Article

Automatic Differentiation-Based Multi-Start for Gradient-Based Optimization Methods

Francesco Della Santa [1,2]

[1] Dipartimento di Scienze Matematiche (DISMA), Politecnico di Torino, Corso Duca degli Abruzzi 24, 10129 Turin, Italy; francesco.dellasanta@polito.it
[2] INdAM-GNCS Research Group, Istituto Nazionale di Alta Matematica, Piazzale Aldo Moro 5, 00185 Rome, Italy

Abstract: In global optimization problems, diversification approaches are often necessary to overcome the convergence toward local optima. One approach is the multi-start method, where a set of different starting configurations are taken into account to designate the best local minimum returned by the multiple optimization procedures as the (possible) global optimum. Therefore, parallelization is crucial for multi-start. In this work, we present a new multi-start approach for gradient-based optimization methods that exploits the reverse Automatic Differentiation to perform efficiently. In particular, for each step, this Automatic Differentiation-based method is able to compute the N gradients of N optimization procedures extremely quickly, exploiting the implicit parallelization guaranteed by the computational graph representation of the multi-start problem. The practical advantages of the proposed method are illustrated by analyzing the time complexity from a theoretical point of view and showing numerical examples where the speed-up is between $\times 40$ and $\times 100$, with respect to classic parallelization methods. Moreover, we show that our AD-based multi-start approach can be implemented by using tailored shallow Neural Networks, taking advantage of the built-in optimization procedures of the Deep Learning frameworks.

Keywords: global optimization; multi-start methods; Automatic Differentiation; Neural Networks

MSC: 90C26; 90C30; 90C35; 65Y20

check for updates

Citation: Della Santa, F. Automatic Differentiation-Based Multi-Start for Gradient-Based Optimization Methods. *Mathematics* **2024**, *12*, 1201. https://doi.org/10.3390/math12081201

Academic Editors: Roberto Alejo Eleuterio, Vicente García and Rosa María Valdovinos-Rosas

Received: 22 March 2024
Revised: 11 April 2024
Accepted: 13 April 2024
Published: 17 April 2024

1. Introduction

Multi-start methods [1–5] are approaches used to increase the possibility of finding a global optimum instead of a local optimum in global optimization problems. This family of methods consists of making comparisons between many local optima obtained by running a chosen optimization procedure with respect to different starting configurations. Alternatively, multi-start is adopted when a set of local optima is desired by the optimization task. For these reasons, multi-start methods can be applied both to unconstrained and constrained optimization, both to derivative-free and derivative-based procedures, and both to meta-heuristic and theoretical-based optimization methods.

In global optimization problems, other approaches include population-based methods, like Genetic Algorithms (see, [6–9]) and Swarm Intelligence-based methods (e.g., Particle Swarm Optimization methods, see [10–12]). These nature-inspired methods are somehow similar to multi-start methods since they are based on a set of starting guesses; however, this set is used as a swarm of interacting agents that move in the domain of the loss function, looking for a global minimizer. Even if these latter methods are very efficient, they suffer a lack of understanding of the convergence properties, and it is not clear how much their efficiency is preserved if applied to large-scale problems [13]. Therefore, in high-dimensional domains, multi-start methods based on optimization procedures with well-known convergence properties can be preferred.

In the literature, sometimes the term multi-start is used to denote general global search methods, e.g., a random search or a re-start method can be defined as multi-start (see [14]), or vice-versa (see [15]). Nonetheless, in this work and according to [5], we denote as multi-start methods only those methods that consist in running the same optimization procedure with respect to a set of $N \in \mathbb{N}$ distinct starting points $\boldsymbol{x}_1^{(0)}, \dots, \boldsymbol{x}_N^{(0)} \in \mathbb{R}^n$. Obviously, this approach can be modified into more specialized and sophisticated methods (e.g., [15–18]), but it is also useful in its basic form; indeed, it is implemented even in the most valuable computational frameworks (e.g., see [19]). However, the main difficulty of using a multi-start approach is that the number N of starting points typically must be quite large, due to the unknown number of local minima. For this reason, parallel computing is very important in this context, and the real exploitation of multi-start seems to be restricted to specific algorithms that are able to take advantage of the computer hardware for parallelization [16,17,20,21]. According to [21,22], three main parallelization approaches can be determined: (*i*) parallelization of the loss function computation and/or the derivative evaluations; (*ii*) parallelization of the numerical methods performing the linear algebra operations; and (*iii*) modifications of the optimization algorithms in a manner that improves the intrinsic parallelism (e.g., the number of parallelizable operations). In this work, we focus on the parallelization schemes of the first case (*i*), specifically on the parallelization of the derivative evaluations, because they are typically used for generating general purpose parallel software [21].

The main drawback of parallelization for multi-start methods is the difficulty of finding a trade-off between efficiency and easy implementation, especially for solving optimization problems of moderate dimensions on non-High Performance Computing (non-HPC) hardware. Typically, the simplest parallelization approach consists in distributing the computations among the available machine workers (e.g., via routines as [23,24]); however, very rarely is this also the most efficient method. Alternatively, a parallel program specifically designed for the optimization problem and for the hardware can be developed, but the time spent in writing this code may not be worth it. Of course, the cost of the parallelization of the derivative evaluations also depends on the computation methods used; when the gradient of the loss function is not available, Finite Differences are typically adopted in literature but, as observed in [22], the advent of Automatic Differentiation in recent decades has presented new interesting possibilities, allowing for the adoption of this technique for gradient computation in optimization procedures (e.g., see [25–29]).

The reverse Automatic Differentiation (AD), see [30] (Ch. 3.2)), was originally developed by Linnainmaa [31] in the 1970s, to be re-discovered by Neural Network researchers (who were not aware of the existence of AD [32]) in the 1980s under the name of backpropagation [33]. Nowadays, reverse AD characterizes almost all the training algorithms for Deep Learning models [32]. AD is a numerical method useful for computing the derivatives of a function, through its representation as an augmented computer program made of a composition of elementary operations (arithmetic operations, basic analytic fuctions, etc.) for which the derivatives are known. Then, using the chain rule of calculus, the method combines these derivatives to compute the overall derivative [32,34]. In particular, AD is divided into two sub-types: forward and reverse AD. The forward AD, developed in the 1960s [35,36], conceptually is the simplest one; it computes the partial derivative of a function $\partial f / \partial x_i$ by recursively applying the chain rule of calculus with respect to the elementary operations of f. On the other hand, reverse AD [31,33] "backwardly" reads the composition of elementary operations constituting the function f; then, still exploiting the chain rule of calculus, it computes the gradient ∇f. Both the AD methods can be extended to vectorial functions $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^m$ and used for computing the Jacobian. Due to the nature of the two AD methods, usually the reverse AD is more efficient if $n > m$ because it can compute the gradient of the j-th output of $\boldsymbol{F}$ at once; otherwise, if $n < m$, the forward AD is the more efficient method because it computes the partial derivatives with respect to x_i for all the outputs of $\boldsymbol{F}$ at once (see [30] (Ch. 3)). The characteristic described above for the reverse AD is the base for the new multi-start method proposed in this work, where we focus on the unconstrained optimization problem of a scalar function $f : \mathbb{R}^n \to \mathbb{R}$.

To improve the efficiency in exploiting the chain rule, the computational frameworks focused on AD (e.g., [37]) are based on computational graphs, i.e., they compile the program as a graph where operations are nodes and data "flow" through the network [38–40]. This particular configuration guarantees high parallelization and high efficiency (both for function evaluations and for AD-based derivative computations). Moreover, the computational graph construction is typically automatic and optimized for the hardware available, keeping the implementation relatively simple.

In this paper, we describe a new efficient multi-start method where the parallelization is not explicitly defined, thanks to the reverse Automatic Differentiation (see [30] (Ch. 3.2)) and the compilation of a computational graph representing the problem [39,40]. The main idea behind the proposed method is to define a function $G : \mathbb{R}^{nN} \to \mathbb{R}$ such that $G(\boldsymbol{x}_1, \dots, \boldsymbol{x}_N) := f(\boldsymbol{x}_1) + \cdots + f(\boldsymbol{x}_N)$, for any set of $N \in \mathbb{N}$ vectors in $\mathbb{R}^n$, where $f : \mathbb{R}^n \to \mathbb{R}$ is the loss function of the optimization problem and N is fixed. Since the gradient of G with respect to $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_N) \in \mathbb{R}^{nN}$ is equivalent to the concatenation of the N gradients $\nabla f(\boldsymbol{x}_1), \dots, \nabla f(\boldsymbol{x}_N)$, by applying the reverse AD on G we can compute the N gradients of the loss function very efficiently; therefore, we can implicitly and efficiently parallelize a multi-start method of N procedures running one AD-based optimization procedure for the function G. The main advantage of this approach is the good trade-off between efficiency and easy implementation. Indeed, nowadays the AD frameworks (e.g., [37]) compile functions and programs as computational graphs that automatically but very efficiently exploit the available hardware; then, with the proposed method, the user just needs to define the function G and the differentiation through the reverse AD, obtaining a multi-start optimization procedure that (in general) is more time-efficient than a direct parallelization of the processes, especially excluding an HPC context. Moreover, the method can be extended, implementing it by using tailored shallow Neural Networks and taking advantage of the built-in optimization procedures of the Deep Learning frameworks.

The work is organized as follows. In Section 2, we briefly recall and describe the AD method. In Section 3, we start introducing a new formulation of the multi-start problem that is useful for the exploitation of the reverse AD, and the time complexity estimations are illustrated. Then, we show numerical results illustrating a context where the proposed method is advantageous with respect to a classic parallelization. In Section 4, we show how to implement the AD-based multi-start approach using a tailored shallow Neural Network (NN) in those cases where the user wants to take advantage of the optimization procedures available in most of the Deep Learning frameworks. In particular, we illustrate an example where the new multi-start method is implemented as a shallow NN and used to find three level set curves of a given function. Finally, some conclusions are drawn in Section 5.

2. Automatic Differentiation

Automatic Differentiation (AD) is a numerical method to compute the exact derivatives of a function, representing it as an augmented computer program made of a composition of elementary operations for which the derivatives are known [30] (Ch. 3.2); typically, this program is represented as a graph connecting the elementary operations (e.g., see Figure 1).

The first idea about AD dates back to the 1950s [35], and the *forward mode AD* was developed in the 1960s [36]. In the 1970s, Linnainmaa developed the *reverse mode AD* [31]. This latter method is equivalent to the one independently developed by NN researchers in the late 1980s, with the name of backpropagation [33]. Nowadays, backpropagation is recognized as the application of the reverse mode AD for derivatives computation in NNs.

