorithm based on forgetting factor for ehealth networks. In: 2016 IEEE international conference on communications, ICC 2016, Kuala Lumpur, Malaysia, May 22–27, 2016, pp. 1–6. IEEE, Piscataway, NJ, USA. https://doi.org/10.1109/ICC.2016.7511386
23. Loshchilov I, Hutter F (2019) Decoupled weight decay regularization. In: 7th international conference on learning representations, ICLR 2019, New Orleans, LA, USA, May 6–9, 2019. OpenReview.net, Online. https://openreview.net/forum?id=Bkg6RiCqY7
24. Xie Z, Sato I, Sugiyama M (2020) Stable weight decay regularization. CoRR arxiv:2011.11152
25. Xie Z, Sato I, Sugiyama M (2020) Understanding and scheduling weight decay. arXiv. Arxiv:2011.11152
26. Dvurechensky PE, Shtern S, Staudigl M (2021) First-order methods for convex optimization. EURO J Comput Optim 9:100015. https://doi.org/10.1016/J.EJCO.2021.100015
27. Zucchet N, Orvieto A (2024) Recurrent neural networks: vanishing and exploding gradients are not the end of the story. In: Globersons A, Mackey L, Belgrave D, Fan A, Paquet U, Tomczak JM, Zhang C (eds.) advances in neural information processing systems 38: annual conference on neural information processing systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10–15, 2024 http://papers.nips.cc/paper_files/paper/2024/hash/fbb07254ef01868967dc891ea3fa6c13-Abstract-Conference.html
28. Rehmer A, Kroll A (2020) On the vanishing and exploding gradient problem in gated recurrent units. IFAC-PapersOnLine 53(2):1243–1248. https://doi.org/10.1016/j.ifacol.2020.12.1342
29. Esther B, Aviv G, Karl-Heinz K (2020) Nesterov perturbations and projection methods applied to IMRT. J Nonlinear Var Analy. https://doi.org/10.23952/jnva.4.2020.1.06
30. Jin C, Netrapalli P, Ge R, Kakade SM, Jordan MI (2021) On nonconvex optimization for machine learning: gradients, stochasticity, and saddle points. J ACM 68(2):11–11129. https://doi.org/10.1145/3418526

