Université Toulouse 1 Capitole

École Doctorale de Mathématiques, Informatique, Télécommunications de Toulouse (ED475)

Laboratoire TSE-R (CNRS UMR 5314)

On deep network training: complexity, robustness of nonsmooth backpropagation, and inertial algorithms

Entraînement des réseaux profonds : complexité, robustesse de la rétropropagation non lisse et algorithmes inertiels

Par Ryan BOUSTANY

Thèse de doctorat de mathématiques appliquées

Dirigée par Jérôme BOLTE
et Edouard PAUWELS

Présentée et soutenue publiquement le 31 mars 2025.

Devant un jury composé de :

Audrey REPETTI	Heriot-Watt University	Rapporteure
Samir ADLY	Université de Limoges	Président du jury
Peter OCHS	Saarland University	Rapporteur
Pierre ABLIN	Apple ML Research	Examinateur
Jérôme BOLTE	Toulouse School of Economics	Directeur de thèse
Edouard PAUWELS	Toulouse School of Economics	Co-directeur de thèse
Béatrice PESQUET-POPESCU	Thales LAS France	Co-encadrante
Andrei PURICA	Thales LAS France	Co-encadrant

Title: On deep network training: complexity, robustness of nonsmooth backpropagation, and inertial algorithms

Abstract : Learning based on neural networks relies on the combined use of first-order non-convex optimization techniques, subsampling approximation [1], and algorithmic differentiation, which is the automated numerical application of differential calculus [2]–[4]. These methods are fundamental to modern computing libraries such as TensorFlow [5], PyTorch [6] and JAX [7]. However, these libraries use algorithmic differentiation beyond their primary focus on basic differentiable operations [8]. Often, models incorporate non-differentiable activation functions like ReLU or generalized derivatives for complex objects (solutions to sub-optimization problems [9]). Consequently, understanding the behavior of nonsmooth algorithmic differentiation and its impact on learning has emerged as a key issue in the machine learning community [8]. To address this, a new concept of nonsmooth differentiation, called conservative gradients, has been developed to model nonsmooth algorithmic differentiation in modern learning contexts [10]. This concept also helps in formulating learning guarantees and ensuring algorithm stability in practical implementation within deep neural networks [10], [11].

In this context, we propose two extensions of the conservative calculus, finding a wide range of applications in machine learning. The first result provides a simple model to estimate the computational costs of the backward and forward modes of algorithmic differentiation for a wide class of nonsmooth programs. A second result focuses on the reliability of automatic differentiation for nonsmooth neural networks operating with floating-point numbers. Finally, we propose a new optimizer algorithm exploiting second-order information only using noisy first-order nonsmooth nonconvex automatic differentiation. Starting from a dynamical system (an ordinary differential equation), we build INNAprop, derived from a combination of INNA [12] and RMSprop [13].

Keywords: Nonsmooth algorithms, automatic differentiation, conservative gradients, nonsmooth nonconvex optimization.

Titre: Entraînement des réseaux profonds : complexité, robustesse de la rétropropagation non lisse et algorithmes inertiels

Résumé : L'apprentissage basé sur les réseaux neuronaux repose sur l'utilisation combinée de techniques d'optimisation non convexe de premier ordre, d'approximation par sous-échantillonnage [1], et de différenciation algorithmique, qui est l'application numérique automatisée du calcul différentiel [2]–[4]. Ces méthodes sont fondamentales pour les bibliothèques informatiques modernes telles que TensorFlow [5], PyTorch [6] et JAX [7]. Cependant, ces bibliothèques utilisent la différenciation algorithmique au-delà de leur cadre primaire sur les opérations différentiables [8]. Souvent, les modèles intègrent des fonctions d'activation non différentiables comme ReLU ou des dérivées généralisées pour des objets complexes (solutions à des problèmes de sous-optimisation [9]). Par conséquent, comprendre le comportement de la différenciation algorithmique non lisse et son impact sur l'apprentissage est devenu un enjeu clé dans la communauté de l'apprentissage automatique [8]. Pour aborder cela, un nouveau concept de différenciation non lisse, appelé gradients conservatifs, a été développé pour modéliser la différenciation algorithmique non lisse dans les contextes d'apprentissage modernes [10]. Ce concept facilite également la formulation de garanties d'apprentissage et la stabilité des algorithmes dans les réseaux neuronaux profonds tels qu'ils sont pratiquement implémentés [10], [11].

Dans ce contexte, nous proposons deux extensions des gradients conservatifs, trouvant une large gamme d'applications dans l'apprentissage automatique. Le premier résultat fournit un modèle simple pour estimer les coûts computationnels des modes backward et forward de la différenciation algorithmique pour une large classe de programmes non lisses. Un deuxième résultat se concentre sur la fiabilité de la différenciation automatique pour les réseaux de neurones non lisses opérant avec des nombres en virgule flottante. Enfin, nous proposons un nouvel algorithme d'optimisation exploitant uniquement des informations de second ordre en utilisant la différenciation automatique non lisse non convexe de premier ordre bruitée. Partant d'un système dynamique (une équation différentielle ordinaire), nous construisons INNAprop, dérivé d'une combinaison d'INNA [12] et de RMSprop [13].

Mots-clés : Algorithmes non lisses, différenciation automatique, gradients conservatifs, optimisation non convexe non lisse.

Remerciements

Comme on ne le fait jamais assez souvent, ces quelques lignes sont l'occasion de remercier toutes les personnes qui, de près ou de loin, ont contribué à cette thèse.

Je tiens d'abord à remercier Jérôme et Edouard, mes deux directeurs de thèse, pour tout ce que vous m'avez apporté sur le plan scientifique. Merci pour votre bienveillance et vos précieux conseils, qui m'ont permis de gagner en autonomie au fil du temps. Ce fut un vrai plaisir de travailler avec vous, et je garderai d'excellents souvenirs de nos échanges, qu'ils concernent la recherche, les sujets d'actualité ou le sport.

Je tiens également à remercier Andrei et Béatrice, mes encadrants chez Thales. Vos compétences scientifiques sont indéniables et ont grandement contribué à la qualité des travaux menés durant cette thèse. Merci, Andrei, pour ton aide dans la réalisation des expériences numériques de tous ces articles. Merci aussi pour ta patience, ta bienveillance, et tes conseils toujours avisés. J'ai été vraiment impressionné par tes connaissances en code, en recherche, et sur l'état de l'art en IA.

Je tiens à remercier les membres du jury de soutenance. Un grand merci à Audrey Repetti, Samir Adly et Peter Ochs d'avoir rapporté ma thèse, ainsi qu'à Pierre Ablin pour avoir été examinateur. Je suis très honoré et reconnaissant que vous ayez accepté d'évaluer mes travaux.

Bien que les conférences et séminaires marquent des temps forts dans la vie de la recherche et favorisent des rencontres enrichissantes, ce sont les échanges du quotidien qui ont réellement tisser le fil de mon aventure en thèse. Je remercie donc chaleureusement tous mes collègues de Thales Rungis pour leurs conseils avisés et leur bienveillance. Un merci particulier à Rémy, Jules, Sylvain, Anita, Manu, Sophie, Raymond, Daniel, et Benoît. Je remercie aussi toutes les personnes que j'ai croisées au laboratoire de TSE. Merci à mes collègues de bureau : Tam, Tony, Marine, Colombe, Camille, Lukas, Étienne, Joseph. Merci également aux enseignants-chercheurs avec qui j'ai pu échanger: Abdelaati, Laurent, Anne, Éric, Adrien, Thibault, Jérôme et Bénédicte. Un grand merci au service informatique pour votre aide précieuse avec les accès aux clusters, et une mention spéciale pour Mélodie Angeletti et Céline Parzani. Je remercie également Nicolas Renon, Christophe Marteau, Laurent Cabanas, Alejandro Estana et toute l'équipe Calmip pour leur aide précieuse concernant l'accès au cluster Turpan.

Je profite de ces quelques lignes pour remercier tous mes amis. Je ne vais pas vous citer un par un, de peur d'en oublier et de m'attirer des ennuis... Mais sachez que je vous suis infiniment reconnaissant pour votre soutien tout au long de cette aventure. Merci à mes amis d'Enghien-les-Bains, cela fait bientôt plus de la moitié de ma vie que l'on se connaît, et même si je ne le dis pas assez (voire pas du tout), merci pour votre soutien. Merci aussi à mes amis de l'ENSAE, qui ont sans doute influencé positivement mon choix de poursuivre en thèse après notre dernière année là-bas. Enfin, merci à mes amis de Dauphine, et en particulier aux

jumeaux Nathan et Sacha. Merci à Samy d'avoir relu une partie de ce manuscrit.

Je remercie Lola d'avoir toujours été là pour moi, même à distance pendant ces deux premières années, et pour son immense soutien qui m'a porté au quotidien. Enfin, je dédie ces quelques mots à mes parents et à ma soeur: merci pour votre présence constante et tout ce que vous avez fait pour moi. Rien de tout cela n'aurait été possible sans vous.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Neural networks, which were introduced decades ago [14], [15], were largely ignored by the computer science community until their resurgence in the 1980s [16]. Their popularity began in the late 2000s [17]–[19], driven by breakthroughs in applications such as computer vision [20] and natural language processing [21]. This renewed interest gave rise to deep learning, a branch of machine learning focused on neural networks. The success of the convolutional neural network AlexNet [20] in the ImageNet 2012 Challenge marked a significant milestone. More broadly, the success of deep learning can be attributed to advancements in computing power (particularly GPUs[1]), the availability of large-scale datasets [22]–[24], and the proliferation of open-source software tools [5], [6], [25].

A key challenge in machine learning problems is training more complex neural networks, which implies minimizing a large, non-convex, and nonsmooth[2] loss function. This training relies on nonsmooth automatic differentiation [3], [10] and stochastic first-order methods [1], particularly stochastic gradient descent [26]. Another major challenge in training large models is the high computational costs and energy requirements [27]–[29]. For example, the significant expense of language model pre-training means that even a small improvement in optimization algorithms can substantially reduce training time, costs, and energy consumption. For example, Google claimed that the training cost for Gemini [30] exceeded 100 million dollars. This implies a need for more cost-efficient optimization algorithms.

The practical success of deep learning algorithms is frequently valued more highly by practitioners than their theoretical guarantees. The field is driven by strong empirical performance, while the unexplained aspects of deep learning remain under active research. For example, most existing optimization algorithms, such as AdaGrad [31], AdamW [32], and Lion [33] are designed for convex optimization and do not fully exploit the nonconvex nonsmooth nature of deep learning loss functions. The convergence is well understood for convex functions [34] and smooth networks typically by using Lipschitz gradient continuity assumptions [35], [36]. However, guarantees for nonsmooth neural networks are limited, with a few exceptions [10], [37]. Actually, smooth and nonconvex is also a challenge [38]–[40].

[1]Graphics Processing Unit.

[2]Nonsmoothness refers to the lack of differentiability at certain points.

This thesis focuses on the numerical and theoretical aspects of deep learning problems arising from nonsmoothness during neural network training. Indeed, common operations in neural networks, like finding the maximum and thresholding values, introduce points of nondifferentiability that need specific analysis. In addition, the thesis proposes a new optimization algorithm that combines Hessian-driven damping with adaptive gradient scaling. It leverages second-order information and rescaling while keeping the computational and memory requirements of standard deep learning optimization algorithms.