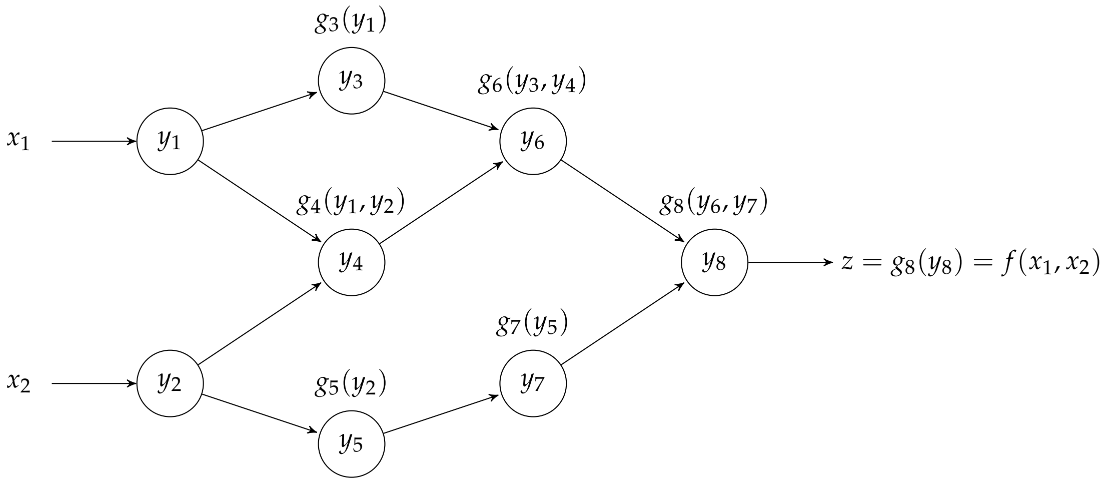

Figure 1. Example of computational graph representing $f(x_1, x_2)$ as a composition of elementary functions $g_3, \ldots, g_8$.

2.1. Forward Mode AD

The forward AD is conceptually the simplest of the two methods. Given a scalar function $f : \mathbb{R}^n \to \mathbb{R}$, this method consists of a recursive application of the chain rule of calculus for computing the partial derivative $\partial f / \partial x_i$, for a fixed $i \in \{1, \ldots, n\}$. More specifically, let us us assume that f is the result of a composition of M elementary operations with known derivatives. Let us denote with y_j the intermediate variables such that $y_j = x_j$, for each $1 \leq j \leq n$, and y_j is the result of the elementary operation g_j applied to a subset of "previous" intermediate variables, for each $j = n+1, \ldots, n+M$; for the ease of notation, we use the same index j both for the elementary operation g_j and the intermediate variable y_j returned by it (see Figure 1).

Now, let $x_i \in \{x_1, \ldots, x_n\}$ be a fixed input variable and $\dot{y}_j := \partial y_j / \partial x_i$. Then, it holds that

$$\dot{y}_j = \begin{cases} \delta_{ij}\,, & \forall\, j = 1, \ldots, n \\ \sum_{y_k \text{ parent of } y_j} \frac{\partial y_j}{\partial y_k}\, \dot{y}_k\,, & \forall\, j > n \end{cases}\,, \tag{1}$$

where the term "parent of" refers to the computational graph of f (see Figure 1). Then, we can build another "composition graph" with the same connectivity of the one of f but with elementary operations given by (1). Therefore, given a vector $\boldsymbol{x}^* \in \mathbb{R}^n$, we can compute the value of the derivative $\partial f(\boldsymbol{x}^*) / \partial x_i$ with the following steps:

1. We compute $f(\boldsymbol{x}^*)$, and we store all the intermediate variable values involved in the computations of the second "forward AD graph";
2. We compute the output value of the forward AD graph, giving as input (a.k.a. *seed*) the canonical basis vector $\boldsymbol{e}_i$ (i.e., selecting x_i as input variable for the partial derivatives).

2.2. Reverse Mode AD

The reverse mode of AD "backwardly" reads the connections of the composition graph of the function to exploit the differentiation dependencies. This procedure is somehow equivalent to "exploding" the partial derivatives $\partial z / \partial x_i$ and identifying their elementary functions. Let us introduce the following quantities:

$$\widetilde{y}_i := \sum_{y_j \text{ child of } y_i} \frac{\partial y_i}{\partial y_j}\, \widetilde{y}_j \quad \text{and} \quad \widetilde{z} := 1\,. \tag{2}$$

Then , for each $i = 1, \ldots, n$, the following holds:

$$\widetilde{y}_i = \frac{\partial z}{\partial x_i}\,.$$

It is easy to observe that the differentiation dependencies written in (2) are evident when reading the computational graph of f (see Figure 1) from right to left. Then, the reverse mode AD consists of building another composition graph that has the same connectivity but is executed in the opposite way and where the elementary operations are given by (2). If we consider the example function of Figure 1, we can build the "reverse AD graph" illustrated in Figure 2.

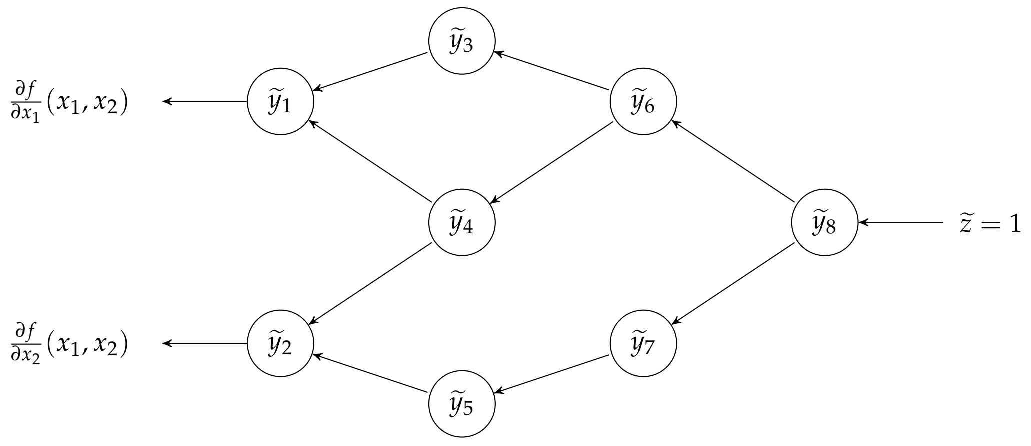

Figure 2. Example of reverse AD computational graph for f as in Figure 1.

Then, given a vector $\boldsymbol{x}^* \in \mathbb{R}^n$, we can compute the value of the entire gradient $\nabla f(\boldsymbol{x}^*)$ with the following steps:

1. We compute $f(\boldsymbol{x}^*)$, and we store all the intermediate variable values involved in the computations of the reverse AD graph;
2. We compute the output values of the function represented by the reverse AD graph, giving as the seed the scalar value equal to 1 (i.e., $\widetilde{z} = 1$).

2.3. Automatic Differentiation for Jacobians

AD can be easily extended to vectorial functions $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^m$ [30] (Ch. 3.2). In particular, given a vector $\boldsymbol{x}^* \in \mathbb{R}^n$, the Jacobian matrix $J_{\boldsymbol{F}}(\boldsymbol{x}^*) \in \mathbb{R}^{m\times n}$ at $\boldsymbol{x}^*$ can be computed running m times the reverse AD, using the m canonical basis vectors $\boldsymbol{e}_1, \dots, \boldsymbol{e}_m \in \mathbb{R}^m$ as seeds to the reverse AD graph; alternatively, it can be computed n times running the forward AD, using the n canonical basis vectors $\boldsymbol{e}_1, \dots, \boldsymbol{e}_n \in \mathbb{R}^n$ as seeds to the forward AD graph. In other words, the reverse AD graph with input vector $\boldsymbol{e}_j$ returns the j-th row of the Jacobian matrix, while the forward AD graph with input vector $\boldsymbol{e}_i$ returns the i-th column of the Jacobian matrix. For these reasons, in general the reverse AD is more efficient if $n > m$; otherwise, if $n < m$, the forward AD is the more efficient method.

Focusing on scalar functions, it is trivial to observe that we can compute the gradient running once the reverse AD instead of running the forward one n times.

3. Reverse Automatic Differentiation for Multi-Start

Let us consider the unconstrained optimization problem

$$\min_{\boldsymbol{x}\in\mathbb{R}^n} f(\boldsymbol{x})\,, \tag{3}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a given function, and let us approach the problem with a gradient-based optimization method (e.g., the steepest descent). Moreover, we assume f is the composition of elementary operations so that it is possible to use the reverse AD to compute its gradient.

The main idea behind the reverse AD-based multi-start method is to define a function $G : \mathbb{R}^{nN} \to \mathbb{R}$ such that it is a linear combination of loss function evaluations at N vectors, i.e.,

$$G([\boldsymbol{x}_1^T, \dots, \boldsymbol{x}_N^T]^T) := \lambda_1 f(\boldsymbol{x}_1) + \cdots + \lambda_N f(\boldsymbol{x}_N)\,,$$

for each $\boldsymbol{x}_i \in \mathbb{R}^n$, $\lambda_i \in \mathbb{R}^+$ fixed, $i = 1, \dots, N$. Then, for any set of points $\{\widehat{\boldsymbol{x}}_1, \dots, \widehat{\boldsymbol{x}}_N\} \subset \mathbb{R}^n$ where f is differentiable, the gradient of G at $\widehat{\boldsymbol{\xi}} = [\widehat{\boldsymbol{x}}_1^T, \dots, \widehat{\boldsymbol{x}}_N^T]^T$ is equal to the concatenation of the vectors $\lambda_1 \nabla f(\widehat{\boldsymbol{x}}_1), \dots, \lambda_N \nabla f(\widehat{\boldsymbol{x}}_N)$. Therefore, using the reverse AD to compute the gradient $\nabla_{\boldsymbol{\xi}} G(\widehat{\boldsymbol{\xi}})$, we obtain a very fast method to evaluate the N exact gradients $\nabla f(\widehat{\boldsymbol{x}}_1), \dots, \nabla f(\widehat{\boldsymbol{x}}_N)$, where $\boldsymbol{\xi}$ denotes the domain variable of G in $\mathbb{R}^{nN}$. Therefore, we can apply the steepest descent method for G, which actually corresponds to applying N times the steepest descent methods to f. In the following, we formalize this idea.

Definition 1 (N-concatenation). *For each fixed $N \in \mathbb{N}$, for any function $\boldsymbol{f} : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}^m$, a function $\boldsymbol{F} : \Omega^N \subseteq \mathbb{R}^{nN} \to \mathbb{R}^{mN}$ is an* N-concatenation *of $\boldsymbol{f}$ if*

$$\boldsymbol{F}([\boldsymbol{x}_1^T, \dots, \boldsymbol{x}_N^T]^T) = [\boldsymbol{f}(\boldsymbol{x}_1)^T, \dots, \boldsymbol{f}(\boldsymbol{x}_N)^T]^T, \tag{4}$$

for each set of N vectors $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_N\} \in \Omega$.