31. Soydaner D (2020) A comparison of optimization algorithms for deep learning. Int J Pattern Recognit Artif Intell 34(13):2052013–1205201327. https://doi.org/10.1142/S0218001420520138
32. Marin I, Kuzmanic Skelin A, Grujic T (2020) Empirical evaluation of the effect of optimization and regularization techniques on the generalization performance of deep convolutional neural network. Appl Sci 10(21):7817. https://doi.org/10.3390/app10217817
33. Emiola I, Adem R (2021) Comparison of minimization methods for rosenbrock functions. In: 29th Mediterranean conference on control and automation, MED 2021, Bari, Italy, June 22–25, 2021, pp. 837–842. IEEE, Piscataway, NJ, USA. https://doi.org/10.1109/MED51440.2021.9480200
34. Ngartera L, Diallo C (2024) A comparative study of optimization techniques on the Rosenbrock function. Open J Optim 13(03):51–63. https://doi.org/10.4236/ojop.2024.133004
35. Shang J, Ma T, Xiao C, Sun J (2019) Pre-training of graph augmented transformers for medication recommendation. ijcai.org. https://doi.org/10.24963/IJCAI.2019/825
36. Gupta C, Balakrishnan S, Ramdas A (2021) Path length bounds for gradient descent and flow. J Mach Learn Res 22:68–16863
37. Goh GB, Hodas NO, Vishnu A (2017) Deep learning for computational chemistry. J Comput Chem 38(16):1291–1307. https://doi.org/10.1002/jcc.24764
38. Cohen J, Kaur S, Li Y, Kolter JZ, Talwalkar A (2021) Gradient descent on neural networks typically occurs at the edge of stability. In: 9th international conference on learning representations, ICLR 2021, Virtual Event, Austria, May 3–7, 2021. OpenReview.net, Online. https://openreview.net/forum?id=jh-rTtvkGeM
39. Ma C, Wu L, E, W (2021) A qualitative study of the dynamic behavior for adaptive gradient algorithms. In: Bruna, J., Hesthaven JS, Zdeborová L (eds.) Mathematical and scientific machine learning, 16–19 August 2021, virtual conference/Lausanne, Switzerland. Proceedings of machine learning research, vol. 145, pp. 671–692. PMLR, Online. https://proceedings.mlr.press/v145/ma22a.html
40. Luo L, Xiong Y, Liu Y, Sun X (2019) Adaptive gradient methods with dynamic bound of learning rate
41. Duchi JC, Hazan E, Singer Y (2011) Adaptive subgradient methods for online learning and stochastic optimization. J Mach Learn Res 12:2121–2159. https://doi.org/10.5555/1953048.2021068
42. Kingma DP, Ba J (2015) Adam: A method for stochastic optimization. In: Bengio Y, LeCun Y (eds.) 3rd international conference on learning representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, conference track proceedings. arxiv:1412.6980
43. Ding J, Ren X, Luo R, Sun X (2019) An adaptive and momental bound method for stochastic learning. CoRR arxiv:1910.12249
44. Heo B, Chun S, Oh SJ, Han D, Yun S, Kim G, Uh Y, Ha J (2021) Adamp: Slowing down the slowdown for momentum optimizers on scale-invariant weights. In: 9th international conference on learning representations, ICLR 2021, virtual event, Austria, May 3–7, 2021. OpenReview.net, Online. https://openreview.net/forum?id=Iz3zU3M316D
45. Loshchilov I, Hutter F (2019) Decoupled weight decay regularization. In: 7th international conference on learning representations, ICLR 2019, New Orleans, LA, USA, May 6–9, 2019. OpenReview.net, Online. https://openreview.net/forum?id=Bkg6RiCqY7
46. Lucas J, Zemel RS, Grosse RB (2018) Aggregated momentum: stability through passive damping. CoRR arxiv:1804.00325
47. Reddi SJ, Kale S, Kumar S (2019) On the convergence of Adam and beyond. CoRR arxiv:1904.09237
48. Dubey SR, Chakraborty S, Roy SK, Mukherjee S, Singh SK, Chaudhuri BB (2020) diffgrad: an optimization method for convolutional neural networks. IEEE Trans Neural Networks Learn Syst 31(11):4500–4511. https://doi.org/10.1109/TNNLS.2019.2955777
49. Eglynas T, Lizdenis D, Raudys A, Jakovlev S, Voznák M (2025) Exploring generative adversarial networks: comparative analysis of facial image synthesis and the extension of creative capacities in artificial intelligence. IEEE Access 13:19588–19597. https://doi.org/10.1109/ACCESS.2025.3531726
50. You Y, Li J, Reddi SJ, Hseu J, Kumar S, Bhojanapalli S, Song X, Demmel J, Keutzer K, Hsieh C (2020) Large batch optimization for deep learning: training BERT in 76 minutes. In: 8th international conference on learning representations, ICLR 2020, Addis Ababa, Ethiopia, April 26–30, 2020. OpenReview.net, Online. https://openreview.net/forum?id=Syx4wnEtvH
51. You Y, Gitman I, Ginsburg B (2017) Large batch training of convolutional networks. https://doi.org/10.48550/ARXIV.1708.03888
52. Defazio A, Jelassi S (2021) Adaptivity without compromise: a momentumized, adaptive, dual averaged gradient method for stochastic optimization. CoRR arxiv:2101.11075
53. Dozat T (2016) Incorporating Nesterov momentum into Adam
54. Lin J, Song C, He K, Wang L, Hopcroft JE (2020) Nesterov accelerated gradient and scale invariance for adversarial attacks. In: 8th international conference on learning representations, ICLR 2020, Addis Ababa, Ethiopia, April 26–30, 2020. OpenReview.net, Online. https://openreview.net/forum?id=SJlHwkBYDH