1.1 Training nonsmooth neural networks

In the context of machine learning, training a model can be formally viewed as a risk minimization problem. Let $\mathcal{Z} \subset \mathbb{R}^d$ denote the space of possible data points, where each data point $z \in \mathcal{Z}$ is a d-dimensional vector, and let $\mathcal{P}$ be the underlying data distribution over $\mathcal{Z}$. The objective is to find a model $f(\theta, \cdot)$, parameterized by $\theta \in \mathbb{R}^p$, that minimizes the *expected risk*, defined as:

$$\min_{\theta \in \mathbb{R}^p} F(\theta) := \mathbb{E}_{z \sim \mathcal{P}} \left[\ell(f(\theta, z)) \right] \tag{1.1}$$

where $\ell : \mathbb{R}^p \times \mathbb{R}^d \to \mathbb{R}$ is a loss function that evaluates the model's performance $f(\theta, z)$ on a data point z, and $\mathbb{E}$ represents the expectation over the distribution $\mathcal{P}$. Nonsmooth optimization problems often arise when the loss function ℓ includes nonsmooth operations such as ReLU [41] or MaxPool [42]. Conversely, nonconvexity is inherent in many modern machine learning models, especially deep neural networks. The nonsmoothness and nonconvexity make the optimization landscape challenging, as traditional convex optimization techniques [43], [44] are not directly applicable.

1.1.1 Deep learning framework

Supervised learning. In the context of *supervised* learning, we consider a dataset $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^N$ consisting of input-output pairs $(x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$. The goal is to find a predictor function $f(\theta, \cdot)$ that maps inputs x to outputs y by minimizing the expected risk over the joint distribution of (x, y). This can be formally expressed as:

$$\min_{\theta \in \mathbb{R}^p} F(\theta) := \mathbb{E}_{(x,y) \sim \mathcal{P}} \left[\ell(f(\theta, x), y) \right] \tag{1.2}$$

where $\mathcal{P}$ denotes the underlying joint distribution of input-output pairs, and ℓ is a loss function such as the cross-entropy loss or mean squared error. The probability law $\mathcal{P}$ is often unknown and represents the distribution of real-world data. Hence, the *empirical risk* $\mathcal{J}$ over the training dataset $\mathcal{D}$ is minimized in practice:

$$\min_{\theta \in \mathbb{R}^p} \mathcal{J}(\theta) := \frac{1}{N} \sum_{n=1}^{N} \ell(f(\theta, x_n), y_n) \tag{1.3}$$

We denote $\mathcal{J}_n$ the n-th term of the sum in (1.3), i.e, for all $\theta \in \mathbb{R}^p$, $\mathcal{J}_n := \ell(f(\theta, x_n), y_n)$.

Neural networks. In Equation (1.3), the predictor function $f(\theta, \cdot)$ can take various forms. Here, we focus on predictors used in deep learning, known as *neural networks*. They are compositions of nonlinear functions $(\sigma_l)_{l=0}^L$ and affine transformations parameterized by $(W_l, b_l)_{l=0}^L$. Formally, a neural network with L layers can be represented as:

$$f(\theta, x) = \sigma_L\Big(W_L\sigma_{L-1}\Big(W_{L-1}\cdots\sigma_1(W_1x + b_1)\cdots + b_{L-1}\Big) + b_L\Big), \tag{1.4}$$

where σ_l are the nonlinear activation functions (for e.g ReLU, LeakyReLU or MaxPool), $W_l \in \mathbb{R}^{d_l \times d_{l-1}}$ are the weight matrices, and $b_l \in \mathbb{R}^{d_l}$ are the bias vectors for layer l. The parameter θ consists of all weight matrices and bias vectors $(W_l, b_l)_{l=0}^L$.

The Rectified Linear Unit (ReLU), defined as $\mathrm{ReLU}(x) := \max(0, x)$, is one of the most widely used activation functions in deep learning. ReLU is frequently incorporated into various model architectures, such as convolutional neural networks (CNNs) [20], deep residual networks [45], and transformer blocks in attention-based models [46], which play a critical role in natural language processing. The ReLU activation function introduces nonsmoothness due to its piecewise linear nature (refer to Figure 1.1).

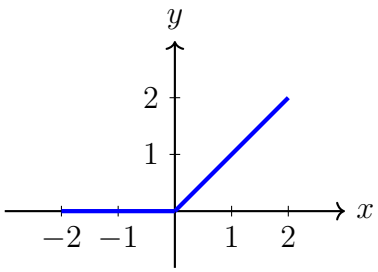


Figure 1.1: ReLU function.

Another commonly used nonlinear transformation for image data is the MaxPooling operation. Given a matrix input X, structured as a block matrix with blocks of size $d \times d$ (typically $d = 2$), MaxPool outputs a matrix where each element represents the maximum value within each block. This transformation enables image down-sampling, reducing spatial dimensions while preserving key features. When applied to uniform pixel values (as in Figure 1.2), MaxPool can cause nonsmoothness and identical points can be chosen arbitrarily (see Section 3.1). Below is an example of MaxPooling applied with a window size of 2:

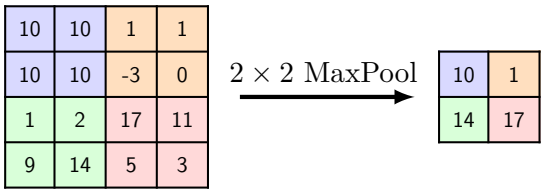


Figure 1.2: MaxPool function

Popular deep learning architectures. The range of neural network architectures is rapidly growing, with each designed to tackle specific challenges in different fields. While a two-layer network can theoretically approximate any continuous function [47], deeper networks have proven more effective in practice, especially for tasks involving large datasets [48]. These include applications like character recognition, object and speech recognition, and natural language processing.

One of the first influential architectures for image classification was LeNet-5 [49], developed for handwritten digit recognition. It had seven layers, including convolutional and subsampling layers, and laid the foundation for modern CNNs. AlexNet, introduced in 2012, built on this by using 60 million parameters across 8 layers and was trained on the ImageNet dataset, which contained 1.2 million images [22]. Residual Networks (ResNets) [45] and VGG [50] pushed network complexity, with configurations reaching up to 152 layers. More recently, Vision Transformers (ViTs) [51] have gained popularity by using self-attention mechanisms to process images, offering an alternative to CNNs. As large language models (LLMs) have grown, so has model size. For example, the Mistral 7B [52], an advanced open-source model, incorporates 7 billion parameters.

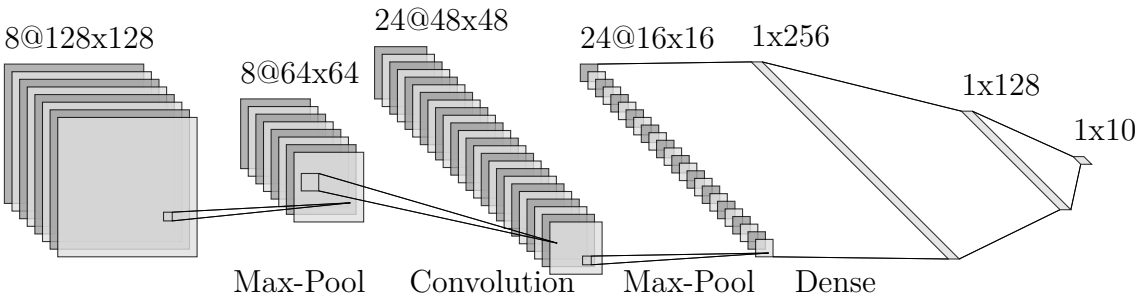


Figure 1.3: Example of a convolutional architecture (LeNet-5 [49]).

1.2 Automatic differentation

Many numerical methods rely on calculating derivatives. Classic examples include using gradient descent or Newton's method to optimize an objective function. In machine learning, gradients are essential for training neural networks [53]. However, derivatives often cannot be computed analytically and must be approximated numerically. Even when analytical solutions exist, manual calculations can be complex and error-prone. Three methods automate derivative calculations: finite differentiation, symbolic differentiation, and automatic differentiation (also called autodiff or AD). For more details on the computation of derivatives in computer programs, refer to [2], [8].

Finite differentiation. A simple way to approximate the gradient of a differentiable function $f : \mathbb{R}^p \to \mathbb{R}$ is through finite differences. This method estimates each partial derivative by slightly shifting θ along each coordinate axis:

$$\nabla f(\theta) \approx \frac{1}{t} \begin{bmatrix} f(\theta + t\mathbf{e}_1) - f(\theta) \\ f(\theta + t\mathbf{e}_2) - f(\theta) \\ \vdots \\ f(\theta + t\mathbf{e}_p) - f(\theta) \end{bmatrix}$$

where t is a small positive number, and $\mathbf{e}_i$ represents the i^{th} unit vector in the canonical basis of $\mathbb{R}^p$. The computational cost scales with the number of parameters p, making this method expensive in machine learning, especially for large models.

Symbolic differentiation. This method calculates exact derivatives by applying algebraic rules directly to the function's symbolic form. Unlike numerical methods, which approximate derivatives, symbolic differentiation provides an exact gradient for a function f by manipulating its mathematical expression $\nabla_\theta f(\theta) = \frac{\partial f(\theta)}{\partial \theta}$. For example, for $f(\theta_1, \theta_2) = \theta_1^2 + \sin(\theta_2)$, the derivatives are:

$$\frac{\partial f}{\partial \theta_1} = 2\theta_1, \quad \frac{\partial f}{\partial \theta_2} = \cos(\theta_2).$$

While this method is precise and avoids approximation errors, it can become inefficient for complex functions due to "expression swell," where derivative expressions grow large and unwieldy. This limits its practicality for large-scale models in machine learning, which involve many nested operations.

1.2.1 Practical implementation of AD

For now, we set aside the nonsmooth case. AD is a highly efficient method for computing exact gradients by systematically applying the chain rule. In deep learning, AD is better known as *backpropagation*, where it automates gradient computation by breaking down complex functions into simpler ones [3], [8], [49]. AD works by evaluating the gradient of a function $f : \mathbb{R}^p \to \mathbb{R}$ through a sequence of elementary operations (e.g., sum, product, exponential, sine, cosine). Mathematically, if f is composed of several operations, $f(\theta) = f_n \circ f_{n-1} \circ \cdots \circ f_1(\theta)$, AD uses the chain rule as follows:

$$\nabla f(\theta) = \frac{\partial f_n}{\partial f_{n-1}(\theta)} \times \frac{\partial f_{n-1}}{\partial f_{n-2}(\theta)} \cdots \times \frac{\partial f_1}{\partial \theta}$$

AD operates in two main modes: *forward mode* and *reverse mode*. Forward mode computes derivatives by propagating from inputs to outputs, which is efficient when the number of inputs is smaller than the number of outputs. In contrast, reverse mode computes derivatives by propagating from outputs back to inputs, which is ideal for functions with many inputs and fewer outputs, such as neural networks. This mode is commonly used in deep learning for backpropagation. As a result, AD is a core component of modern machine learning frameworks like TensorFlow [5], PyTorch [6], and JAX [7], enabling efficient training of large-scale models with high accuracy.

1.2.2 Complexity of AD

For rational functions, Baur and Strassen [54] show that the cost of computing gradients using AD is at most five times the cost of evaluating the function itself. This result has been extended to cover differentiable functions more generally [8]. For a differentiable function $f : \mathbb{R}^p \to \mathbb{R}^m$, if the cost to evaluate f is denoted as $\text{cost}(f)$, then the time required to compute the $m \times p$ Jacobian matrix using forward mode AD is proportional to $p \times c \times \text{cost}(f)$. In contrast, reverse mode AD computes the same Jacobian in $m \times c \times \text{cost}(f)$, where c is a constant with $c < 6$, typically around 2 to 3 [8].

This implies that reverse mode AD is more efficient when $m \ll p$, which is often the case in machine learning, where we compute the gradient of a scalar-valued loss function ($m = 1$) with respect to a large number of parameters (p).