Notation 1. *For the sake of simplicity, from now on vectors $[\boldsymbol{x}_1^T, \dots, \boldsymbol{x}_N^T]^T \in \mathbb{R}^{nN}$ will be denoted by $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_N)$. Analogously, vectors returned by an N-concatenation of a function $\boldsymbol{f}$ will be denoted by $(\boldsymbol{f}(\boldsymbol{x}_1), \dots, \boldsymbol{f}(\boldsymbol{x}_N))$.*

Definition 2 ($\boldsymbol{\lambda}$-concatenation). *For each fixed $\boldsymbol{\lambda} \in \mathbb{R}^N$, for any function $\boldsymbol{f} : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}^m$, a function $\boldsymbol{F} : \Omega^N \subseteq \mathbb{R}^{nN} \to \mathbb{R}^{mN}$ is a* $\boldsymbol{\lambda}$-concatenation *of $\boldsymbol{f}$ if*

$$\boldsymbol{F}((\boldsymbol{x}_1, \dots, \boldsymbol{x}_N)) = (\lambda_1 \boldsymbol{f}(\boldsymbol{x}_1), \dots, \lambda_N \boldsymbol{f}(\boldsymbol{x}_N)), \tag{5}$$

for each set of N vectors $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_N\} \in \Omega$.

Remark 1. *Obviously, an N-concatenation of $\boldsymbol{f}$ is a $\boldsymbol{\lambda}$-concatenation of $\boldsymbol{f}$ with $\boldsymbol{\lambda} = \boldsymbol{e} = [1, \dots, 1]^T \in \mathbb{R}^N$.*

Given these definitions, the idea for the new multi-start method is based on the observation that the steepest descent method for G, starting from $\boldsymbol{\xi}^{(0)}$ and with step-length factor $\alpha \in \mathbb{R}^+$, represented by

$$\begin{cases} \boldsymbol{\xi}^{(0)} = (\boldsymbol{x}_1^{(0)}, \dots, \boldsymbol{x}_N^{(0)}) \in \mathbb{R}^{nN} \\ \boldsymbol{\xi}^{(k+1)} = \boldsymbol{\xi}^{(k)} - \alpha \, \nabla_{\boldsymbol{\xi}} G(\boldsymbol{\xi}^{(k)}), \quad \forall\, k \in \mathbb{N} \end{cases}, \tag{6}$$

is equivalent to

$$\begin{bmatrix} \boldsymbol{x}_1^{(k+1)} \\ \vdots \\ \boldsymbol{x}_N^{(k+1)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_1^{(k)} \\ \vdots \\ \boldsymbol{x}_N^{(k)} \end{bmatrix} - \begin{bmatrix} \alpha \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_1^{(k)}) \\ \vdots \\ \alpha \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_N^{(k)}) \end{bmatrix}, \tag{7}$$

if the gradient of G is an N-concatenation of the gradients of f, i.e,

$$\nabla_{\boldsymbol{\xi}} G((\boldsymbol{x}_1, \dots, \boldsymbol{x}_N)) = (\nabla_{\boldsymbol{x}} f(\boldsymbol{x}_1), \dots, \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_N)). \tag{8}$$

Analogously, if the gradient of G is a $\boldsymbol{\lambda}$-concatenation of the gradients of f, with $\boldsymbol{\lambda} \in \mathbb{R}^N$, $\boldsymbol{\lambda} > \boldsymbol{0}$, then (6) is equivalent to

$$\begin{bmatrix} \boldsymbol{x}_1^{(k+1)} \\ \vdots \\ \boldsymbol{x}_N^{(k+1)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_1^{(k)} \\ \vdots \\ \boldsymbol{x}_N^{(k)} \end{bmatrix} - \begin{bmatrix} \alpha \lambda_1 \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_1^{(k)}) \\ \vdots \\ \alpha \lambda_N \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_N^{(k)}) \end{bmatrix}, \tag{9}$$

i.e., is equivalent to applying N times the steepest descent methods to f, with step-length factors $\alpha\lambda_1, \dots, \alpha\lambda_N$.

Remark 2 (**Extension to other gradient-based methods**). *It is easy to see that we can generalize these observations to other gradient-based optimization methods than the steepest descent (e.g., momentum methods [41,42]). Let $\mathcal{M}$ be the function characterizing the iterations of a given gradient-based method, i.e, such that*

$$\boldsymbol{x}^{(k+1)} = \mathcal{M}(\boldsymbol{x}^{(k)}, \nabla g(\boldsymbol{x}^{(k)})\,;\alpha) = \begin{bmatrix} \mathcal{M}(x_1^{(k)}, g_{x_1}(\boldsymbol{x}^{(k)})\,;\alpha) \\ \vdots \\ \mathcal{M}(x_n^{(k)}, g_{x_n}(\boldsymbol{x}^{(k)})\,;\alpha) \end{bmatrix}, \tag{10}$$

for each $n \in \mathbb{N}$, each step-length $\alpha \in \mathbb{R}^+$ and each objective function $g : \mathbb{R}^n \to \mathbb{R}$. Then, the iterative process for G, starting from $\boldsymbol{\xi}^{(0)}$, and with respect to a gradient-based method characterized by $\mathcal{M}$ is

$$\begin{cases} \boldsymbol{\xi}^{(0)} = (\boldsymbol{x}_1^{(0)}, \dots, \boldsymbol{x}_N^{(0)}) \in \mathbb{R}^{nN} \\ \boldsymbol{\xi}^{(k+1)} = \mathcal{M}\left(\boldsymbol{\xi}^{(k)}, \nabla_{\boldsymbol{\xi}} G(\boldsymbol{\xi}^{(k)})\,;\alpha\right), \quad \forall\, k \in \mathbb{N} \end{cases}, \tag{11}$$

that is equivalent to the multi-start approach with respect to f:

$$\begin{bmatrix} \boldsymbol{x}_1^{(k+1)} \\ \vdots \\ \boldsymbol{x}_N^{(k+1)} \end{bmatrix} = \begin{bmatrix} \mathcal{M}\left(\boldsymbol{x}_1^{(k)}, \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_1^{(k)})\,;\alpha\right) \\ \vdots \\ \mathcal{M}\left(\boldsymbol{x}_N^{(k)}, \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_N^{(k)})\,;\alpha\right) \end{bmatrix}. \tag{12}$$

Moreover, we can further extend the generalization if we assume that, for each $\lambda \in \mathbb{R}^+$ and each $\alpha \in \mathbb{R}^+$, the following holds:

$$\mathcal{M}(\boldsymbol{x}^{(k)}, \lambda \nabla g(\boldsymbol{x}^{(k)});\alpha) = \mathcal{M}(\boldsymbol{x}^{(k)}, \nabla g(\boldsymbol{x}^{(k)});m(\lambda)\alpha)\,, \tag{13}$$

where $m : \mathbb{R}^+ \to \mathbb{R}^+$ is a fixed function. Indeed, if the gradient of G is a $\boldsymbol{\lambda}$-concatenation of the gradients of f, Equation (12) changes into

$$\begin{bmatrix} \boldsymbol{x}_1^{(k+1)} \\ \vdots \\ \boldsymbol{x}_N^{(k+1)} \end{bmatrix} = \begin{bmatrix} \mathcal{M}\left(\boldsymbol{x}_1^{(k)}, \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_1^{(k)})\,;\alpha_1\right) \\ \vdots \\ \mathcal{M}\left(\boldsymbol{x}_N^{(k)}, \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_N^{(k)})\,;\alpha_N\right) \end{bmatrix}, \tag{14}$$

where $\alpha_i := m(\lambda_i)\alpha$, for each $i = 1, \dots, N$.

3.1. Using the Reverse Automatic Differentiation

To actually exploit the reverse AD for a multi-start steepest descent (or another gradient-based method), we need to define a function G with gradient a $\boldsymbol{\lambda}$-concatenation of the gradients of f, the objective function of problem (3).

Proposition 1. *Let us consider a function $f : \mathbb{R}^n \to \mathbb{R}$. Let $\boldsymbol{F} : \mathbb{R}^{nN} \to \mathbb{R}^N$ be an N-concatenation of f, for a fixed $N \in \mathbb{N}$, and let $L_{\boldsymbol{\lambda}} : \mathbb{R}^N \to \mathbb{R}$ be the linear function*

$$L_{\boldsymbol{\lambda}}(\boldsymbol{y}) := \boldsymbol{\lambda}^T \boldsymbol{y} = \sum_{i=1}^{N} \lambda_i y_i\,, \tag{15}$$

for a fixed $\boldsymbol{\lambda} \in \mathbb{R}^N$, $\boldsymbol{\lambda} > \boldsymbol{0}$.

Then, for each set of $N \in \mathbb{N}$ points where f is differentiable, the gradient of the function $G := L_{\boldsymbol{\lambda}} \circ \boldsymbol{F}$ is a $\boldsymbol{\lambda}$-concatenation of the gradient of f.

Proof. The proof is straightforward since

$$\nabla_{\xi} L_{\lambda}(\boldsymbol{F}((\boldsymbol{x}_1,\dots,\boldsymbol{x}_N))) = \nabla_{\xi}\left(\sum_{i=1}^{N} \lambda_i f(\boldsymbol{x}_i)\right) = \begin{bmatrix} \lambda_1 \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_1) \\ \vdots \\ \lambda_N \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_N) \end{bmatrix},$$

for each set of $N \in \mathbb{N}$ points $\{\boldsymbol{x}_1,\dots,\boldsymbol{x}_N\} \subset \mathbb{R}^n$ where f is differentiable. □

Assuming that the expression of $\nabla_{\boldsymbol{x}} f$ is unknown, a formulation such as (14) obtained from $G = L_{\lambda} \circ \boldsymbol{F}$ seems to give no advantages without AD and/or tailored parallelization. Indeed, the computation of the gradient $\nabla_{\xi} G$ through classical gradient approximation methods (e.g., Finite Differences) needs N different gradient evaluations for each point $\boldsymbol{x}_1^{(k)},\dots,\boldsymbol{x}_N^{(k)}$, respectively, and each one of them needs $O(n)$ function evaluations (assuming no special structures for f); then, e.g., the time complexity of the gradient approximation with Finite Differences for G (denoted by $\nabla_{\xi}^{\mathrm{FD}} G$) is

$$\mathrm{T}(\nabla_{\xi}^{\mathrm{FD}} G) = N \cdot \mathrm{T}(\nabla_{\boldsymbol{x}}^{\mathrm{FD}} f) = O(nN \cdot \mathrm{T}(f)), \tag{16}$$

where $\mathrm{T}(\,\cdot\,)$ denotes the time complexity.

On the other hand, reverse AD permits the efficient operation of the gradient-based method (11), equivalent to the N gradient-based methods, even for large values of n and N. Indeed, from [30] (Ch. 3.3), for any function $g : \mathbb{R}^n \to \mathbb{R}$, it holds that the time complexity of a gradient evaluation of g with reverse AD (in a point where the operation is defined) is such that

$$\mathrm{T}(\nabla^{\mathrm{AD}} g) \leq 4 \cdot \mathrm{T}(g), \tag{17}$$

where ∇^{AD} denotes the gradient computed with reverse AD. The following lemma characterizes the relationship between the time complexity of $\nabla_{\xi}^{\mathrm{AD}} G$ and the time complexity of f.