55. Ginsburg B, Castonguay P, Hrinchuk O, Kuchaiev O, Lavrukhin V, Leary R, Li J, Nguyen H, Cohen JM (2019) Stochastic gradient methods with layer-wise adaptive moments for training of deep networks. CoRR arxiv:1905.11286
56. An W, Wang H, Sun Q, Xu J, Dai Q, Zhang L (2018) A PID controller approach for stochastic optimization of deep networks. In: 2018 IEEE conference on computer vision and pattern recognition, CVPR 2018, Salt Lake City, UT, USA, June 18–22, 2018, pp. 8522–8531. Computer vision foundation/IEEE computer society, Los Alamitos, CA, USA. https://doi.org/10.1109/CVPR.2018.00889. http://openaccess.thecvf.com/content_cvpr_2018/html/An_A_PID_Controller_CVPR_2018_paper.html
57. Ma J, Yarats D (2018) Quasi-hyperbolic momentum and Adam for deep learning. CoRR arxiv:1810.06801
58. Liu L, Jiang H, He P, Chen W, Liu X, Gao J, Han J (2019) On the variance of the adaptive learning rate and beyond. CoRR arxiv:1908.03265
59. Graves A (2013) Generating sequences with recurrent neural networks. CoRR arxiv:1308.0850
60. Igel C, Hüsken M (2003) Empirical evaluation of the improved Rprop learning algorithms. Neurocomputing 50:105–123. https://doi.org/10.1016/S0925-2312(01)00700_7
61. Loshchilov I, Hutter F (2017) SGDR: stochastic gradient descent with warm restarts. In: 5th international conference on learning representations, ICLR 2017, Toulon, France, April 24–26, 2017, conference track proceedings. OpenReview.net, Online. https://openreview.net/forum?id=Skq89Scxx
62. Keskar NS, Socher R (2017) Improving generalization performance by switching from Adam to SGD. CoRR arxiv:1712.07628
63. Lessard L, Recht B, Packard A (2016) Analysis and design of optimization algorithms via integral quadratic constraints. SIAM J Optim 26(1):57–95. https://doi.org/10.1137/15M1009597
64. Zaheer M, Reddi SJ, Sachan DS, Kale S, Kumar S (2018) Adaptive methods for nonconvex optimization. In: Bengio S, Wallach HM, Larochelle H, Grauman K, Cesa-Bianchi N, Garnett R (eds.) advances in neural information processing systems 31: annual conference on neural information processing systems 2018, NeurIPS 2018, December 3–8, 2018, Montréal, Canada, pp. 9815–9825. https://proceedings.neurips.cc/paper/2018/hash/90365351ccc7437a1309dc64e4db32a3-Abstract.html
65. Nesterov Y (1983) A method for solving the convex programming problem with convergence rate $o(1/k^2)$. Proc USSR Acad Sci 269:543–547
66. Walkington NJ (2023) Nesterov's method for convex optimization. SIAM Rev 65(2):539–562. https://doi.org/10.1137/21M1390037
67. Jain P, Kakade SM, Kidambi R, Netrapalli P, Sidford A (2018) Accelerating stochastic gradient descent for least squares regression. In: Bubeck S, Perchet V, Rigollet P (eds.) conference on learning theory, COLT 2018, Stockholm, Sweden, 6–9 July 2018. Proceedings of machine learning research, vol. 75, pp. 545–604. PMLR, Online. http://proceedings.mlr.press/v75/jain18a.html
68. Karimi H, Nutini J, Schmidt M (2016) Linear convergence of gradient and proximal-gradient methods under the polyak-łojasiewicz condition. In: Frasconi P, Landwehr N, Manco G, Vreeken J (eds.) machine learning and knowledge discovery in databases: European conference, ECML PKDD 2016, Riva del Garda, Italy, September 19–23, 2016, proceedings, Part I. Lecture notes in computer science, vol. 9851, pp. 795–811. Springer, Cham. https://doi.org/10.1007/978-3-319-46128-1_50
69. Iiduka H (2022) Appropriate learning rates of adaptive learning rate optimization algorithms for training deep neural networks. IEEE Trans Cybern 52(12):13250–13261. https://doi.org/10.1109/TCYB.2021.3107415
70. Défossez A, Bottou L, Bach FR, Usunier N (2022) A simple convergence proof of Adam and Adagrad. Trans. Mach. Learn. Res. 2022
71. Zeiler MD (2012) ADADELTA: an adaptive learning rate method. CoRR arxiv:1212.5701
72. Reddi SJ, Kale S, Kumar S (2019) On the convergence of adam and beyond. CoRR arxiv:1904.09237
73. Liu X, Pan Z, Tao W (2022) Provable convergence of Nesterov's accelerated gradient method for over-parameterized neural networks. Knowl Based Syst 251:109277. https://doi.org/10.1016/J.KNOSYS.2022.109277
74. Gotmare A, Keskar NS, Xiong C, Socher R (2019) A closer look at deep learning heuristics: Learning rate restarts, warmup and distillation. In: 7th international conference on learning representations, ICLR 2019, New Orleans, LA, USA, May 6–9, 2019. OpenReview.net, Online. https://openreview.net/forum?id=r14EOsCqKX
75. Keskar NS, Mudigere D, Nocedal J, Smelyanskiy M, Tang PTP (2017) On large-batch training for deep learning: Generalization gap and sharp minima. In: 5th international conference on learning representations, ICLR 2017, Toulon, France, April 24–26, 2017, conference track proceedings. OpenReview.net, Online. https://openreview.net/forum?id=H1oyRlYgg
76. Wilson AC, Roelofs R, Stern M, Srebro N, Recht B (2017) The marginal value of adaptive gradient methods in machine learning. In: Guyon I, Luxburg U, Bengio S, Wallach HM, Fergus R, Vishwanathan SVN, Garnett R (eds.) advances in neural information processing systems 30: annual conference on neural information processing systems 2017, December 4–9, 2017, Long Beach, CA, USA, pp. 4148–4158. https://proceedings.neurips.cc/paper/2017/hash/81b3833e2504647f9d794f7d7b9bf341-Abstract.html