Justifying nonsmooth calculus is essential to understanding the complexity of algorithms in practice, especially as they are implemented in nonsmooth settings. In domains like deep learning, where numerous nonsmooth models are used; a flexible and user-friendly theoretical framework is highly desirable. In Section 2.4, we present our first contribution [55], introducing a simple model to estimate the computational costs of both forward and reverse modes of AD for a wide range of nonsmooth programs.

1.2.3 Nonsmooth AD implementation

As seen previously, AD is frequently applied to nonsmooth functions in deep learning, which often arise in practical implementations due to conditional statements such as *if* and *else*. These conditionals allow AD to operate on different parts of the computational graph, even when the function is not differentiable at all points. For example, consider the representation of the ReLU function, which can be defined using conditional statements as follows:

$$\mathrm{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{else.} \end{cases}$$

In PyTorch's native implementation of ReLU, the autodiff (backpropagation) output is calculated as:

$$\text{backprop ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{else.} \end{cases}$$

AD operates on programs, not functions [11]. While AD is theoretically designed for smooth functions [8], it is commonly used for nonsmooth functions in practice. Here, we describe the spurious behavior of nonsmooth AD, as discussed by Bolte and Pauwels in [11]. The key point is that the output of AD depends on how the function is implemented, as AD operates on the program that represents the function, rather than the function itself. To illustrate this, we recall a simple experiment in PyTorch with the ReLU function, as done in [11]. The results, shown in Figure 1.4, confirm that $\mathrm{ReLU}'(0) = 0$.

$$\mathrm{ReLU}_2 \colon t \mapsto \mathrm{ReLU}(-t) + t, \qquad \mathrm{ReLU}_3 \colon t \mapsto \frac{1}{2}(\mathrm{ReLU}(t) + \mathrm{ReLU}_2(t)).$$

As mathematical functions on $\mathbb{R}$, these variations are equivalent to ReLU. However, in practice, PyTorch returns different results: $\mathrm{ReLU}_2'(0) = 1$ and $\mathrm{ReLU}_3'(0) = 1/2$, as shown in Figure 1.4.

This approach enables what can be called "formal automatic differentiation." For example, the chain rule can be applied to nonsmooth functions by substituting classical Jacobians with the outputs from automatic differentiation, effectively providing a surrogate first-order oracle. This method of handling nonsmooth functions forms the foundation of neural network training [3], [16].

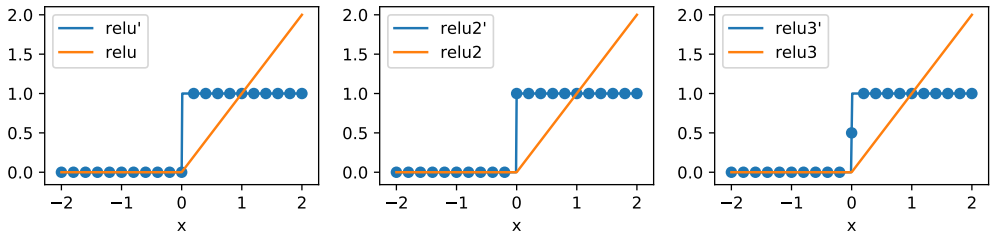


Figure 1.4: Top: AD applied to ReLU and two different implementations of the same function. Figure taken from [11].

1.3 Numerical precision in deep learning

Numerical precision is crucial for key computations in deep learning, including automatic differentiation and optimization algorithms, which rely on accurate manipulation of gradients, weights, and intermediate values. However, the finite precision of floating-point arithmetic can introduce rounding errors and numerical instability. Modern computers represent floating-point numbers using the IEEE 754 standard [56]. According to this standard, a floating-point number x is represented as:

$$x = (-1)^s \times (1 + f) \times 2^e$$

where s is the sign bit, f is the fraction (mantissa), and e is the exponent. This standard allows for a vast spectrum of values, which is crucial for the various scales present in deep learning tasks.

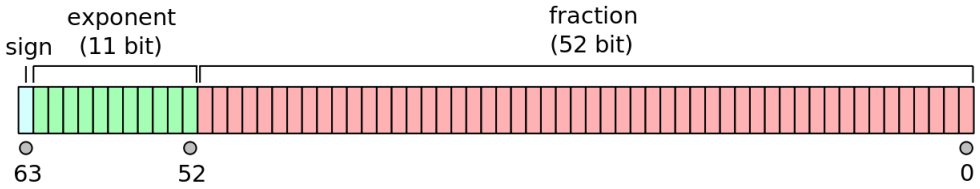


Figure 1.5: Bit representation of 64-bit double-precision floats.

Floating-point numbers have limited precision—roughly 7 decimal digits for single precision and 16 for double precision. Additionally, since floating-point addition and multiplication are not associative, the order of operations can significantly impact the computed results.

Floating numbers and deep learning. Floating-point precision formats are essential in deep learning as they balance computational speed, memory usage, and numerical accuracy. Common formats include half precision (float16), single precision (float32), and double precision (float64). These formats differ in terms of storage size, with float16 using 16 bits, float32 using 32 bits, and float64 using 64 bits, each offering increasing precision but also requiring more memory and computational resources. See Table 1.1 for more details.

Feature	**float16**	**float32**	**float64**
Total bits	16	32	64
Sign bits	1	1	1
Exponent bits	5	8	11
Fraction bits	10	23	52
Range	$\pm 6.55 \times 10^4$	$\pm 3.4 \times 10^{38}$	$\pm 1.8 \times 10^{308}$
Precision	3 digits	7 digits	15 digits
Typical use case	Speed, low-memory	General-purpose	High-accuracy

Table 1.1: Comparison of float16, float32, and float64 formats in deep learning.

The choice of precision format in deep learning significantly impacts memory usage, computational performance, and numerical stability:

• **Memory usage:** Lower precision formats, such as float16, greatly reduce memory requirements for storing model parameters and activations. This enables training larger models or using bigger batch sizes, improving both efficiency and speed, particularly for large-scale models [57]..

• **Computational performance:** Float16 accelerates arithmetic operations due to its smaller bit size, resulting in faster data processing and shorter training times. This advantage is crucial for training large neural networks, such as GPT models with billions of parameters.

• **Numerical stability and accuracy:** Despite its performance benefits, float16 can introduce numerical instability, such as exploding gradients during backpropagation [18], and may fail to capture small gradient updates accurately. In contrast, formats like float64 provide greater numerical stability but require more memory and slower computations.

Mixed-precision training. This approach is widely adopted to balance reduced memory usage and improved computational efficiency with the need for numerical precision [57]. This technique primarily employs float16 for most operations to leverage its efficiency, while reserving float32 for critical tasks, such as gradient accumulation, to ensure accuracy and prevent numerical instability. To address the risk of small gradient values underflowing in float16, loss scaling methods are applied [58]. This strategy enables models to achieve the speed and memory advantages of float16 without sacrificing the precision required for effective training.

In Chapter 3, we present our second contribution, where we study the reliability of nonsmooth autotiff for neural networks across various precision levels (16, 32, and 64 bits).

1.4 Gradient-based optimization for deep learning

After obtaining a first-order oracle of the loss function through autodiff, stochastic gradient descent can be applied to update the model parameters iteratively. It is one of the most widely used algorithms for optimization and is a common method for training neural networks. This section focuses on gradient-based optimization techniques, highlighting first-order methods and various stochastic gradient descent variants commonly employed in machine learning.

1.4.1 First-order sampling

We consider the empirical risk function $\mathcal{J}$, as defined in Equation (1.3), which is differentiable. However, this discussion can be extended to non-differentiable but locally Lipschitz continuous functions (see Section 3.2). Using the backpropagation algorithm, we can efficiently compute the gradient $\nabla\mathcal{J}$, allowing us to apply gradient descent for training deep neural networks. Despite the efficiency of backpropagation, the large dataset size N and the high number of parameters p still make gradient evaluation computationally expensive [1], [27].

Mini-batch sampling. In deep learning, the standard approach to approximating the gradient is through mini-batch sampling. This method estimates both $\mathcal{J}$ and $\nabla\mathcal{J}$ by computing only a subset of the terms in the sum from Equation (1.3). Mathematically, let $\mathsf{B} \subset \{1, \dots, N\}$ represent a subset of indices corresponding to a sample of the full dataset. For any $\theta \in \mathbb{R}^p$, we define:

$$\mathcal{J}_{\mathsf{B}}(\theta) := \frac{1}{|\mathsf{B}|} \sum_{n \in \mathsf{B}} \mathcal{J}_n(\theta) \quad \text{and} \quad \nabla\mathcal{J}_{\mathsf{B}}(\theta) := \frac{1}{|\mathsf{B}|} \sum_{n \in \mathsf{B}} \nabla\mathcal{J}_n(\theta), \tag{1.5}$$

where $|\mathsf{B}|$ is the size of the mini-batch. These mini-batches balance the trade-off between the high variance of stochastic gradient descent (SGD) and the computational cost of full-batch gradient descent [27], leading to faster convergence and better generalization. When $\mathsf{B} = \{1, \dots, N\}$, the mini-batch corresponds to the full dataset, and $\mathcal{J}_{\mathsf{B}}(\theta) = \mathcal{J}(\theta)$.

Stochastic gradient descent. Approximating $\nabla\mathcal{J}$ leads to a modified version of gradient descent (GD) called *stochastic gradient descent* (SGD). In this method, the exact gradient is replaced with an approximation, as shown in Equation (1.5). Let $(\mathsf{B}_k)_{k\in\mathbb{N}}$ be a sequence of nonempty subsets of $\{1, \dots, N\}$ and $(\gamma_k)_{k\in\mathbb{N}}$ be a sequence of step sizes. SGD is then described by the following iterative process:

$$\theta_{k+1} = \theta_k - \gamma_k \nabla\mathcal{J}_{\mathsf{B}_k}(\theta_k) \tag{1.6}$$

In the nonsmooth case, first-order sampling can be done using a backprop oracle instead of the gradient. For example, the stochastic gradient method with backprop is given by:

$$\theta_{k+1} = \theta_k - \gamma_k \text{backprop } \mathcal{J}_{\mathsf{B}_k}(\theta_k). \tag{1.7}$$

In this case, it is necessary to justify the procedure, specifically whether sampling backprop $\mathcal{J}_{\mathsf{B}_k}(\theta_k)$ (resulting from nonsmooth autodiff) provides a valid descent direction for $\mathcal{J}$ at θ_k (see Section 2.2).

Noisy interpretation of SGD. Another way to view SGD is as a *noisy* version of gradient descent. This can be written as the following equivalent formulation of (1.6):

$$\theta_{k+1} = \theta_k - \gamma_k(\nabla \mathcal{J}_{\mathsf{B}_k}(\theta_k) + \xi_k), \tag{1.8}$$

where $\xi_k = \nabla \mathcal{J}_{\mathsf{B}_k}(\theta_k) - \nabla \mathcal{J}(\theta_k)$ represents the part of the gradient not included in the SGD update compared to GD. If B_k is randomly sampled from the dataset, ξ_k is a random variable called *noise*. Moreover, as B_k is chosen such that $\mathbb{E}[\nabla \mathcal{J}_{\mathsf{B}_k}(\theta)] = \nabla \mathcal{J}(\theta)$, the conditional expectation of ξ_k with respect to the current iterate θ_k at iteration k is:

$$\mathbb{E}[\xi_k \mid \theta_k] = \mathbb{E}[\nabla \mathcal{J}_{\mathsf{B}_k}(\theta_k) - \nabla \mathcal{J}(\theta_k) \mid \theta_k] = \mathbb{E}[\nabla \mathcal{J}_{\mathsf{B}_k}(\theta_k) \mid \theta_k] - \nabla \mathcal{J}(\theta_k) = 0.$$

Thus, the sequence $(\xi_k)_{k\in\mathbb{N}}$ forms a zero-mean martingale based on the random mini-batches up to iteration k. This means that, while SGD does not follow the exact steepest descent direction at each step like GD, it does so on average [26], [27].