Lemma 1. *Let f, $\boldsymbol{F}$, and L_{λ} be as in Proposition 1. Let $G : \mathbb{R}^{nN} \to \mathbb{R}$ be such that $G := L_{\lambda} \circ \boldsymbol{F}$, and let us assume that*

$$\mathrm{T}(L_{\lambda} \circ \boldsymbol{F}) = O(N \cdot \mathrm{T}(f)). \tag{18}$$

Then, the time complexity of $\nabla_{\xi}^{\mathrm{AD}} G$ is

$$\mathrm{T}(\nabla_{\xi}^{\mathrm{AD}} G) = O(N \cdot \mathrm{T}(f)). \tag{19}$$

Proof. The proof is straightforward, due to (17) and (18). □

Comparing (19) and the time complexity (16) of the Finite Differences gradient approximation, we observe that reverse AD is much more convenient both because it computes the exact gradient and because of the time complexity. In the following example, we illustrate the concrete efficiency of using $\nabla_{\xi}^{\mathrm{AD}} G$ to compute N gradients of the n-dimensional Rosenbrock function, assuming its gradient is unknown.

Example 1 (Reverse AD and the n-dimensional Rosenbrock function). *Let $f : \mathbb{R}^n \to \mathbb{R}$ be the n-dimensional Rosenbrock function [43–45]:*

$$f(\boldsymbol{x}) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2. \tag{20}$$

Assuming a good implementation of f, we have that the time complexity of the function is $\mathrm{T}(f) = O(\log n)$. Then, it holds that $\mathrm{T}(\nabla_{\xi}^{\mathrm{AD}} G) = O(N \log n)$. In this particular case, we observe that the coefficient $C \in \mathbb{R}^+$, such that

$$\mathrm{T}(\nabla_{\xi}^{\mathrm{AD}} G) \leq C N \log n$$

for sufficiently large N and n is very small; specifically, for the given example, we have $C < 10^{-4}$*; moreover, we observe that the reverse AD can compute* $N = 2^{12} = 4096$ *gradients of* f *in a space of dimension* $n = 2^{12} = 4096$ *in less than one second (see Figure 3). All the computations are performed using a Notebook PC with Intel Core i5 Processor dual-core (2.3 GHz) and 16 GB DDR3 RAM.*

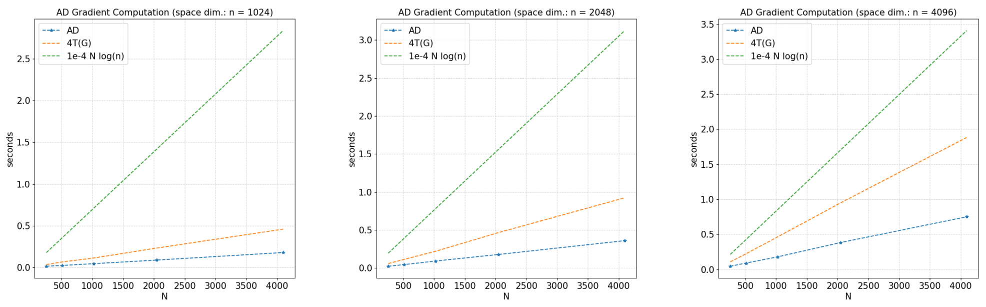

Figure 3. Average computation time of the reverse AD method (blue curves) with respect to four times the average computation time of G (orange curves) and the curve $10^{-4} N \log n$, for each fixed $n = 2^{10}, 2^{11}, 2^{12}$ (from left to right).

3.2. An Insight into AD and Computational Graphs

Equation (19) of Lemma 1 tells us that the time complexity of computing N times the gradient $\nabla_{\boldsymbol{x}}^{\mathrm{AD}} f$ is $O(N \cdot \mathrm{T}(f))$; then, with a proper parallelization procedure of these N gradient computations, the construction of G seems to be meaningless. However, the special implementation of G as a computational graph, made by the frameworks used for AD (see Section 1), guarantees an extremely efficient parallelization, both for the evaluation of G and the computation of $\nabla_{\xi}^{\mathrm{AD}} G$. For example, during the graph compilation, the framework TensorFlow [37] identifies separate sub-parts of the computation that are independent and splits them [40], optimizing the code execution “aggressively” [39]; in particular, this parallelization does not consist only in sending sub-parts to different machine workers but also in optimizing the data reading procedure such that each worker can perform more operations almost at the same time (similarly to implementations that exploit vectorization [46]). Then, excluding a High-Performance Computing (HPC) context, the computational graphs representing G and $\nabla_{\xi}^{\mathrm{AD}} G$ make the AD-based multi-start method (almost always) more time-efficient than a multi-start method where the code explicitly parallelizes the N gradient computations of $\nabla_{\boldsymbol{x}}^{\mathrm{AD}} f$ or the N optimization procedures (see Section 3.3 below); indeed, in the parallelized multi-start case, each worker can compute only one gradient/procedure at a time, and the number of workers is typically less than N. Moreover, the particular structure of the available frameworks makes the implementation of the AD-based method very easy.

As written above, the time-efficiency properties of a computational graph are somehow similar to the ones given by the vectorization of an operation; we recall that the vectorization (e.g., see [46]) of a function $f : \mathbb{R}^n \to \mathbb{R}$ is the code implementation of f as a function $\widetilde{\boldsymbol{F}} : \bigcup_{N=1}^{\infty} \mathbb{R}^{N \times n} \to \bigcup_{N=1}^{\infty} \mathbb{R}^N$ such that, without the use of for/while-cycles, the following holds:

$$\widetilde{\boldsymbol{F}}(X) = (f(\boldsymbol{x}_1), \dots, f(\boldsymbol{x}_N)),$$

for each matrix $X \in \mathbb{R}^{N \times n}$, $N \in \mathbb{N}$, and where $\boldsymbol{x}_i$ denotes the transposed i-th row of X. In this case, especially when $N \gg 1$, using $\widetilde{\boldsymbol{F}}$ is much more time-efficient than parallelizing N computations of f, thanks to the particular data structure read by the machine workers, which is optimized according to memory allocation, data access, and workload reduction.

Therefore, we easily deduce that the maximum efficiency for the AD-based method is obtained when G is built using $\widetilde{\boldsymbol{F}}$ instead of $\boldsymbol{F}$ (if possible).

Summarizing the content of this subsection, the proposed AD-based multi-start method exploits formulation (6) to build an easy and handy solution for an implicit and highly efficient parallelization of the procedure, assuming access to AD frameworks.

3.3. Numerical Experiments

In this section, we illustrate the results of a series of numerical experiments that compare the computational costs (in time) of three multi-start methods for problem (3):

(*i*) AD-based multi-start steepest descent, exploiting vectorization (see Section 3.2);
(*ii*) AD-based multi-start steepest descent, without exploiting vectorization;
(*iii*) Parallel multi-start steepest descent, distributing the N optimization procedures among all the available workers. In this case, the gradient of f is computed using the reverse AD, for a fair comparison with case (*i*) and case (*ii*).

The experiments have been performed using a Notebook PC with Intel Core i5 Processor dual-core (2.3 GHz) and 16 GB DDR3 RAM (see Example 1); in particular, each multi-start method has been executed alone, as the unique running application on the PC (excluded mandatory applications in the background). The methods are implemented in Python, using the TensorFlow module for reverse AD; the parallelization of method (*iii*) is based on the *multiprocessing* built-in module [24].

Since we want to analyze the time complexity behavior of the three methods varying the domain dimension n and the number of starting points N, we perform the experiments considering the Rosenbrock function (see (20), Example 1). This function is particularly suitable for our analyses since its expression is defined for any dimension $n \geq 2$, and it has always a local minimum in $\boldsymbol{e} := \sum_{i=1}^{n} \boldsymbol{e}_i$.

According to the different natures of the three methods considered, the implementation of the Rosenbrock function changes. In particular, for method (*i*), in the AD framework we implement a function G that highly exploit vectorization, avoiding any for-cycle (see Algorithm 1); for method (*ii*), we implement G cycling among the set of N vectors with respect to which we must compute the function (see Algorithm 2); for method (*iii*), we do not implement G, only the Rosenbrock function (see Algorithm 3).

Algorithm 1 G-Rosenbrock implementation—Method (*i*)

```
Input:
    $X$, matrix of $n \in \mathbb{N}$ columns, $N \in \mathbb{N}$ of rows;
Output:
    $y$, output scalar value of $G$ built with respect to (20).
1: $X_{\cdot,2:n} \leftarrow$ submatrix of $X$ given by all the columns, except the first one;
2: $X_{\cdot,1:n-1} \leftarrow$ submatrix of $X$ given by all the columns, except the last one;
3: $Y \leftarrow 100 \times (X_{\cdot,2:n} - X_{\cdot,1:n-1} \,\hat{}\, 2) \,\hat{}\, 2 + (1 - X_{\cdot,1:n-1}) \,\hat{}\, 2$
4: $y \leftarrow$ sum-up all the values in $Y$
5: return : $y$
```

We test the three methods varying all the combinations of N and n, with $N \in \{20 \cdot i \,:\, i = 1, \dots 10\}$ and $n \in \{20 \cdot i \,:\, i = 1, \dots 5\}$, for a total number of 50 experiments. The parameters of the steepest descent methods are fixed: $\alpha = 0.01$, maximum number of iterations $k_{\max} = 10^4$, stopping criterium's tolerance $\tau = 10^{-6}$ for the gradient's norm, and fixed starting points $\boldsymbol{x}_1^{(0)}, \dots, \boldsymbol{x}_N^{(0)}$ sampled from the uniform distribution $\mathcal{U}([-2,3]^n)$; computations are performed in single precision, the default precision of TensorFlow. The choice of a small value for α, together with the sampled starting points, and the "flat" behavior of the function near to the minimum $\boldsymbol{e}$ ensure that the steepest descent methods always converge toward $\boldsymbol{e}$ using all the iterations allowed, i.e., $\boldsymbol{x}_i^{(k_{\max})} \simeq \boldsymbol{e}$ but $\| \nabla f(\boldsymbol{x}_i^{(k_{\max})}) \| > \tau = 10^{-6}$, for each $i = 1, \dots, N$.