77. Zhou L, Fan Q, Huang X, Liu Y (2022) Weak and strong convergence analysis of Elman neural networks via weight decay regularization. Optimization 72(9):2287–2309. https://doi.org/10.1080/02331934.2022.2057852
78. Nakamura K, Hong B (2019) Adaptive weight decay for deep neural networks. IEEE Access 7:118857–118865. https://doi.org/10.1109/ACCESS.2019.2937139
79. Hanson SJ, Pratt LY (1988) Comparing biases for minimal network construction with back-propagation. In: Touretzky DS (ed) advances in neural information processing systems 1, [NIPS Conference, Denver, Colorado, USA, 1988]. Morgan Kaufmann, San Mateo, CA, USA, pp 177–185
80. Kaushik P, Khan Z, Kajla A, Verma A, Khan A (2024) Enhancing object recognition with resnet-50: an investigation of the cifar-10 dataset. In: 2023 international conference on smart devices (ICSD), pp. 1–5. IEEE, Piscataway, NJ, USA. https://doi.org/10.1109/icsd60021.2024.10751316
81. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Kopf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S (2019) Pytorch: an imperative style, high-performance deep learning library. In: advances in neural information processing systems 32, pp. 8024–8035. Curran Associates Inc, Red Hook, NY, USA. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
82. Shaiakhmetov R (2024) Crylab/Optimization-performance: Final charts release. Zenodo. https://doi.org/10.5281/zenodo.14534553
83. Rosenbrock HH (1960) An automatic method for finding the greatest or least value of a function. Comput J 3(3):175–184. https://doi.org/10.1093/COMJNL/3.3.175
84. Yu T, Zhu H (2020) Hyper-parameter optimization: a review of algorithms and applications. CoRR arxiv:2003.05689
85. Bergstra J, Bardenet R, Bengio Y, Kégl B (2011) Algorithms for hyper-parameter optimization. In: Shawe-Taylor J, Zemel RS, Bartlett PL, Pereira FCN, Weinberger KQ (eds.) Advances in Neural Information Processing Systems 24: 25th annual conference on neural information processing systems 2011. Proceedings of a meeting Held 12–14 December 2011, Granada, Spain, pp. 2546–2554. https://proceedings.neurips.cc/paper/2011/hash/86e8f7ab32cfd12577bc2619bc635690-Abstract.html
86. Ozaki Y, Tanigaki Y, Watanabe S, Nomura M, Onishi M (2022) Multiobjective tree-structured parzen estimator. J Artif Intell Res 73:1209–1250. https://doi.org/10.1613/JAIR.1.13188
87. Rong G, Li K, Su Y, Tong Z, Liu X, Zhang J, Zhang Y, Li T (2021) Comparison of tree-structured parzen estimator optimization in three typical neural network models for landslide susceptibility assessment. Remote Sens 13(22):4694. https://doi.org/10.3390/RS13224694

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Ruslan Shaiakhmetov[1] · Danilo Pianini[1] · Angelo Filaseta[1] · Gabriele D'Angelo[1] · Valter Venusti[2]

✉ Ruslan Shaiakhmetov
ruslan.shaiakhmetov@unibo.it

Danilo Pianini
danilo.pianini@unibo.it

Angelo Filaseta
angelo.filaseta@unibo.it

Gabriele D'Angelo
g.dangelo@unibo.it

Valter Venusti
v.venusti@dallara.it

[1] Department of Computer Science and Engineering, University of Bologna, Mura Anteo Zamboni 7, 40126 Bologna, BO, Italy

[2] Department of Digital Innovation and Vehicle Electronics Solutions, Dallara Automobili S.p.A., Provinciale 33, 43040 Parma, PR, Italy