Computational complexity and convergence of SGD. Stochastic gradient descent (SGD), introduced by Robbins and Monro [26], is a widely used stochastic algorithm. The mini-batch version of SGD, as shown in Equation (1.6), reduces the computational cost by sub-sampling data, allowing faster iterations compared to gradient descent. Backpropagation works efficiently with mini-batches, making the computation of $\nabla \mathcal{J}_{\mathsf{B}}$ for $\mathsf{B} \subset \{1, \ldots, N\}$ about $\frac{N}{|\mathsf{B}|}$ times faster than $\nabla \mathcal{J}$. Comparing methods by epochs (where one epoch equals N backpropagations) shows that while SGD iterates faster, its updates are less precise. Despite this, in large-scale optimization, especially for deep learning, SGD is often empirically faster than GD [1].

Using gradient approximations, $\nabla \mathcal{J}$, introduces additional complexities in optimization, particularly in the nonconvex setting. Notably, critical points in stochastic gradient descent are not necessarily stationary points— where the algorithm halts if reached—in contrast to gradient descent. Specifically, there can exist an iteration $k \in \mathbb{N}$ of SGD where $\nabla \mathcal{J}(\theta_k) = 0$ but $\nabla \mathcal{J}_{\mathcal{B}_k}(\theta_k) \neq 0$, allowing the algorithm to escape the critical point θ_k. To address this, a standard strategy is to adopt a sequence of diminishing step-sizes $(\gamma_k)_{k\in\mathbb{N}}$ that satisfies conditions such as those proposed by Robbins and Monro [26]:

$$\sum_{k=1}^{+\infty} \gamma_k = +\infty, \quad \text{and} \quad \sum_{k=1}^{+\infty} \gamma_k^2 < +\infty. \tag{1.9}$$

Assuming conditions such as the noise in Equation (1.8) being uniformly bounded, this is sufficient to guarantee the convergence of SGD to critical points. However, using a sequence of step sizes that vanishes too quickly can significantly slow down SGD. Thus, selecting the appropriate sequence is crucial and can be challenging. Techniques like back-tracking line search [59] can help in choosing step sizes. Unfortunately, these methods often require computing exact values of $\mathcal{J}$, which is nearly as costly as computing exact gradients $\nabla \mathcal{J}$ in deep learning (see Section 1.2.2) and is therefore not suitable for training neural networks.

Other challenges. Mini-batch gradient descent comes with several challenges. Choosing an appropriate step size (also called learning rate) is crucial: if it is too small, convergence is slow; if it is too large, the algorithm may become unstable or even diverge. Learning rate schedules, such as annealing strategies [60], adjust the step size over time, but are fixed in advance and do not adapt to the specific characteristics of the dataset. Additionally, a uniform learning rate may not work well for sparse data, where infrequent features often require larger updates. Another issue is the nonconvex nature of neural network error functions, which can trap the model in suboptimal local minima. As noted by Dauphin et al. [61], the main difficulty arises from saddle points, where gradients are nearly zero, making it hard for SGD to escape.

1.4.2 Alternatives of the gradient method used in machine learning

In deep learning, many variants of the stochastic gradient method have been developed to speed up computationally expensive training. These variants include strategies to accelerate convergence, improve stability, and adapt to changes in the optimization process. In this section, we assume that $\mathcal{J}$ is differentiable, $(\gamma_k)_{k\in\mathbb{N}}$ is a sequence of step sizes, $(\mathsf{B}_k)_{k\in\mathbb{N}}$ are mini-batches, and θ_k represents the current iteration of the algorithm. We will present the mini-batch version of each method.

Momentum methods. Polyak's heavy ball method [62] improves gradient descent by adding a fraction of the previous update to the current one, smoothing the gradient trajectory. The update rule is:

$$m_{k+1} = \beta m_k + (1-\beta)\nabla \mathcal{J}_{\mathsf{B}_k}(\theta_k), \tag{1.10}$$

$$\theta_{k+1} = \theta_k - \gamma_k m_{k+1}, \tag{1.11}$$

where $\beta \in [0,1]$ is the momentum coefficient. In this method, think of the loss function $\mathcal{J}(\theta)$ as a landscape with valleys representing local minima. The iterates $(\theta_k)_{k\in\mathbb{N}}$ are like a ball rolling down this landscape, with the gradient $\nabla\mathcal{J}$ acting like gravity. The term m_{k+1} represents the ball's velocity, accumulating over time and speeding up the descent, while the momentum term βm_k prevents overshooting and helps the ball settle at a minimum.

There are several variations of the heavy ball method; the version here follows [63]. A popular alternative is the Nesterov accelerated gradient (NAG) method [64], with the update rule:

$$m_{k+1} = \mu_k m_k - \gamma_k \nabla \mathcal{J}_{\mathsf{B}_k}(\theta_k + \mu_k m_k), \tag{1.12}$$

$$\theta_{k+1} = \theta_k + m_{k+1}, \tag{1.13}$$

where μ_k may vary (originally $\mu_k = \frac{k}{k+3}$). Unlike the heavy ball method (Equation (1.11)), NAG evaluates the gradient at the extrapolated point $\theta_k + \mu_k m_k$, incorporating future gradient information into the current step. This anticipatory update can speed up convergence by adjusting the direction based on the expected position. While no theoretical explanation fully accounts for its success in nonsmooth, nonconvex settings, NAG is widely used in deep learning, popularized by works like [20], [63].

Adaptive methods. A common challenge in gradient methods is choosing the step sizes $(\gamma_k)_{k\in\mathbb{N}}$, especially in deep learning, where the optimal step sizes are unknown. Long training phases can lead to vanishing step sizes, slowing down progress [26], while tuning a constant step size can be costly. As a result, adaptive methods that adjust the learning rate dynamically are often preferred. Methods like AdaGrad, RMSprop, and Adam adjust the learning rate based on historical gradient information.

- **AdaGrad** [31] scales the learning rate inversely with the square root of all historical gradients:

$$\theta_{k+1} = \theta_k - \frac{\eta}{\sqrt{G_k + \epsilon}} \nabla \mathcal{J}_{\mathsf{B}_k}(\theta_k) \tag{1.14}$$

  where $\eta > 0$ is a global learning rate, $\epsilon > 0$ is a small constant for numerical stability, and $G_k \in \mathbb{R}^p$ is the vector of coordinate-wise accumulated squared gradients ($G_k = \sum_{i=1}^{k} \nabla \mathcal{J}_{\mathsf{B}_i}(\theta_i)^2$).

- **RMSprop** [13] improves AdaGrad by considering only recent gradients:

$$G_k = \beta G_{k-1} + (1-\beta) \nabla \mathcal{J}_{\mathsf{B}_k}(\theta_k)^2 \tag{1.15}$$

- **Adam** [65] combines momentum and adaptive learning rates:

$$m_k = \beta_1 m_{k-1} + (1-\beta_1) \nabla \mathcal{J}_{\mathsf{B}_k}(\theta_k) \tag{1.16}$$
$$v_k = \beta_2 v_{k-1} + (1-\beta_2) \nabla \mathcal{J}_{\mathsf{B}_k}(\theta_k)^2 \tag{1.17}$$
$$\hat{m}_k = \frac{m_k}{1-\beta_1^k}, \quad \hat{v}_k = \frac{v_k}{1-\beta_2^k} \tag{1.18}$$
$$\theta_{k+1} = \theta_k - \eta \frac{\hat{m}_k}{\sqrt{\hat{v}_k} + \epsilon} \tag{1.19}$$

Some other methods. While we have covered the most popular methods for training DNNs, many variations exist. Similar methods to Adam and AdaGrad include AdamW [32], AdaBelief [66], AdaDelta [67], and AMSgrad [68]. Other approaches include the Lookahead algorithm [69] and more recently, Lion optimizer [33].

1.4.3 Using second-order information in gradient methods

In this section, let $\theta \in \mathbb{R}^p$ represent the parameter vector, and $\mathsf{B} \subset \{1, \dots, N\}$ a mini-batch. The main idea behind second-order methods in deep learning is to develop more practical versions of Newton's method. Newton's method, assuming $\mathcal{J}$ is twice-differentiable, updates parameters as:

$$\theta_{k+1} = \theta_k - (\nabla^2 \mathcal{J}(\theta_k))^{-1} \nabla \mathcal{J}(\theta_k), \tag{1.20}$$

where $\nabla^2 \mathcal{J}(\theta_k)$ is the Hessian matrix. While this method converges quickly to a local minimum, it is computationally expensive due to the cost of computing and inverting the Hessian, especially for high-dimensional problems. Variations of Newton's method have been proposed for large-scale machine learning [70]–[72],

but few are feasible for deep learning due to the computational cost and mini-batch sub-sampling. Some methods, like AdaGrad [31], K-FAC [73], and Sophia [74], adapt Newton's method by using mini-batch estimates $\nabla \mathcal{J}_{\mathsf{B}}(\theta)$ and surrogate matrices that are cheaper to compute and invert than $\nabla^2 \mathcal{J}(\theta)$. These approaches form the basis of stochastic quasi-Newton updates.

In general, second-order information could greatly improve DNN training, but there are significant challenges. Computing or storing the Hessian, with p^2 elements, is impractical, and inverting it for Newton's method or calculating its eigenvalues is even harder. For non-smooth functions, second-order information may be less useful; for example, the ReLU function is almost everywhere twice-differentiable but has a zero second-order derivative. Additionally, mini-batch sub-sampling introduces noise, reducing the effectiveness of higher-order derivatives, even for smooth functions. As noted in Section 1.4.1 (Equation 1.8), the update direction in SGD is less precise due to this noise. For further details on second-order methods in DNN training, see [27]. In Chapter 4, we introduce a new optimization algorithm exploiting only second-order information, using noisy first-order nonsmooth nonconvex automatic differentiation. Starting from a dynamical system (an ordinary differential equation), we construct INNAprop, derived from a combination of INNA and RMSprop. It leverages second-order information and rescaling while maintaining the computational and memory efficiency of Adam.

1.5 Thesis outline and contributions

This thesis is organized into three parts, each exploring different aspects of nonsmooth optimization and automatic differentiation in deep learning.

In Chapter 2, we introduce key concepts such as nonsmooth calculus, conservative gradients, and nonsmooth automatic differentiation. The focus is on the computational complexity of nonsmooth autodiff, extending Bauer-Strassen's results to nonsmooth programs. The main contributions include a model to estimate the computational costs of the backward and forward modes for a broad class of nonsmooth programs. We also show that conservative gradients have computational properties similar to smooth derivatives, which are much more favorable than those of alternative nonsmooth oracles (subgradients or directional derivatives). This chapter is based on [55], spotlighted at ICLR 2023.

In Chapter 3, we investigate the numerical precision and reliability of nonsmooth autodiff in neural networks using the MaxPool operation. This chapter examines errors from finite precision arithmetic during training and analyzes numerical bifurcation zones, offering insights into learning stability at different precision levels. Empirical benchmarks highlight the impact of numerical precision on model accuracy and performance. This part is based on [42], published at TMLR.

In Chapter 4, we introduce INNAprop, a novel optimizer that leverages second-order information in deep learning without the high computational costs typical of such methods. This chapter details the derivation of INNAprop, which combines elements of INNA [12] and RMSprop, using noisy first-order nonsmooth auto-

matic differentiation. We validate its performance through extensive experiments on large-scale vision and language models. This last part is based on [75].