Algorithm 2 *G*-Rosenbrock implementation—Method (*ii*)

```
Input:
        $X$, matrix of $n \in \mathbb{N}$ columns, $N \in \mathbb{N}$ of rows;
Output:
        $y$, output scalar value of $G$ built with respect to (20).
1: $y \leftarrow 0$
2: for $i = 1, \dots, N$ do
3:     $\boldsymbol{x}_{1:n-1} \leftarrow$ sub-row $X_{i,1:n-1}$
4:     $\boldsymbol{x}_{2:n} \leftarrow$ sub-row $X_{i,2:n}$
5:     $\boldsymbol{y}_i \leftarrow 100 \times (\boldsymbol{x}_{2:n} - \boldsymbol{x}_{1:n-1} \hat{} 2) \hat{} 2 + (1 - \boldsymbol{x}_{1:n-1}) \hat{} 2$
6:     $y_i \leftarrow$ sum up values in $\boldsymbol{y}_i$
7:     $y \leftarrow y + y_i$
8: end for
9: return : $y$
```

Algorithm 3 Rosenbrock implementation—Method (*iii*)

```
Input:
        $x$, vector in $\mathbb{R}^n$;
Output:
        $y$, output scalar value of (20).
1: $\boldsymbol{x}_{1:n-1} \leftarrow$ sub-row $X_{i,1:n-1}$
2: $\boldsymbol{x}_{2:n} \leftarrow$ sub-row $X_{i,2:n}$
3: $\boldsymbol{y} \leftarrow 100 \times (\boldsymbol{x}_{2:n} - \boldsymbol{x}_{1:n-1} \hat{} 2) \hat{} 2 + (1 - \boldsymbol{x}_{1:n-1}) \hat{} 2$
4: $y \leftarrow$ sum-up values in $\boldsymbol{y}$
5: return : $y$
```

Looking at the computation times of the methods, reported in Tables 1–3 and illustrated in Figure 4, the advantage of using the AD-based methods is evident, in particular when combined with vectorization; indeed, we observe that in its slowest case, $(N, n) = (200, 100)$, method (i) is still faster than the fastest case, $(N, n) = (20, 20)$, of both method (ii) and method (iii).

Table 1. Method (*i*). AD-based multi-start plus vectorization. Computation times w.r.t. N starting points in $\mathbb{R}^n$, $k = k_{\max} = 10^4$ iterations. Time notation: *minutes:seconds.decimals* .

$n \backslash N$	20	40	60	80	100	120	140	160	180	200
20	0:10.6	0:10.9	0:10.9	0:11.5	0:11.6	0:11.8	0:12.2	0:12.6	0:12.9	0:14.2
40	0:10.9	0:11.5	0:12.4	0:12.5	0:12.9	0:13.3	0:14.0	0:14.4	0:15.2	0:17.3
60	0:11.2	0:12.1	0:13.0	0:13.5	0:14.4	0:15.1	0:16.6	0:17.2	0:17.7	0:18.7
80	0:11.4	0:12.6	0:13.3	0:14.5	0:15.9	0:17.1	0:18.2	0:20.0	0:21.7	0:22.6
100	0:11.9	0:12.8	0:14.5	0:16.1	0:17.2	0:19.6	0:20.4	0:21.9	0:23.2	0:25.2

The linear behaviour of the methods' time complexity with respect to N is confirmed (see Figure 4); however, the parallel multi-start method's behavior is afflicted by higher noise, probably caused by the machine's management of the jobs' queue. Nonetheless, the linear behaviors of the methods are clearly different: the AD-based multi-start methods are shown to be faster than the parallel multi-start method, with a speed-up factor between $\times 3$ and $\times 6$ (method (ii)) and between $\times 40$ and $\times 100$ (method (i)).

Table 2. Method (*ii*). AD-based multi-start, no vectorization. Computation times w.r.t. N starting points in $\mathbb{R}^n$, $k = k_{\max} = 10^4$ iterations. Time notation: *minutes:seconds.decimals*.

$n \backslash N$	20	40	60	80	100	120	140	160	180	200
20	0:37.7	1:06.7	1:35.4	2:02.9	2:37.2	3:02.0	3:33.5	4:16.2	4:29.7	5:15.2
40	0:38.1	1:07.3	1:36.1	2:10.4	2:37.3	3:02.8	3:38.0	4:15.8	4:37.8	5:16.1
60	0:39.3	1:08.0	1:39.4	2:10.9	2:40.3	3:09.0	3:56.5	4:34.9	5:14.1	5:56.4
80	0:38.5	1:07.7	1:38.3	2:12.5	2:42.5	3:28.9	4:05.7	4:31.4	5:08.1	5:51.4
100	0:38.9	1:10.6	1:41.4	2:13.7	2:57.8	3:31.8	4:07.4	4:42.1	5:19.7	6:00.8

Table 3. Method (*iii*). Parallel multi-start (four workers). Computation times w.r.t. N starting points in $\mathbb{R}^n$, $k = k_{\max} = 10^4$ iterations. Time notation: *minutes:seconds.decimals*.

$n \backslash N$	20	40	60	80	100	120	140	160	180	200
20	1:46.1	3:35.9	5:27.8	7:28.1	9:39.6	11:53.3	15:05.5	18:49.0	34:18.6	39:07.0
40	1:47.0	3:38.3	5:29.8	7:28.5	9:50.1	11:52.8	14:41.4	17:41.9	29:21.4	43:09.4
60	1:47.8	3:39.7	5:31.2	7:33.1	9:41.6	11:55.7	14:36.4	18:14.4	33:42.5	38:36.3
80	1:47.0	3:39.9	5:30.8	7:32.2	9:41.9	11:58.2	14:35.8	17:46.1	35:09.3	19:08.5
100	1:46.5	3:40.5	5:25.6	7:32.4	9:43.4	12:09.6	14:59.0	15:11.1	17:14.2	19:15.3

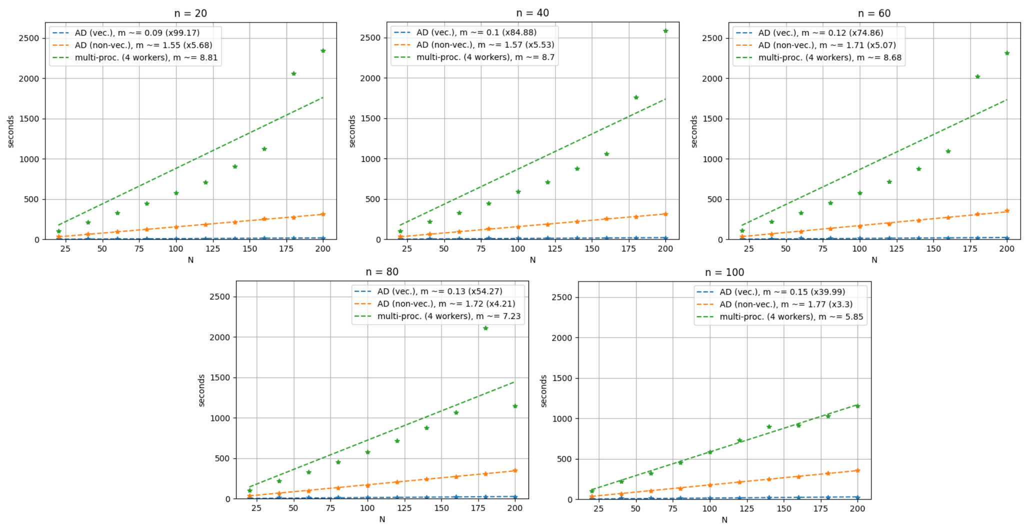

Figure 4. Cases $n = 20, \ldots, 100$, from left to right, from top to bottom. Computation times w.r.t. N starting points in $\mathbb{R}^n$ of method (*i*) (blue color), method (*ii*) (orange color), and method (*iii*) (green color). Dots are the computation times (in seconds) of the methods w.r.t. (N, n). Lines describe the linear time complexity behaviour of the methods. The value m in the legends denotes the angular coefficient of the approximating lines.

3.4. Scalability and Real-World Applications

The experiments illustrated in the subsection above are a good representation of the general behavior that we can expect to observe in a real-world scenario, even further increasing the number n of dimensions and/or the number N of optimization procedures.

We recall that the we are assuming that the loss function f is the composition of known elementary operations, so that it is possible to use AD frameworks for the implementation of G (built on f). Under these assumptions, the experiments with the Rosenbrock function are a good representation of the general behavior of the proposed method with respect to a "classically parallelized" multi-start approach. Indeed, the main difference is only in the elementary operations necessary for implementing f and G, varying the time complexity $\mathrm{T}(f)$. Therefore, even in the worst case of no vectorization of f and/or G, the experiments show that the AD-based method is faster with respect to the parallelized multi-start (see method (*ii*) and method (*iii*)). Moreover, the implementation of the AD-based method is much simpler; indeed, the user needs only to run one optimization procedure on the function G instead of writing a code for optimally parallelizing N procedures with respect to f.

Our AD-based approach is intended to be an easy, efficient, and implicit parallelization of a general purpose multi-start optimization procedure, applicable to problems with f that are relatively simple to define/implement and typically solved by excluding an HPC context. On the other hand, for highly complex and expensive optimization problems and/or loss functions, a tailored parallelization of the multi-start procedure is probably the most efficient approach, possibly designed for that specific optimization problem and run on an HPC. Of course, the AD-based method cannot be applied to optimization problems with a loss function that is not implementable in an AD framework.

3.5. Extensions of the Method and Integration in Optimization Frameworks

The proposed AD-based multi-start method has been developed for general gradient-based optimization methods for unconstrained optimization (see Remark 2). Nonetheless, it has the potential to be extended to second-order methods (e.g., via matrix-free implementations [47] (Ch. 8)) and/or to constrained optimization. Theoretically, the method is also relatively easy to be integrated in many existing optimization frameworks or software tools, especially if they already include AD for evaluating derivatives (e.g., see [29]); indeed, the user just needs to apply these tools/frameworks with respect to the function G, instead of f, by asking to compute the gradient with reverse AD. Moreover, as we will see in the next section, the AD-based method can also take advantage of the efficient training routines of the Deep Learning frameworks for solving optimization problems.

However, the extensions mentioned above need further analysis before being implemented, and some concrete limitations for the method still exist. For example, line-search methods (e.g., see [47–50]) for the step-lengths of the N multi-start procedures have not been considered yet; indeed, the current formulation has only one shared step-length α for all the N procedures. Similarly, distinct stopping criteria for the procedures are not defined. In future work, we will extend the AD-based multi-start method to the cases listed in this subsection.

4. Multi-Start with Shallow Neural Networks

In this section, we show how to exploit the available Neural Network (NN) frameworks to build a shallow NN that, trained on fake input–output pairs, performs a gradient-based optimization method with respect to N starting points and a given loss function function f. Indeed, the usage of NN frameworks (typically, coincident with or part of AD frameworks) let us exploit the already implemented gradient-based optimization methods defined for NN training, also taking advantage of their highly optimized code implementation. Therefore, from a practical point of view, this approach is useful to easily implement a reverse AD-based multi-start method with respect to the built-in optimizers of the many available NN frameworks.

In the following definition, we introduce the archetype of such a NN. Then, we characterize its use with two propositions, to explain how to use the NN for reverse AD-based multi-start.