This thesis contributes to efficient methods for nonsmooth automatic differentiation, robust training of nonsmooth neural networks under limited precision, and optimization algorithms that use second-order information with minimal computational overhead in deep learning.

References. This thesis is based on the following articles:

- [55] Jérôme Bolte, Ryan Boustany, Edouard Pauwels, Béatrice Pesquet-Popescu. On the complexity of nonsmooth automatic differentiation, *Spotlight at ICLR*, 2023.
- [42] Ryan Boustany. On the numerical reliability of nonsmooth autodiff: a MaxPool case study, *Transactions on Machine Learning Research*, 2024.
- [75] Jérôme Bolte, Ryan Boustany, Edouard Pauwels, Andrei Purica. A second-order-like optimizer with adaptive gradient scaling for deep learning, *submitted*, 2024.

Chapter 2

On the complexity of nonsmooth automatic differentiation

Abstract

The research presented here is the initial work of this PhD, which is part of Thales' certification process for critical systems that integrate artificial intelligence and neural networks. Neural network training relies on nonsmooth nonconvex first-order optimization, sub-sampling approximations, and algorithmic differentiation, which automates the computation of derivatives. These techniques are central to modern libraries like TensorFlow [5] and PyTorch [6]. However, these libraries extend algorithmic differentiation beyond its core function, which is limited to differentiable operations. In this chapter, we recall the concept of *conservative gradients* [10], [11], a flexible framework for nonsmooth automatic differentiation in modern learning contexts. Using this concept, we also present our contribution to the complexity of nonsmooth automatic differentiation [55].

The main thesis of this work is that conservative gradients have computational properties similar to smooth derivatives, making them more favorable than other nonsmooth oracles like subgradients or directional derivatives.

This chapter is organized as follows:

- In Section 2.1, we cover the background and related work. In Sections 2.2 and 2.3, we introduce nonsmooth analysis and the concept of conservative gradients used throughout this thesis.
- In Section 2.4, we present a general computational model to express the cost and complexity of programs, functions, and their conservative gradients. We also introduce an abstract framework for algorithmic differentiation, culminating in Section 2.4.3, where we extend the Baur-Strassen result with the cheap conservative gradient principle.
- In Section 2.5, we describe computational lower bounds for evaluating directional derivatives and distinct subgradients for simple programs.

2.1 Introduction

Automatic evaluation of derivatives. Algorithmic differentiation (AD) was introduced around 60 years ago [76], [77] and has since been continuously developed and applied in various fields. For a detailed discussion, see [8], [78]. Today, AD is central to modern learning architectures [2], [79], [80], where training a neural network is largely the result of AD computations. Recent advancements include flexible numerical libraries [5]–[7], implicit differentiation [8], [81] and its extensions [82]–[85], the adjoint method [86]–[88] for neural ODEs [89], and differentiation of optimization algorithms [81], [90]–[92], including conjugate gradient methods [93].

Backward algorithmic differentiation, or backpropagation, is crucial for smooth optimization tasks, as it computes gradients with a cost proportional to function evaluations, regardless of dimension. This property, known as the *cheap gradient principle* [8], [94], underpins the revolution in machine learning libraries. For arithmetic circuits, this principle is formalized in the Baur-Strassen theorem [54].

Theorem 1 (Baur-Strassen [54]) *For any rational function $f : \mathbb{R}^p \to \mathbb{R}$, we have*

$$\text{cost}(\nabla f) \leq 5 \times \text{cost}(f)$$

where $\text{cost}(\cdot)$ *denotes the execution time to evaluate a function.*

Extensions exist for smooth differentiable functions [8], [54] but standard computational practice of AD consists of little known about the nonsmooth case.

Computational complexity. Key numerical operations like sorting, pooling, thresholding, and finding nearest points are common in machine learning and optimization. These nonsmooth operations are often much cheaper to compute than their smoothed counterparts. For example, the widely used ReLU activation in deep learning, which thresholds negative values to zero to simulate neuron inactivity, theoretically requires only one bit of encoding. In contrast, other nonlinear activations may need auxiliary algorithms for evaluation, leading to higher computational costs. While simple to use, this raises challenges for training models and differentiating nonsmooth functions. Standard AD practice applies differential calculus directly to nonsmooth objects, replacing gradients with surrogates like Clarke subgradients, as done in TensorFlow, PyTorch, and Jax. This approach has been highly successful [80] and widely adopted over the past decade.

Yet, despite this empirical success, Barton et al. claimed in [95] that "*there does not seem to exist [*at this day*] a true analogous reverse AD mode to compute generalized derivatives for nonsmooth functions*", illustrating the difficulty of nonsmooth AD. Conservative gradients were introduced as a faithful mathematical model capturing the formal application of calculus rules to subdifferentials by [10], [11], [84].

Related work. Conservative gradients were introduced in [10], [11] to model "formal subdifferentiation" used by practitioners and nonsmooth backpropagation. They were further studied in [84], [96], [97] and empirically investigated in [41]. Computational AD complexity was only qualitatively considered. We provide a rigorous description of this aspect based an arithmetic computational cost framework capturing programming with nondifferentiable components. The quest for a computationally cheap nonsmooth derivative has a long history in AD literature. Existing works of Griewank [8], [98]–[100] are essentially based on piecewise smoothness structures [101]. A cheap subgradient principle was also given in [102], but it requires a very strong qualification condition. As illustrated in [99], such qualification conditions can be computationally hard to check in practice.

In another research line, based on chain rules for directional derivatives, Khan-Barton [95], [103]–[105] studied the vector forward mode AD. In particular, they investigated the forward AD framework to evaluate elements of the lexicographic subdifferential (see [106]), which is contained in the Clarke subdifferential. In the worst case, the computational overhead ratio they obtain is proportional to the ambient dimension. This contrasts with our cheap gradient principle, whose constant is dimension-less. While these contributions are most relevant to nonsmooth AD, their applicability to large-scale learning models is limited, due to the central role of forward AD.

2.2 Preliminaries on nonsmooth optimization

Before the introduction of neural networks and automatic differentiation, there was a need for a nonsmooth calculus in the convex optimization. It was noted that it's not always possible to obtain the subgradient of the sum by simply summing subgradients, and there isn't a general counterpart to the chain rule for computing subderivatives in compositions [107]. Consequently, specific conditions were established to ensure the validity of basic calculus rules, such as those for sums or compositions. These conditions often required convexity, Clarke regularity, or qualification conditions for constrained problems [102], [107], [108].
For convex functions, the gradient can be extended to nonsmooth functions $F : \mathbb{R}^p \to \mathbb{R}$ by using the set-valued map ∂F. This map satisfies the condition that for all x, and for all $v \in \partial F(x)$,

$$F(z) \geq F(x) + \langle v, z - x \rangle \quad \text{for all } z \in \mathbb{R}^p. \tag{2.1}$$

Rockafellar introduced the concept of the subgradient ∂F for convex functions [44], [109], as did Moreau [110]. In the convex setting, the inequality (2.1) shows that ∂F has a natural variational interpretation, capturing the local variations of the function. This can also be understood as a first-order approximation on one side.

Clarke extended the notion of the subgradient to locally Lipschitz nonsmooth functions, motivated by minimizing maximum value functions [107]. He defined it as the convex closure of the gradient. For a locally Lipschitz function $F : \mathbb{R}^p \to \mathbb{R}$,

the Clarke subgradient at any $x \in \mathbb{R}^p$ is given by

$$\partial^c F(x) = \operatorname{conv}\left\{ \lim_{k\to+\infty} \nabla F(x_k) : x_k \in \operatorname{diff}_F, x_k \underset{k\to+\infty}{\to} x \right\} \tag{2.2}$$

where diff_F is the full measure set where F is differentiable and ∇F is the standard gradient. The subdifferential is set-valued; it takes values in subsets of $\mathbb{R}^p$, which we write $\partial^c F \colon \mathbb{R}^p \rightrightarrows \mathbb{R}^p$. For each $x \in \mathbb{R}^p$, elements of $\partial^c F(x)$ are called *Clarke subgradients* of F.

Example 1 For $\mathrm{ReLU} \colon t \mapsto \max(0, t)$, we have $\partial^c \mathrm{ReLU}(t)$ is $\{0\}$ if $t < 0$, $\{1\}$ if $t > 0$ and $[0, 1]$ if $t = 0$. We may define the function ReLU' as a selection in $\partial^c \mathrm{ReLU}$:

$$\mathrm{ReLU}'(t) = 1, \text{ if } t > 0, \qquad \mathrm{ReLU}'(t) = 0, \text{ otherwise.}$$

Unlike the convex (Fenchel–Moreau–Rockafellar) subdifferential, which is only defined for convex functions, the Clarke subdifferential is well-defined for any locally Lipschitz function, including nonconvex ones. To illustrate this, we present an example where the convex subdifferential is not applicable, but the Clarke subdifferential remains valid.

Example 2 Let $F(x) = |x| + \sin(x)$ for $x \in \mathbb{R}$. Then F is locally Lipschitz and nonconvex. The convex subdifferential (see Equation (2.1)) at $x = 0$ is empty. For $x \neq 0$, F is differentiable with

$$\nabla F(x) = \operatorname{sign}(x) + \cos(x).$$

To compute the Clarke subdifferential at $x = 0$, consider sequences $(x_k)_{k\in\mathbb{N}}$ approaching 0 from both sides:

$$x_k > 0 \Rightarrow \nabla F(x_k) \to 2, \qquad x_k < 0 \Rightarrow \nabla F(x_k) \to 0.$$

Taking the convex hull of these limits yields

$$\partial^c F(0) = [0, 2].$$

The Jacobian of a composition of two differentiable functions can be derived using the chain rule, while for the sum of two differentiable functions, the gradient of the sum is simply the sum of the gradients. However, these basic calculus principles do not extend to Clarke derivatives. For locally Lipschitz functions, only one-sided inclusions hold, as given by Clarke's generalized derivatives [107]:
• (Composition or chain rule) For $F : \mathbb{R}^q \to \mathbb{R}^r$ and $G : \mathbb{R}^p \to \mathbb{R}^q$ locally Lipschitz, $\mathrm{Jac}^{\mathrm{c}}\,(F \circ G) \subset \operatorname{conv} \mathrm{Jac}^{\mathrm{c}}\, F(G) \mathrm{Jac}^{\mathrm{c}}\, G$.
• (Sum rule) For $f : \mathbb{R}^p \to \mathbb{R}$ and $g : \mathbb{R}^p \to \mathbb{R}$, $\partial^c(f + g) \subset \partial^c f + \partial^c g$.

The inclusions are generally strict, and the usual conditions to obtain equalities are convexity or Clarke regularity [107] which are often too restrictive for machine learning applications. The equality doesn't hold even in simple cases. For example, $f = 2\mathrm{ReLU}(x)$ and $g = -\frac{1}{3}\mathrm{ReLU}(-x)$. The Clarke subdifferential of $f + g$ at $x = 0$ is $[\frac{1}{3}, 2]$ and $\partial^c f(0) + \partial^c g(0) = [0, \frac{7}{3}]$.

In the following section, we investigate a novel variational model termed *conservative gradients*, introduced in [10]. This model extends classical calculus rules—such as those for composition and sum operations—to a broad nonsmooth, nonconvex setting. A practical advantage of this framework is its independence from qualification conditions, allowing a rigorous and practical justification for the formal application of the Clarke subdifferential, as commonly employed in practice.

2.3 Nonsmooth calculus with conservative derivatives

Bolte and Pauwels [10] proposed the notion of conservative gradients, which are set-valued maps satisfying the chain rule along absolutely continuous curves.

Definition 1 (Conservative gradients [10]) *Let $f\colon \mathbb{R}^p \to \mathbb{R}$ be a locally Lipschitz continuous function and $D_f\colon \mathbb{R}^p \rightrightarrows \mathbb{R}^p$ a locally bounded, nonempty and graph closed set-valued map. Then D_f is a* conservative gradient *for f, if for any absolutely continuous curve $\gamma\colon [0,1] \to \mathbb{R}^p$,*

$$\frac{d}{dt} f(\gamma(t)) = \langle v, \dot{\gamma}(t) \rangle \qquad \forall v \in D_f(\gamma(t)), \qquad \text{for almost all } t \in [0,1]. \tag{2.3}$$

Conservative gradients may be taken convex-valued: if D_F is a conservative gradient, conv D_F is also a conservative gradient. The chain rule along curves may be extended to multivariate maps in order to define conservative Jacobians.

Definition 2 (Conservative Jacobians [10]) *Let $F\colon \mathbb{R}^p \to \mathbb{R}^m$ be a locally Lipschitz continuous function and $D_F\colon \mathbb{R}^p \rightrightarrows \mathbb{R}^{m\times p}$ a locally bounded, nonempty and graph closed set-valued map. Then D_F is a* conservative Jacobian *for F, if for any absolutely continuous curve $\gamma\colon [0,1] \to \mathbb{R}^p$,*

$$\frac{d}{dt} F(\gamma(t)) = V\dot{\gamma}(t) \qquad \forall V \in D_F(\gamma(t)), \qquad \text{for almost all } t \in [0,1]. \tag{2.4}$$

In this case, F is called *path differentiable*. A rich class of path differentiable functions is given by locally Lipschitz continuous semi-algebraic functions with the Clarke subdifferential as a conservative gradient.

Definition 3 (Path differentiable function) *A locally Lipchitz function $f : \mathbb{R}^p \to \mathbb{R}$ is called path differentiable if it satisfies the following equivalent properties:*

- *there exists a conservative gradient for f*
- *$\partial^c f$ is a conservative gradient.*

Actually, virtually all functions used in machine learning are path differentiable [10], [11]. For instance, in deep learning, ReLU and MaxPooling are path differentiable. A fundamental theorem is the following:

Theorem 2 (Path differentiable functions are ubiquitous [10]) *Locally Lipchitz semialgebraic (or definable) functions are path differentiable.*

Proposition 1 (Some path differentiable functions) *Let $f : \mathbb{R}^p \to \mathbb{R}$ be Lipschitz continuous, the following are sufficient conditions for f to be path differentiable*

(i) f is convex or concave.

(ii) f is real semialgebraic (or more generally tame, i.e., definable in some o-minimal structure).

The most salient facts about path differentiable functions and their conservative gradients are:
• (**Clarke subgradient**), for all $x \in \mathbb{R}^p$, $\partial^c f(x) \subset \text{conv}(D_f(x))$.
• (**Gradient almost everywhere**) Conservative gradients are gradients almost everywhere [10].
• (**First-order oracle**) Selection in conservative gradients can be used as surrogate gradients while preserving convergence guarantees [10], [11], [84].

Conservative Jacobians can be composed while preserving conservativity [10], a feature which do not enjoy Clarke Jacobians.

Proposition 2 (Composition rule [10]) *For $F\colon \mathbb{R}^q \to \mathbb{R}^r$, $G\colon \mathbb{R}^p \to \mathbb{R}^q$ path differentiable with conservative Jacobians D_F and D_J respectively, $D_F(G(x)) \times D_G(x)$ is a conservative Jacobian for $F \circ G$.*

Corollary 1 (Sum rule) *For $f : \mathbb{R}^p \to \mathbb{R}$ and $g : \mathbb{R}^p \to \mathbb{R}$ path differentiable with conservative gradients D_f and D_g respectively, then $D_f + D_g$ is a conservative gradient for $f + g$.*

Conservative gradients offer a convenient framework for modeling automatic differentiation (AD) in the nonsmooth setting [10], [11]. The composition rule supports AD by applying the chain rule to Clarke subderivatives. For example, AD on a neural network composed of ReLU and affine functions may not yield an element of the Clarke Jacobian but rather a selection from a conservative Jacobian. While a framework to justify AD within the Clarke subdifferential model was proposed in [102], it relies on qualification conditions and rules that may not align with common practice.

2.4 A cheap conservative gradient principle

2.4.1 Calculus model, programs, computational cost and complexity

A *dictionary* $\mathcal{D}$ is a finite set of real functions (*e.g.* $\{+, -, \times, /\}$); it is paired with $\mathcal{P}^0(\mathcal{D})$, a set of elementary programs implementing them in real arithmetic. Starting from $\mathcal{P}^0(\mathcal{D})$, we aim at capturing the notion of "program of programs" at any depth. As this is an inductive process, we call $k \in \mathbb{N}$ a program "level", which is simply an induction counter needed for consistency. Recursively, programs of level $k+1$, in $\mathcal{P}^{k+1}(\mathcal{D})$, consist of combinations of outputs of programs of level k,

in $\mathcal{P}^k(\mathcal{D})$. For example if P_1 and P_2 are elementary programs in $\mathcal{P}^0(\mathcal{D})$, then the program which sums the outputs of P_1 and P_0 is of level 1. More precisely :

Let p, q be input and output sizes respectively and $m \geq p + q$ a memory size. A *predecessor relation* is a set valued map $\mathtt{pr}\colon \{1, \dots, m\} \rightrightarrows \{1, \dots, m\}$ such that for $i = 1, \dots, m$:
- for $j \in \mathtt{pr}(i)$, $j < i$.
- $\mathtt{pr}(i)$ is empty if $i \leq p$ and nonempty otherwise.

An *adapted program sequence* $(g_i)_{i=p+1}^m$ in $\mathcal{P}^k(\mathcal{D})$, is a set of programs such that g_i has $|\mathtt{pr}(i)|$ input arguments and a single output, for all $i = p+1, \dots, m$. Given $\left(p, q, m, \mathtt{pr}, (g_i)_{i=p+1}^m\right)$, the program given in Algorithm 1 is a level $k+1$ *program on* $\mathcal{D}$.

Algorithm 1:

Program data: $\left(p, q, m, \mathtt{pr}, (g_i)_{i=p+1}^m\right)$.

Input: $x = (x_1, \dots x_p)$
1: **for** $i = p+1, p+2, \dots m$ **do**
2: $\quad x_i = g_i(x_{\mathtt{pr}(i)})$ where
3: $\quad x_{\mathtt{pr}(i)} = (x_j)_{j \in \mathtt{pr}(i)}$.
Return: $y := (x_j)_{j=m-q+1}^m$.

The set of *programs with dictionary* $\mathcal{D}$ is $\mathcal{P}(\mathcal{D}) = \bigcup_{k \geq 0} \mathcal{P}^k(\mathcal{D})$. We shall see however that $\mathcal{P}^k(\mathcal{D}) = \mathcal{P}^1(\mathcal{D})$ for all k, using modification of the computational graph.

A *cost on a dictionary* $\mathcal{D}$ is a nonnegative function on $\mathcal{D}$, it extends additively by induction on programs on $\mathcal{D}$ through the rule $\mathrm{cost}(P) = \sum_{i=p+1}^m \mathrm{cost}(g_i)$ where P is a program on $\mathcal{D}$ as described in Algorithm 1. A direct example is the dictionary of arithmetic functions $\{+, -, \times, /\}$, together with addition or multiplication by fixed constants, denoted by $+c$ and $\times c$ respectively[1], see also Appendix 2.6.1. Throughout the section, we assume that dictionaries contain at least operations $+$ and $\times$. Each program on $\mathcal{D}$ may be represented by a program in $\mathcal{P}^1(\mathcal{D})$ with the same cost by expanding all subprograms until they reduce to an elementary program. Cost evaluation is thus well-defined in such programs. As detailed in Appendix 2.6.1, this model of computation is equivalently expressed using directed acyclic graphs.

To sum up, we have defined the set of programs $\mathcal{P}(\mathcal{D})$ on $\mathcal{D}$, which includes programs of programs. The programs g_i in Algorithm 1 may be taken in $\mathcal{P}(\mathcal{D})$. The cost of a program is evaluated through the calls it makes to elementary programs in the dictionary.

Programs vs functions: A program P defines a unique input-output function f: we say that P "computes" f, or "implements" f, and with a slight abuse of notation, we will identify P and f when there is no ambiguity (*e.g.* derivative of P). We use the equivalence relation $\sim$ to relate programs computing the same

[1]Constants need to be distinguished from variables (for instance to define a polynomial)

function. The equivalence classes correspond to functions expressible by programs with a given dictionary $\mathcal{D}$. Given a function $f\colon \mathbb{R}^p \to \mathbb{R}^q$ and a program P on dictionary $\mathcal{D}$, with p inputs and q outputs, we write $f = [P]$ to denote the fact that P is in the equivalence class of programs computing f, that is, P implements f.

Complexity of a function: The complexity of a function f over a dictionary $\mathcal{D}$ is the quantity $\mathrm{comp}(f, \mathcal{D}) = \inf\{\mathrm{cost}(P),\ s.t \quad P \in \mathcal{P}(\mathcal{D}),\ f = [P]\}$, the infimum being over all programs implementing f on dictionary $\mathcal{D}$. It could be infinite, if it is finite then it is attained.

2.4.2 Automatic differentiation

We pertain to programs implementing functions, that is Algorithm 1 with single outputs $q = 1$.

Given a dictionary $\mathcal{D}$ of locally Lipschitz path differentiable functions, a *derived dictionary* is a set of functions $\mathcal{D}' \supset \mathcal{D}$ which extends $\mathcal{D}$ and contains operations required to express at least an element in a conservative gradient for each of the functions in $\mathcal{D}$, for example, an element in the Clarke subdifferential. We also consider a cost function on $\mathcal{D}'$, which we denote by cost and which extends to programs over $\mathcal{D}'$. Given programs g_i on $\mathcal{D}$, $i = p+1, \dots, m$, we define d_i a *derived program* on $\mathcal{D}'$, with $|\mathtt{pr}(i)|$ inputs and outputs, which returns an element of a conservative gradient for g_i (as for instance a Clarke subgradient, or simply a gradient in the C^1 case). By gd_i, we denote a program on $\mathcal{D}'$ evaluating $(g_i(x), d_i(x))$ jointly for a given x. We denote by Algorithm 1', an extension of Algorithm 1 which additionally returns $w_i = d_i(x_{\mathtt{pr}(i)})$ for $i = p+1, \dots, m$, by replacing line 2 in Algorithm 1 with a call to gd_i instead of g_i. The backward (resp. forward) AD program backprop(P) (resp. forprop(P)) is defined as follows:

Algorithm 2: Algorithmic differentiation of P as in Section 2.4.1

Input: variables $(x_i)_{i=1}^p$

Forward evaluation with derivatives: evaluate $w_i = d_i(x_{\mathtt{pr}(i)})$, $i = p+1, \dots, m$,

with Algorithm 1': Algorithm 1 with gd_i instead of g_i on line 2.

1: **Forward mode:**
2: Initialize: $\frac{\partial x_i}{\partial x} = e_i$, $i = 1, \dots, p$, from canonical basis in $\mathbb{R}^p$.
3: **for** $i = p+1, \dots m$ **do**
4:

$$\frac{\partial x_i}{\partial x} = \sum_{j \in \mathtt{pr}(i)} \frac{\partial x_j}{\partial x} w_i[j]$$

where $x = (x_1, \dots, x_p)$.