Definition 3 (**Multi-Start Optimization Neural Network**). *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a loss function, and let $\mathcal{N}$ be an NN with the following architecture:*

1. *One input layer of n units;*
2. *One layer of N units and fully connected with the input layer. In particular, for each $i = 1, \dots, N$, the unit u_i of the layer returns a scalar output*

$$\widehat{y}_i = f(\boldsymbol{w}_i) = f([w_{1i}, \dots, w_{ni}]^T)\,, \tag{21}$$

where $\boldsymbol{w}_i = [w_{1i}, \dots, w_{ni}]^T \in \mathbb{R}^n$ is the vector of the weights of the connections between the input layer units and u_i.

Then, we define $\mathcal{N}$ as multi-start optimization NN *(MSO-NN) of N units with respect to f.*

Remark 3. *We point the reader to the fact that (21) does not depend on the NN inputs but only on the layer weights. Then, an MSO-NN does not change its output, varying the inputs. The reasons behind this apparently inefficient property are explained in the next propositions.*

Proposition 2. *Let $\mathcal{N}$ be an MSO-NN of N units with respect to f. Let us endow $\mathcal{N}$ with a training loss function ℓ such that,*

$$\ell(\boldsymbol{x}\,,\,\boldsymbol{y}\,;\boldsymbol{w}) = \ell(\boldsymbol{w}) := \lambda \sum_{i=1}^{N} f(\boldsymbol{w}_i) = \lambda \sum_{i=1}^{N} \widehat{y}_i\,, \tag{22}$$

for any input–output pair $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{R}^n \times \mathbb{R}^N$, where $\lambda \in \mathbb{R}^+$ is fixed, and $\boldsymbol{w} = (\boldsymbol{w}_1, \dots, \boldsymbol{w}_N) \in \mathbb{R}^{nN}$ denotes the vector collecting all the weights of $\mathcal{N}$. Moreover, for the training process, let us endow $\mathcal{N}$ with a gradient-based method that exploits the backpropagation, defined as in Remark 2.

Given N vectors $\boldsymbol{x}_1^{(0)}, \dots \boldsymbol{x}_N^{(0)} \in \mathbb{R}^n$, let us initialize the weights of $\mathcal{N}$ such that the weight vector $\boldsymbol{w}_i = \boldsymbol{w}_i^{(0)}$ is equal to $\boldsymbol{x}_i^{(0)}$, for each $j = 1, \dots, n$ and $i = 1, \dots, N$. Then,

1. *For any training set, the updating of the weights of $\mathcal{N}$ at each training epoch is equivalent to the multi-start step (14), computed with reverse AD and such that $\lambda_1 = \dots = \lambda_N = \lambda$;*
2. *For each $k \in \mathbb{N}$, the weights $\boldsymbol{w}^{(k+1)} = (\boldsymbol{w}_1^{(k+1)}, \dots, \boldsymbol{w}_N^{(k+1)})$ of $\mathcal{N}$, after $k+1$ training epochs, are such that*

$$\boldsymbol{w}_i^{(k+1)} = \boldsymbol{x}_i^{(k+1)}\,, \quad \forall\, i = 1, \dots, N\,,$$

where $\boldsymbol{x}_i^{(k+1)}$ are the vectors defined by (14) and $\lambda_1 = \dots = \lambda_N = \lambda$.

Proof. Since the second item is a direct consequence of the first item, we prove only item 1.

The optimization method for the training of $\mathcal{N}$ is characterized by a function $\mathcal{M}$ that satisfies (10) and (13). Then, the training of $\mathcal{N}$ consists of the following iterative process that updates the weights:

$$\begin{cases} \boldsymbol{w}^{(0)} = (\boldsymbol{x}_1^{(0)}, \dots, \boldsymbol{x}_N^{(0)}) \in \mathbb{R}^{nN} \\ \boldsymbol{w}^{(k+1)} = \mathcal{M}\left(\boldsymbol{w}^{(k)}, \nabla_{\boldsymbol{w}}^{\mathrm{AD}} \ell(\boldsymbol{w}^{(k)})\,; \alpha\right), \quad \forall\, k \in \mathbb{N} \end{cases}, \tag{23}$$

independently of the data used for the training (see (22)). We recall that ∇^{AD} denotes the gradients computed with reverse AD and that it is used in (23) because the optimization method exploits the backpropagation for hypothesis.

Now, by construction, we observe that $\ell = L_\lambda \circ \boldsymbol{F}$, with L_λ and $\boldsymbol{F}$ defined as in Proposition 1. Then, due to Proposition 1 and Remark 2, the thesis holds.
□

Proposition 3. *Let $\mathcal{N}$ be as in the hypotheses of Proposition 2, with the exception of the loss function ℓ that is now defined as*

$$\ell(\boldsymbol{x}\,,\,\boldsymbol{y}\,;\boldsymbol{w}) = \ell(\boldsymbol{y}\,;\boldsymbol{w}) := \lambda \sum_{i=1}^{N} (f(\boldsymbol{w}_i) - y_i)^2 = \lambda \sum_{i=1}^{N} (\widehat{y}_i - y_i)^2\,, \tag{24}$$

for any input–output pair $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{R}^n \times \mathbb{R}^N$ and where $\lambda \in \mathbb{R}^+$ is fixed. Let $\mathcal{T}$ be a training set where the input–output pairs have fixed output $\boldsymbol{y} = [y^*, \dots, y^*]^T \in \mathbb{R}^N$. Then,

1. *Given the training set $\mathcal{T}$, the updating of the weights of $\mathcal{N}$ at each training epoch is equivalent to the multi-start step (14) applied to the merit function $(f(\boldsymbol{x}) - y^*)^2$, computed with reverse AD and such that $\lambda_1 = \cdots = \lambda_N = \lambda$;*
2. *For each $k \in \mathbb{N}$, the weights $\boldsymbol{w}^{(k+1)} = (\boldsymbol{w}_1^{(k+1)}, \dots, \boldsymbol{w}_N^{(k+1)})$ of $\mathcal{N}$, after $k+1$ training epochs, are such that*

$$\boldsymbol{w}_i^{(k+1)} = \boldsymbol{x}_i^{(k+1)}, \quad \forall\, i = 1, \dots, N,$$

where $\boldsymbol{x}_i^{(k+1)}$ are the vectors defined by the multi-start process of item 1.

Proof. The proof is straightforward, from the proof of Proposition 2. $\square$

With the last proposition, we introduced the minimization of a merit function. The reason is that multi-start methods can be useful not only in looking for one global optimum but also when a set of global/local optima is asked for. An example is the level curve detection problem where, for a given function $f : \mathbb{R}^n \to \mathbb{R}$ and a fixed $y^* \in \mathbb{R}$, we look for the level curve set $Y^* = \{\boldsymbol{x} \in \mathbb{R}^n \,|\, f(\boldsymbol{x}) = y^*\}$ minimizing a merit function, e.g., $(f(\boldsymbol{x}) - y^*)^2$. In this case, a multi-start approach is very important since the detection of more than one element of Y^* is typically sought (when Y^* is not empty or given by one vector only). In the next section, we consider this case problem for the numerical tests.

We conclude this section with remarks concerning the practical implementation of MSO-NNs.

Remark 4 (GitHub Repository and MSO-NNs). *The code for implementing an MSO-NN is available on GitHub (https://github.com/Fra0013To/AD_MultiStartOpt, accessed on 11 April 2024). The code of the example illustrated in Section 4.1 is reported and can be modified for adapting it to other optimization problems.*

Remark 5 (Avoiding overflow). *In some cases, if the objective function f is characterized by large values, an overflow may occur while using an MSO-NN to minimize f with an AD-based multi-start approach. In particular, the overflow may occur during the computation of $G(\boldsymbol{\xi}^{(k)}) = (L_{\boldsymbol{\lambda}} \circ \boldsymbol{F})(\boldsymbol{\xi}^{(k)}) = \sum_{i=1}^N \lambda_i f(\boldsymbol{x}_i^{(k)})$ if the parameters λ_i are not small enough. One of the possible (and easiest) approaches is to select $\lambda_1 = \cdots = \lambda_N = 1/N$; then, in the case illustrated in Proposition 3, it is equivalent to select ℓ as the Mean Square Error (MSE) loss function.*

4.1. Numerical Example

In this section, we report the results about the use of an MSO-NN to find level curve sets of the Himmelblau function

$$f(\boldsymbol{x}) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 \,. \tag{25}$$

The Himmelblau function [51] is characterized by non-negative values, with lower bound $M = 0$. In particular, f has four local minima:

$$\begin{array}{ll} \boldsymbol{x}_1^* = (3, 2)\,, & \boldsymbol{x}_2^* = (-2.805118, 3.131312)\,, \\ \boldsymbol{x}_3^* = (-3.779310, -3.283186)\,, & \boldsymbol{x}_4^* = (3.584428, -1.848126)\,; \end{array} \tag{26}$$

these local minima are also global minima since $f(\boldsymbol{x}_i^*) = 0$, for each $i = 1, \dots, 4$.

For our experiments, we consider an MSO-NN $\mathcal{N}$ defined as in Proposition 3, with $\lambda = 1/N$ (see Remark 5). The NN is implemented using the TensorFlow (TF) framework [37], and we endow $\mathcal{N}$ with the Adam optimization algorithm [52], the default algorithm for training NNs in TF, with a small and fixed step-length $\alpha = 10^{-3}$. We use Adam to show examples with a generic gradient-based method, and we set $\alpha = 10^{-3}$ to emphasize the efficiency of the AD-based multi-start even when a large number of iterations are required to obtain the solutions.