Return: $\frac{\partial x_m}{\partial x}$ and x_m.

1: **Backward mode:**
2: Initialize: $v = e_m$
3: **for** $t = m, \dots p+1$ **do**
4: **for** $j \in \mathtt{pr}(t)$ **do**
5: Update coordinate j of v:

$$v[j] := v[j] + v[t] w_t[j]$$

Return: $(v[j])_{j=1}^p$ and x_m.

Note that Algorithm 2 starts with Algorithm 1', i.e., Algorithm 1 with gd_i instead of g_i on line 2. Its computational cost, denoted $\mathrm{cost}(gd_i)$, should be thought

of as an exogenous parameter: *it may model, for instance, the use of underlying software libraries or the hardware properties.*

2.4.3 Computational complexity of nonsmooth AD

We now evaluate the complexity of the forprop and backprop operations for conservative gradients in the path differentiable case – which encompasses, as mentioned earlier, all semi-algebraic and definable locally Lipschitz functions. We show, in particular, that backpropagation with conservative gradients has a computational overhead ratio that is independent of the dimension. This is in contrast with the best known algorithmic oracles for the Clarke subdifferential (see [95], [103]–[105]), whose computational overhead ratio scales linearly with the dimension. The following theorem is proved in Section 2.7.

Theorem 3 (Complexity of nonsmooth AD) *Let P be a program over a dictionary $\mathcal{D}$ of path-differentiable functions with p inputs as in Algorithm 1 & 2. Then, the corresponding function $[P]$ is path differentiable, there is a conservative gradient D_P for the function $[P]$ such that:*
(i) **(Cost of backward mode)** *At each input point $x \in \mathbb{R}^p$, the output of program* $\mathrm{backprop}(P)$ *is in $D_P(x)$ and we have* $\mathrm{cost}(\mathrm{backprop}(P)) \leq \omega_b \ \mathrm{cost}(P)$, *where*

$$\omega_b = \max_{i=p+1,m} \{(\mathrm{cost}(gd_i) + 2\max(\mathrm{cost}(+), \mathrm{cost}(\times))|\mathtt{pr(i)}|) \ / \ \mathrm{cost}\,(g_i)\}\,. \tag{2.5}$$

(ii) **(Cost of forward mode)** *At each input point $x \in \mathbb{R}^p$, the output of program* $\mathrm{forprop}(P)$ *is in $D_P(x)$ and we have* $\mathrm{cost}(\mathrm{forprop}(P)) \leq \omega_f \times \mathrm{cost}(P)$ *where*

$$\omega_f = \max_{i=p+1,m} \{(\mathrm{cost}\,(gd_i) + p|\mathtt{pr}(i)|\mathrm{cost}(\times) + p(|\mathtt{pr}(i)| - 1)\mathrm{cost}(+)) \ / \ \mathrm{cost}\,(g_i)\}\,.$$

There is a dissymmetry between the two modes since the constant ω_b is independent of the dimension p. This is why property (i) is sometimes called the "cheap conservative gradient principle" extending the classical smooth one which was derived by [54] for real rational functions. Theorem 3 describes worst case upper bounds (maximum over i), which are tight, for example if $pr(i)$, costs of g_i and gd_i are independent of i.

We will consider several examples now.

The class of ReLU programs: Let $\mathcal{D}_{\mathrm{ReLU}}$ be the dictionary composed of elementary arithmetic operations, logarithm, exponential and the ReLU function:

$$\mathcal{D}_{\mathrm{ReLU}} := \{+, \times, +c, \times c, \mathrm{inv}, \exp, \log, \mathrm{ReLU}\}. \tag{2.6}$$

A ReLU program P is a program with dictionary $\mathcal{D}_{\mathrm{ReLU}}$; it can be expressed in a compositional form (Section 2.4.1) with program sequences in $\mathcal{D}_{\mathrm{ReLU}}$. Note that this yields path differentiable functions.

Assumption 1 (Computational Cost) In Algorithms 2, define the dictionary $\mathcal{D}'_{\mathrm{ReLU}} := \mathcal{D}_{\mathrm{ReLU}} \cup \{\mathrm{ReLU}'\}$ as in Example 1; then, all operations from $\mathcal{D}'_{\mathrm{ReLU}}$ have unit cost (see Remark 1).

Corollary 2 (Backprop complexity of ReLU **programs)** *Let P be a* ReLU *program, under Assumption 1, we have:* $\text{cost}(\text{backprop}(P)) \leq 5 \times \text{cost}(P)$. *This extends to more complex cost weighting schemes (Remark 1) and to selection functions which virtually capture all losses in ML (Remark 2).*

Table 2.1: Complexity constant of ω_b in Theorem 3 for elementary g in $\mathcal{D}_{\text{ReLU}}$ and derived program with dictionary $\mathcal{D}'_{\text{ReLU}}$. This proves Corollary 2 (more details in Appendix 2.7.1).

g	$(+, \times)$	$(+c, \times c)$	log	exp	inv	ReLU
$(\text{cost}(gd) + 2\text{cost}(\times)\lvert\texttt{pr}\rvert)\ /\ \text{cost}\,(g)$	5	3	4	3	5	3

A numerical example on ReLU **networks:** We illustrate the backpropagation complexity of ReLU networks through a numerical experiment on the standard MNIST dataset of handwritten digits. We consider a multilayer perceptron (MLP) with ReLU activations and analyze how the computational overhead associated with nonsmooth automatic differentiation manifests in practice. The architecture is varied systematically: we consider networks with increasing depth (from 1 to 20 hidden layers) and varying hidden layer widths. Each hidden unit uses a ReLU nonlinearity. The networks are trained using stochastic gradient descent with fixed hyperparameters across all settings to ensure comparability. For each architecture, we compute the theoretical complexity overhead ω_b defined in Theorem 3. As shown in Figure 2.1, our results confirm the theoretical predictions of Theorem 3. In particular, we observe that the overhead ω_b remains independent of the network dimension (i.e., the number of layers or neurons).

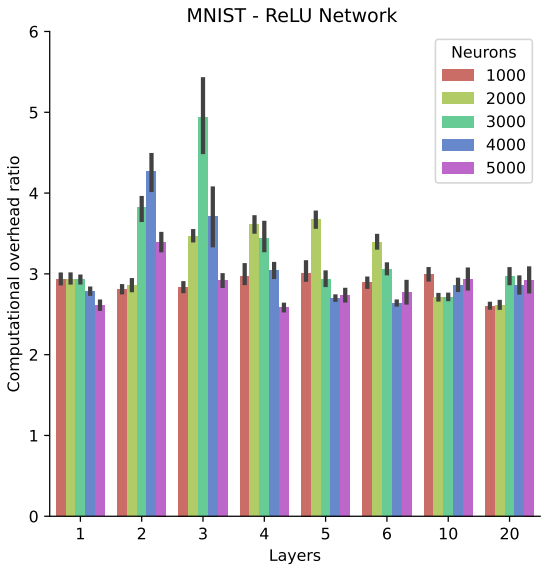


Figure 2.1: Analysis of computational costs ω_b for backpropagation in ReLU networks using the MNIST dataset.

Remark 1 (On refined cost systems) Unit cost in Assumption 1 gives a simple interpretation to Corollary 2: the cost of a program is the total number of numerical operations. This rough estimate of computational complexity, could be refined with different weighting schemes. However, the obtained constant 5 is robust to many different weighting choices, far beyond Assumption 1. We detail an example in the Appendix 2.7.2 for which the cost of all smooth nonlinear operations different from + or × is $c_{\text{nonlin}} \geq 1$ and we model the cost of sign branching in computation of ReLU and ReLU$'$ with constant $c_{\text{ReLU}} \geq 0$. This yields the same constant as in Corollary 2.

Remark 2 (Beyond ReLU programs) Many other dictionaries could be considered. ReLU is an example chosen for its simplicity, but Corollary 2 would hold similarly (with the same constant 5) for many different nonsmooth activations or components such as absolute value, max-pooling, ELU function, ℓ_1 and ℓ_∞ norms. Similar results could be developed for the class of selection functions, which encompasses the vast majority of ML building blocks (see [11]). This is sketched in Appendix 2.7.3.

Chaining backpropagation derived programs: Our approach is flexible to describe "programs of programs" and backpropagation chaining. Let P be a program as in Algorithm 1, with adapted ReLU program sequence $\{(g_i)_{i=p+1}^m\}$. If $\text{cost}(g_i) \gg |\text{pr}(i)|$, g_i is a "long program", with many operations per input. We may set $gd_i = \text{backprop}(g_i)$ using Algorithm 2, $i = p+1, \ldots, m$. From Corollary 2, we have $\text{cost}(gd_i)/\text{cost}(g_i) \leq 5$, and for long programs $\omega_b \simeq 5$ in Theorem 3. This illustrates the versatility of our approach as it captures the complexity of chaining backprop operations, the resulting estimate being quite sharp in the regime of long programs. Refer to Figure 2.1 for a numerical illustration.

2.5 On the computational hardness of generalized gradients

Let P and DP be two programs such that DP evaluates jointly P and a derivative of P. In the sequel, we use the term (*computational*) *overhead ratio* of DP to denote the quantity $\frac{\text{cost}(DP)}{\text{cost}(P)}$ and computational overhead ratio of derivatives of P to denote the quantity $\frac{\text{comp}(DP)}{\text{cost}(P)}$. As established in Theorem 3, this ratio is dimensionless in the case of backpropagation with conservative gradients. Are there other ways to compute cheap nonsmooth gradients? Toward an answer to this question, we discuss this ratio for other nonsmooth differentiation oracles: directional derivatives (for which we relate worst-case complexity to that of matrix multiplication), lexicographic derivatives with forward AD (with an overhead ratio of order p [95]). As for the Clarke subdifferential, we prove the hardness of subgradients enumeration. Our motivation to estimate the complexity of these particular types of derivatives (directional, lexicographic and Clarke) is that they serve as a basis to alternative implementable AD approaches (see [95] and references therein), and are thus concurrent strategies of conservative gradient backpropagation. The results presented below do not provide a definitive answer, but they strongly sug-

gest that backpropagation of conservative gradients has a much more favorable complexity.

2.5.1 The overhead ratio for evaluating p directional derivatives

Given $G\colon \mathbb{R}^p \to \mathbb{R}$ locally Lipschitz and $x, d \in \mathbb{R}^p$, the directional derivative of G at x in direction d is given by $\lim_{t\downarrow 0}(G(x+td)-G(x))/t$ when the limit exists. This section considers a family of functions with p inputs and q real parameters, represented by a locally Lipschitz function $F\colon \mathbb{R}^p \times \mathbb{R}^q \to \mathbb{R}$, for which we investigate hardness of evaluation of p directional derivatives. The function F may describe, for instance, a ReLU feedforward neural network empirical loss, parameterized by q real weights, with p inputs. For functions represented by ReLU programs, we prove an overhead ratio of order $p^{\omega-2+o(1)}$ where ω is the matrix multiplication exponent (see definition below). In all rigor, it is not known whether $\omega > 2$ or $\omega = 2$, so the derived ratio could be essentially dimensionless (if $\omega = 2$), though all practical evidences are against this so far. The best known lower bound is $\omega < 2.37$, and in practice, the matrix multiplication exponent is closer to 2.7, both corresponding to a dimension-dependent overhead, in contrast with the smooth case with essentially dimensionless overhead ratio to evaluate p directional derivatives (essentially a gradient).