Remark 6 (**Adam satisfies** (13))**.** *In these experiments, we can use the Adam optimization algorithm because Equation* (13) *holds for Adam, with $m(\lambda) = 1$. Actually,* (13) *holds from a theoretical point of view, due to a small parameter ϵ introduced in the algorithm to avoid zero division errors during the computations. Nonetheless, in practice, Proposition 3 and Remark 2 still hold for Adam if we set ϵ such that $\epsilon/\lambda = \epsilon N$ is sufficiently small (e.g., $\epsilon N = 10^{-7}$). Specifically, the Adam optimization algorithm updates the NN weights according to the rule*

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \alpha \frac{\widehat{\boldsymbol{m}}^{(k+1)}}{\sqrt{\widehat{\boldsymbol{v}}^{(k+1)}} + \epsilon}, \tag{27}$$

where all the operations are intended to be element-wise; $\widehat{\boldsymbol{m}}^{(k+1)}$ and $\widehat{\boldsymbol{v}}^{(k+1)}$ are the first and second bias-corrected moment estimates, respectively; and $\epsilon > 0$ is a small constant value (typically $\epsilon = 10^{-7}$) to avoid divisions by zero [52]. Then, by construction (see [52]), we observe that the bias-corrected moment estimates $\widehat{\boldsymbol{m}}^{(k+1)}, \widehat{\boldsymbol{v}}^{(k+1)}$ with respect to a loss function ℓ are such that

$$\widehat{\boldsymbol{m}}^{(k+1)} = \lambda\, \widehat{\boldsymbol{m}}'^{\,(k+1)}, \quad \widehat{\boldsymbol{v}}^{(k+1)} = \lambda^2\, \widehat{\boldsymbol{v}}'^{\,(k+1)}, \tag{28}$$

where $\widehat{\boldsymbol{m}}'^{\,(k+1)}, \widehat{\boldsymbol{v}}'^{\,(k+1)}$ are the bias-corrected moment estimates with respect to another loss function ℓ' such that $\ell = \lambda \ell'$, $\lambda \in \mathbb{R}^+$. Therefore, assuming $\epsilon = 0$ and considering the effective step, the following holds:

$$\alpha \frac{\widehat{\boldsymbol{m}}^{(k+1)}}{\sqrt{\widehat{\boldsymbol{v}}^{(k+1)}}} = \alpha \frac{\widehat{\boldsymbol{m}}'^{\,(k+1)}}{\sqrt{\widehat{\boldsymbol{v}}'^{\,(k+1)}}}; \tag{29}$$

on the other hand, assuming $\epsilon > 0$, the two methods are equivalent but characterized by different parameters to avoid divisions by zero, i.e.,

$$\alpha \frac{\widehat{\boldsymbol{m}}^{(k+1)}}{\sqrt{\widehat{\boldsymbol{v}}^{(k+1)}} + \epsilon} = \alpha \frac{\widehat{\boldsymbol{m}}'^{\,(k+1)}}{\sqrt{\widehat{\boldsymbol{v}}'^{\,(k+1)}} + \epsilon/\lambda}. \tag{30}$$

Finding Level Curve Sets of the Himmelblau Function

Given the MSO-NN $\mathcal{N}$ described above, we show three cases of level-curve search: the case with $y^* = 100$ (set denoted by Y^*_{100}), the case with $y^* = 10$ (set denoted by Y^*_{10}), and the case with $y^* = 0$ (set denoted by $Y^*_0 = \{\boldsymbol{x}^*_1, \dots, \boldsymbol{x}^*_4\}$, see (26)); in particular, the latter case is equivalent to the global minimization problem of the function. For all the cases, for the multi-start method we select the $N = 10^4$ points $\boldsymbol{x}^{(0)}_1, \dots, \boldsymbol{x}^{(0)}_N$ of the regular grid as starting points

$$\{-7.5 + ih \,|\, i = 0, \dots, 99\}^2 \subset [-7.5, 7.5]^2, \tag{31}$$

where $h = 15/99$ (see Figure 5). Then, for each $y^* = 100, 10, 0$, we train $\mathcal{N}$ for K = 25,000 epochs (i.e., K multi-start optimization steps), initializing the weights with $\boldsymbol{x}^{(0)}_1, \dots, \boldsymbol{x}^{(0)}_N$. We recall that, for each $k = 0, \dots, K-1$, the $(k+1)$-th training epoch of $\mathcal{N}$ is equivalent to $N = 10^4$ Adam optimization steps with respect to the vectors $\boldsymbol{x}^{(k)}_1, \dots, \boldsymbol{x}^{(k)}_N \in \mathbb{R}^2$. The training is executed on a Notebook PC with Intel Core i5 Processor dual-core (2.3 GHz) and 16 GB DDR3 RAM (same PC of Example 1 and Section 3.3).

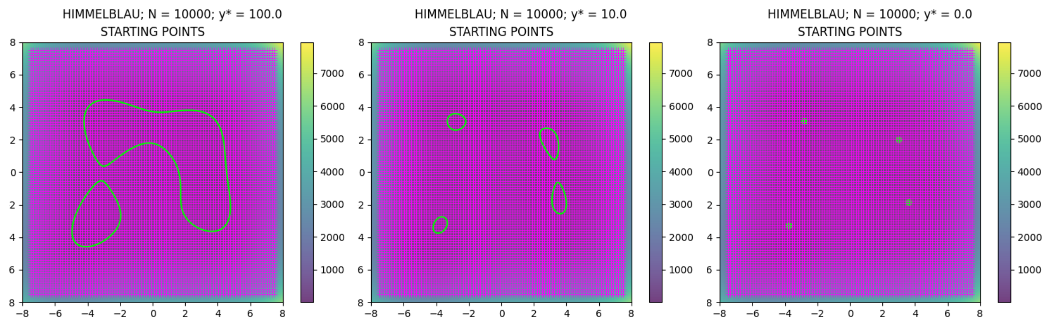

Figure 5. Top view of the Himmelblau function in $[-8,8]^2$. The magenta crosses are the $N = 10^4$ grid points of (31). In green, the level curve sets Y^*_{100}, Y^*_{10}, and Y^*_0 (from left to right).

Now, for each k = 5000, 15,000, 25,000, in Table 4 we report the computation time and the Mean Absolute Error (MAE) of the points $x_i^{(k)}$ with respect to the target value y^*, i.e.,

$$\mathrm{MAE}(k, y^*) = \frac{1}{N}\sum_{i=1}^{N} |f(x_i^{(k)}) - y^*| \,. \tag{32}$$

Table 4. MAE of the points $x_i^{(k)}$ with respect to the target value y^* and total computation time of k = 5000, 15,000, 25,000 iterations for the AD-based multi-start method using the MSO-NN $\mathcal{N}$.

	***k* = 5000**		***k* = 15,000**		***k* = *K* = 25,000**	
y^*	**MAE**	**Tot. Time (s)**	**MAE**	**Tot. Time (s)**	**MAE**	**Tot. Time (s)**
100	44.7998	8.104	5.38×10^{-4}	24.164	2.68×10^{-4}	41.116
10	70.2202	8.168	0.4612	25.347	7.3×10^{-5}	41.897
0	74.8972	8.553	1.5838	24.570	0.01812	40.861

Looking at the values in the table and at Figure 6, we observe the very good performances of the new multi-start method. Indeed, not only all the N sequences are convergent toward a solution, but we observe that the average time of one minimization step, characterized by the computation of $N = 10^4$ gradients in $\mathbb{R}^2$, is equal to 1.6×10^{-3} s. The result is interesting because the method is efficient without the need for defining any particular parallelization routine to manage the N optimization procedures at the same time.

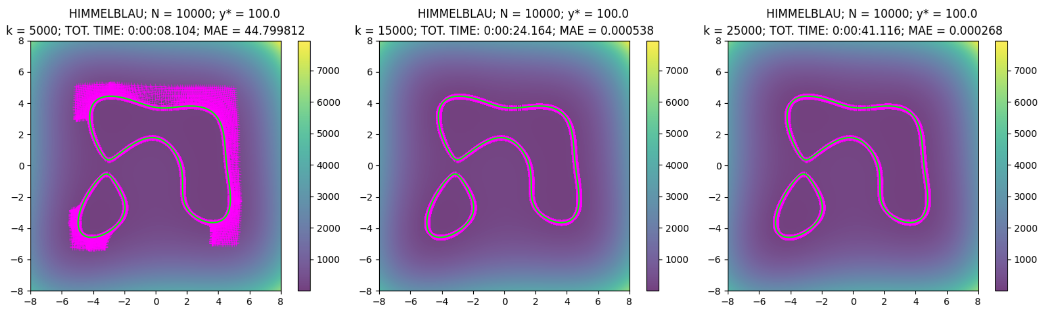

Figure 6. *Cont.*

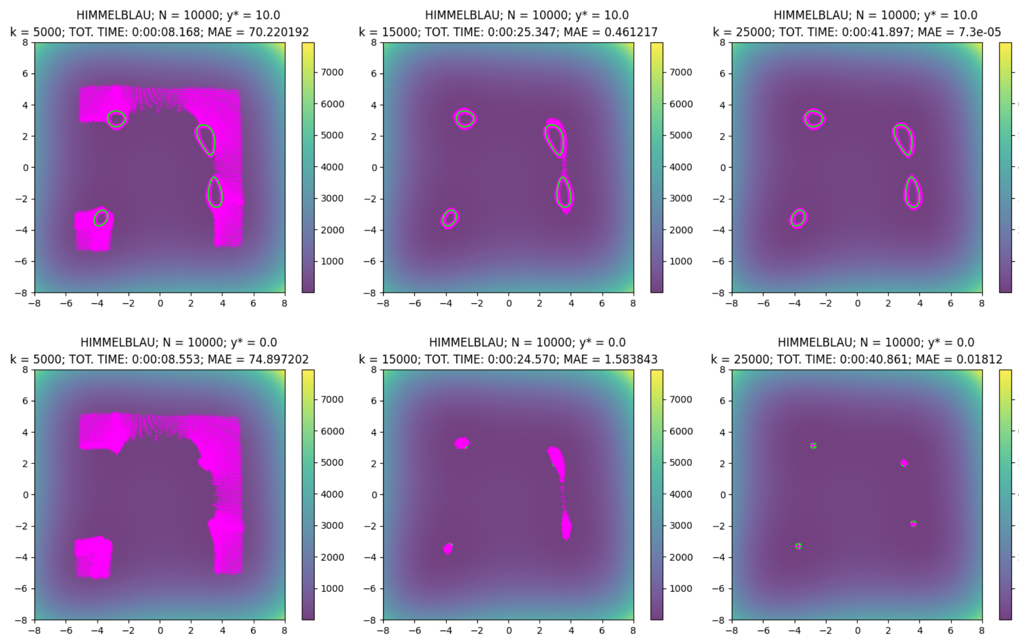

Figure 6. Top view of the Himmelblau function in $[-8,8]^2$ during the search of the level curve sets Y^*_{100}, Y^*_{10}, and Y^*_0 (first, second, and third row, respectively), denoted by the green curves or dots. The magenta crosses are the $N = 10^4$ points $\boldsymbol{x}_1^{(k)}, \dots, \boldsymbol{x}_N^{(k)}$, with respect to iteration k = 5000, 15,000, 25,000 (from left to right).

5. Conclusions

We presented a new multi-start method for gradient-based optimization algorithms, based on the reverse AD. In particular, we showed how to write N optimization processes for a function $f : \mathbb{R}^n \to \mathbb{R}$ as one optimization process for the function $G = L_\lambda \circ \boldsymbol{F}$, computing the gradient with the reverse AD. Then, assuming no HPC availability, this problem formulation defines an easy and handy solution for an implicit and highly efficient parallelization of the multi-start optimization procedure.

Specifically, the method is not supposed to be applied to complex and expensive optimization problems, where detailed and tailored parallelized multi-start methods are probably the best choice; on the contrary, the AD-based multi-start method is intended to be an easy and efficient alternative for parallelizing general purpose gradient-based multi-start methods.

The efficiency of the method has been positively tested on a standard personal computer with respect to 50 cases of increasing dimension and the number of starting points obtained from an n-dimensional test function. These experiments have been performed using a naive steepest-descent optimization procedure.

We observed that the method has the potential to be extended to second-order methods and/or to constrained optimization. In the future, we will focus on extending the method to these cases and implementing custom line search methods and/or distinct stopping criteria for the N optimization procedures.

In the end, we presented a practical implementation of the AD-based multi-start method as a tailored shallow Neural Network, and we tested it on three different values for the level curve set identification problem on the Himmelblau function, where the latter one is equivalent to the global minimization problem. This example highlights another advantage of the new method: the possibility to use NNs for exploiting the already implemented and efficient gradient-based optimization methods defined in the NN frameworks for the models' training.

Funding: This study was carried out within the FAIR-Future Artificial Intelligence Research and received funding from the European Union Next-GenerationEU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR)–MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.3—D.D. 1555 11/10/2022, PE00000013). This manuscript reflects only the authors' views and opinions; neither the European Union nor the European Commission can be considered responsible for them.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

Sample Availability: Code samples are available at this link: https://github.com/Fra0013To/AD_MultiStartOpt, accessed on 21 March 2024.

Abbreviations

The following abbreviations and nomenclatures are used in this manuscript:

AD	Automatic Differentiation
FD	Finite Differences
HPC	High-Performance Computing
MSO-NN	Multi-Start Optimization Neural Network
NN	Neural Network

References

1. Zieliński, R. A statistical estimate of the structure of multi-extremal problems. *Math. Program.* **1981**, *21*, 348–356. [CrossRef]
2. Betrò, B.; Schoen, F. Sequential stopping rules for the multistart algorithm in global optimisation. *Math. Program.* **1987**, *38*, 271–286. [CrossRef]
3. Piccioni, M.; Ramponi, A. Stopping rules for the multistart method when different local minima have different function values. *Optimization* **1990**, *21*, 697–707. [CrossRef]
4. Betrò, B.; Schoen, F. Optimal and sub-optimal stopping rules for the Multistart algorithm in global optimization. *Math. Program.* **1992**, *57*, 445–458. [CrossRef]
5. Schoen, F. Stochastic techniques for global optimization: A survey of recent advances. *J. Glob. Optim.* **1991**, *1*, 207–228. [CrossRef]
6. Yang, X.S. Chapter 6-enetic Algorithms. In *Nature-Inspired Optimization Algorithms*, 2nd ed.; Yang, X.S., Ed.; Academic Press: Cambridge, MA, USA, 2021; pp. 91–100. [CrossRef]
7. Mitchell, M. *Elements of Generic Algorithms—An Introduction to Generic Algorithms*; The MIT Press: Cambridge, MA, USA, 1998; p. 158. [CrossRef]
8. Yadav, P.K.; Prajapati, N.L. An Overview of Genetic Algorithm and Modeling. *Int. J. Sci. Res. Publ.* **2012**, *2*, 1–4.
9. Colombo, F.; Della Santa, F.; Pieraccini, S. Multi-Objective Optimisation of an Aerostatic Pad: Design of Position, Number and Diameter of the Supply Holes. *J. Mech.* **2019**, *36*, 347–360. [CrossRef]
10. Kennedy, J.; Eberhart, R. Particle swarm optimization. In Proceedings of the ICNN'95-International Conference on Neural Networks, Perth, WA, Australia, 27 November–1 December 1995; Volume 4, pp. 1942–1948. [CrossRef]
11. Yang, X.S. Chapter 8-Particle Swarm Optimization. In *Nature-Inspired Optimization Algorithms*, 2nd ed.; Yang, X.S., Ed.; Academic Press: Cambridge, MA, USA, 2021; pp. 111–121. [CrossRef]
12. Isiet, M.; Gadala, M. Sensitivity analysis of control parameters in particle swarm optimization. *J. Comput. Sci.* **2020**, *41*, 101086. [CrossRef]
13. Yang, X.S. Nature-inspired optimization algorithms: Challenges and open problems. *J. Comput. Sci.* **2020**, *46*, 101104. [CrossRef]
14. Martí, R., Multi-Start Methods. In *Handbook of Metaheuristics*; Glover, F., Kochenberger, G.A., Eds.; Springer: Boston, MA, USA, 2003; pp. 355–368. [CrossRef]
15. Hu, X.; Spruill, M.C.; Shonkwiler, R.; Shonkwiler, R. *Random Restarts in Global Optimization*; Technical Report; Georgia Institute of Technology: Atlanta, Georgia, 1994.
16. Bolton, H.; Groenwold, A.; Snyman, J. The application of a unified Bayesian stopping criterion in competing parallel algorithms for global optimization. *Comput. Math. Appl.* **2004**, *48*, 549–560. [CrossRef]
17. Peri, D.; Tinti, F. A multistart gradient-based algorithm with surrogate model for global optimization. *Commun. Appl. Ind. Math.* **2012**, *3*, e393. [CrossRef]
18. Mathesen, L.; Pedrielli, G.; Ng, S.H.; Zabinsky, Z.B. Stochastic optimization with adaptive restart: A framework for integrated local and global learning. *J. Glob. Optim.* **2021**, *79*, 87–110. [CrossRef]

19. Mathworks. MultiStart (Copyright 2009–2016 The MathWorks, Inc.). Available online: https://it.mathworks.com/help/gads/multistart.html (accessed on 11 April 2024).
20. Dixon, L.C.; Jha, M. Parallel algorithms for global optimization. *J. Optim. Theory Appl.* **1993**, *79*, 385–395. [CrossRef]
21. Migdalas, A.; Toraldo, G.; Kumar, V. Nonlinear optimization and parallel computing. *Parallel Comput.* **2003**, *29*, 375–391. [CrossRef]
22. Schnabel, R.B. A view of the limitations, opportunities, and challenges in parallel nonlinear optimization. *Parallel Comput.* **1995**, *21*, 875–905. [CrossRef]
23. Mathworks. Parfor (Copyright 2009–2016 The MathWorks, Inc.). Available online: https://it.mathworks.com/help/matlab/ref/parfor.html (accessed on 11 April 2024).
24. Python. Multiprocessing—Process-Based Parallelism. Available online: https://docs.python.org/3/library/multiprocessing.html (accessed on 11 April 2024).
25. Dixon, L.C.W. On Automatic Differentiation and Continuous Optimization. In *Algorithms for Continuous Optimization: The State of the Art*; Spedicato, E., Ed.; Springer: Dordrecht, The Netherlands, 1994; pp. 501–512. [CrossRef]
26. Enciu, P.; Gerbaud, L.; Wurtz, F. Automatic Differentiation for Optimization of Dynamical Systems. *IEEE Trans. Magn.* **2010**, *46*, 2943–2946. [CrossRef]
27. Nørgaard, S.A.; Sagebaum, M.; Gauger, N.R.; Lazarov, B.S. Applications of automatic differentiation in topology optimization. *Struct. Multidiscip. Optim.* **2017**, *56*, 1135–1146. [CrossRef]
28. Mehmood, S.; Ochs, P. Automatic Differentiation of Some First-Order Methods in Parametric Optimization. In Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics, Online, 26–28 August 2020; Volume 108, pp. 1584–1594.
29. Mathworks. Effect of Automatic Differentiation in Problem-Based Optimization. Available online: https://it.mathworks.com/help/optim/ug/automatic-differentiation-lowers-number-of-function-evaluations.html (accessed on 11 April 2024).
30. Griewank, A.; Walther, A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed.; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2008; p. 426. [CrossRef]
31. Linnainmaa, S. Taylor expansion of the accumulated rounding error. *BIT* **1976**, *16*, 146–160. [CrossRef]
32. Güneş Baydin, A.; Pearlmutter, B.A.; Andreyevich Radul, A.; Mark Siskind, J. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.* **2018**, *18*, 1–43.
33. Rumelhart, D.E.; Hinton, G.E.; Williams, R.J. Learning representations by back-propagating errors. *Nature* **1986**, *323*, 533–536. [CrossRef]
34. Verma, A. An introduction to automatic differentiation. *Curr. Sci.* **2000**, *78*, 804–807. [CrossRef]
35. Beda, L.M.; Korolev, L.N.; Sukkikh, N.V.; Frolova, T.S. *Programs for Automatic Differentiation for the Machine BESM*; Technical Report; Institute for Precise Mechanics and Computation Techniques, Academy of Science: Moscow, Russia, 1959. (In Russian)
36. Wengert, R.E. A simple automatic derivative evaluation program. *Commun. ACM* **1964**, *7*, 463–464. [CrossRef]
37. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Available online: tensorflow.org (accessed on 11 April 2024).
38. Chien, S.; Markidis, S.; Olshevsky, V.; Bulatov, Y.; Laure, E.; Vetter, J. TensorFlow Doing HPC. In Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Rio de Janeiro, Brazil, 20–24 May 2019; pp. 509–518. [CrossRef]
39. Abadi, M.; Isard, M.; Murray, D.G. A Computational Model for TensorFlow: An Introduction. In Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, Barcelona, Spain, 18 June 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 1–7. [CrossRef]
40. TensorFlow. Introduction to Graphs and tf.function. Available online: https://www.tensorflow.org/guide/intro_to_graphs (accessed on 11 April 2024).
41. Polyak, B.T. Some methods of speeding up the convergence of iteration methods. *USSR Comput. Math. Math. Phys.* **1964**, *4*, 1–17. [CrossRef]
42. Qian, N. On the momentum term in gradient descent learning algorithms. *Neural Netw.* **1999**, *12*, 145–151. [CrossRef] [PubMed]
43. Rosenbrock, H.H. An Automatic Method for Finding the Greatest or Least Value of a Function. *Comput. J.* **1960**, *3*, 175–184. [CrossRef]
44. Shang, Y.W.; Qiu, Y.H. A Note on the Extended Rosenbrock Function. *Evol. Comput.* **2006**, *14*, 119–126. [CrossRef] [PubMed]
45. Al-Roomi, A.R. *Unconstrained Single-Objective Benchmark Functions Repository*; Dalhousie University, Electrical and Computer Engineering: Halifax, NS, Canada, 2015.
46. van der Walt, S.; Colbert, S.; Varoquaux, G. The NumPy Array: A Structure for Efficient Numerical Computation. *Comput. Sci. Eng.* **2011**, *13*, 22–30. [CrossRef]
47. Nocedal, J.; Wright, S.J. *Numerical Optimization*, 2nd ed.; Number 9781447122234; Springer: Berlin/Heidelberg, Germany, 2012; pp. 31–45. [CrossRef]
48. Armijo, L. Minimization of functions having Lipschitz continuous first partial derivatives. *Pac. J. Math.* **1966**, *16*, 1–3. [CrossRef]
49. Wolfe, P. Convergence Conditions for Ascent Methods. *SIAM Rev.* **1969**, *11*, 226–235. [CrossRef]
50. Wolfe, P. Convergence Conditions for Ascent Methods. II: Some Corrections. *SIAM Rev.* **1971**, *13*, 185–188. [CrossRef]

51. Himmelblau, D. *Applied Nonlinear Programming*; McGraw-Hill: New York, NY, USA, 1972.
52. Kingma, D.P.; Ba, J.L. Adam: A method for stochastic optimization. In Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015-Conference Track Proceedings, San Diego, CA, USA, 7–9 May 2015; pp. 1–15.