Complexity of matrix multiplication: Throughout this section, we set $\mathcal{D} = \{+, \times, +c, \times c\}$, with unit costs (corresponding to polynomial functions). Denote by $c(p)$ complexity of $p \times p$ matrix multiplication. More precisely, if $f\colon \mathbb{R}^{p\times p} \times \mathbb{R}^{p\times p} \to \mathbb{R}^{p\times p}$ is such that $f(A, B) = AB$ for all, square matrices $A, B \in \mathbb{R}^{p\times p}$, we have $c(p) = \mathrm{comp}(f, \mathcal{D})$, which we may write $c(p) = p^{\omega+o(1)}$ where ω is called the *matrix multiplication exponent*. Note that $c(p) \geq p^2$, as one needs at least one operation for each of the $2p^2$ entries.

Directional derivatives: Given a function $F\colon \mathbb{R}^p \times \mathbb{R}^q \to \mathbb{R}$, we denote by $F_1'\colon \mathbb{R}^p \times \mathbb{R}^q \times \mathbb{R}^{p\times p} \to \mathbb{R}^p$ the function which associates to $x \in \mathbb{R}^p$, $y \in \mathbb{R}^q$ and a matrix $A \in \mathbb{R}^{p\times p}$ the p directional derivatives with respect to x variable, for fixed y, in directions given by the columns of A. The proof of the following theorem is given in Appendix 2.8.

Theorem 4 (Computational ratio for directional derivatives) *There exists a function $F\colon \mathbb{R}^p \times \mathbb{R}^q \to \mathbb{R}$ and a program P_F implementing F on dictionary $\{+, \times, \mathrm{ReLU}, +c, \times c\}$ (all operations have unit cost), such that for any program P' implementing $(y, A) \mapsto F_1'(0, y, A)$ on derived dictionary $\{+, \times, \mathrm{ReLU}, \mathrm{ReLU}', +c, \times c\}$,*

$$\mathrm{cost}(P')/\mathrm{cost}(P_F) \geq (c(p) - 5p)/(40p^2) = p^{\omega-2+o(1)}. \tag{2.7}$$

Theorem 4 has q parameters, parametric dependency is required to express hardness. Indeed, for some parameter values, computation may be trivial (e.g. null values). Alternatively, it states that for some values of the q parameters, computing p directional derivatives has cost as in Equation (2.7).

The bound in (2.7) is sharp up to multiplicative constants for linear ReLU networks, see Remark 5 in Appendix 2.6.2.

Consequences: Our overhead estimate is roughly $p^{\omega-2}$, it constitutes a bottleneck: a "cheap nonsmooth p directional derivatives principle", would imply easy matrix multiplication, to the point that $\omega = 2$. Since the seminal work of [111], it is known that $\omega \leq \log_2(7) \simeq 2.81$. Determining the precise exponent ω has been an object of intense research [112]. Asymptotically, one has $2 \leq \omega < 2.373$, see [113], [114], the best known bound being given in [115]. In this case, the estimate in (2.7) is roughly $p^{0.373}$.

Comparison with the smooth case: If F is C^1, evaluating p directional derivatives is comparatively easier because $F'(x,d) = \langle \nabla F(x), d \rangle$ for all $x, d \in \mathbb{R}^p$. Hence, one may first evaluate ∇F (once), at a cost similar to that of F (cheap gradient principle), and then evaluate p scalar products, at a cost p^2. If the cost of F is of order p^2 at least (for example F is a feedforward neural network with p inputs and a layer of p hidden neurons), then this is overall proportional to the cost of computing F.

2.5.2 Computing Clarke subgradients using forward automatic differentiation

In [103]–[105], several automatic differentiation strategies are proposed to evaluate elements of the Clarke subdifferential. These approaches are based on directional [116] and lexicographic derivatives [106] which satisfy a chain rule under structural assumptions. The chain rule may be implemented using the vector forward mode of automatic differentiation [95], which suffers from computational overhead scaling linearly in p, contrary to the reverse mode in Theorem 3. Reducing this factor is an open question, even for compositional functions involving only univariate nonsmoothness such as absolute value [117]. More details are given in Appendix 2.6.2.

2.5.3 Computational hardness of subgradient enumeration

We investigate in this section the hardness finding subgradients for programs defined on the elementary dictionary $\mathcal{D}_0 = \{+, -, \mathrm{ReLU}\}$ with unit costs. Let us denote by $\mathcal{P}(\mathcal{D}_0)$ the set of such programs. We will, with a slight abuse of notation, identify a program $P \in \mathcal{D}_0 = \{+, -, \mathrm{ReLU}\}$ with the function it computes to state our complexity result (proof in Section 2.9).

Theorem 5 (Clarke subgradients and NP-Hardness)
(i) The problem of finding two distinct subgradients in the Clarke subdifferential of $P \in \mathcal{P}(\mathcal{D}_0)$ at given input (or one single subgradient if it is reduced to a singleton) is NP-hard.
(ii) Deciding if $P \in \mathcal{P}(\mathcal{D}_0)$ is not differentiable at some given input is NP-hard.

Remark 3 In Theorem 5, numerical parameters and inputs are constrained to be in $\{-1, 0, 1\}$, so that the hardness result does not depend on numerical representation and only involves program size (strong NP-hardness). See Appendix 2.9 for more details.

The above problems (i)-(ii) enter the field of computational complexity as we consider programs $P \in \mathcal{P}(\mathcal{D}_0)$ with a natural notion of size, given by their cost, $\text{cost}(P)$, the number of operations (recall that we assumed unit costs). Since the considered programs implement piecewise linear functions, it follows from [95, Proposition 2.7] that, our hardness result also holds for the lexicographic subdifferential [106], which reduces in this case to the set of neighboring gradients (see Appendix 2.9).

The counterpart of the above problem for AD conservative gradients as in Definition 4 is tractable, illustrating a major computational difference between Clarke subdifferential and AD conservative gradient. The proof is in Appendix 2.9.4, by reduction to a graph shortest path problem.

Proposition 3 (Find two elements in AD conservative gradients is tractable) *Given $P \in \mathcal{P}(\mathcal{D}_0)$, with conservative gradient D_P given by Theorem 3, finding two elements in $D_P(x)$ at a given input x (or one single element if $D_P(x)$ is a singleton) is solvable in polynomial time.*

To conclude, we extended the "cheap gradient" principle to nonsmooth automatic differentiation with a flexible version of Baur-Strassen's result: the overhead ratio of conservative gradients is independent of the dimension. On the other hand, we showed that the potential gain in efficiency of forward AD for multiple directional derivatives is limited due to an intrinsic connection to matrix multiplication. Finally, we have shown that for simple ReLU networks, the enumeration of Clarke subgradients is computationally hard, in contrast to the enumeration of conservative gradients. The global picture is significantly different from the smooth case, with a well understood "cheap gradient" principle that yields "cheap p directional derivatives", illustrating the specificities of nonsmoothness. Our results confirm the centrality of conservative gradients in nonsmooth AD and machine learning: they generalize gradients with a clear "cheap principle", contrary to concurrent notions. An important open question in this context is the complexity of subgradients, or, in other words, the existence of a "cheap subgradient principle". We conjecture a negative answer in general.

A Appendix of Chapter 2

2.6 Further comments, discussion and technical elements

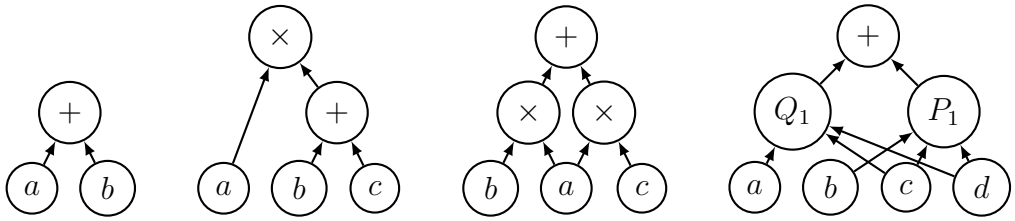
2.6.1 Comments on Section 2.4

Computational model in Section 2.4.1

DAG representation and examples 2.4.1: We start with a remark regarding representations of programs as directed acyclic graphs and use them to illustrate the model of computation proposed in the main text. It reduces to that of arithmetic circuit complexity for a dictionary composed of elementary arithmetic operations.

Remark 4 (Programs as directed graphs) A predecessor relation trivially describes a directed acyclic graph (DAG). Therefore, a program is equivalently represented as a DAG, nodes corresponding either to input variables (empty predecessor) or computation (nonempty predecessor). Directed edges connect predecessor nodes to their successors. Each computation node contains a lower-level program (with a single output), with the number of input edges being coherent with the number of arguments. The cost of a node is that of the underlying program and the cost of P is the sum of the costs of its nodes. Nodes without outer edges are output nodes. See examples in Appendix 2.6.1.

We represent programs using the DAG representation as in Remark 4. Let us define a simple dictionary $\mathcal{D} := \{+, \times\}$ and introduce a level 0 elementary program P_0 such that $P_0(a, b) = a + b$ meaning that P_0 computes the quantity $a + b$. P_0 is identified with $+$ from the dictionary. We also introduce a level 1 program P_1 such that $P_1(a, b, c) = a \times (b + c)$. We can construct an equivalent level 1 program, Q_1 such that $Q_1(a, b, c) = a \times b + a \times c$, in this case, we have $P_1 \sim Q_1$, or $[P_1] = [Q_1]$ since they compute the same quantity. The level 2 program P_2 is such that $P_2(a, b, c, d) = (a + b) \times (c + d) = Q_1(a, c, d) + P_1(b, c, d)$ and uses level 1 programs Q_1 and P_1 in its computation nodes. The Directed Acyclic Graphs (DAGs) representing these programs are given in Figure 2.2. Assuming $\text{cost}(+) = \text{cost}(\times) = 1$, we have $\text{cost}(P_0) = 1$, $\text{cost}(P_1) = 2$, $\text{cost}(Q_1) = 3$ and $\text{cost}(P_2) = \text{cost}(Q_1) + \text{cost}(P_1) + \text{cost}(\times) = 6$.



(a) P_0 (b) P_1 (c) Q_1 (d) P_2

Figure 2.2: DAG illustrating different programs with dictionary $\mathcal{D} := \{+, \times\}$. (a) $P_0(a, b) = a + b$, of level 0 which is identified with $+$ from the dictionary, (b) $P_1(a, b, c) = a(b + c)$, of level 1, (c) $Q_1(a, b, c) = ab + ac$, of level 1 and equivalent to P_1, (d) $P_2(a, b, c, d) = (a + b)(c + d) = Q_1(a, c, d) + P_1(b, c, d)$, of level 2.

2.6.2 Comments on Section 2.5

Forward AD and Clarke subgradients

[106] introduced the notion of lexicographic subdifferential, denoted here $\partial_L F$ for a Lipschitz function $F\colon \mathbb{R}^p \to \mathbb{R}$. The construction of $\partial_L F$ is based on successive local approximations of F with directional derivatives, and one has $\partial_L F(x) \subset \partial^c F(x)$ for all x such that the first term is well defined.

It is known that automatic differentiation can be used to compute directional derivatives, particularly the forward mode of automatic differentiation [8]. Based on this observation, Khan and Barton developed several algorithms to evaluate elements of $\partial^c F$, based on directional derivatives [103]–[105]. They concentrate on piecewise C^1 functions, see for example [101], and propose to handle compositional structures with different restrictions on the function class considered, such as functions in abs-normal forms [103], or broader classes [95], [104].

All these procedures either require to evaluate p directional derivatives [103], [104], or rely on forward chain rule propagation for lexicographic derivatives [95], [105], which also require to maintain p directional derivatives. For this reason, all these methods suffer from a multiplicative computational overhead ratio of the order of p in the worst case, and it is not known if this could be improved [95], although efforts have been made in this direction [117].

Matrix multiplications

Remark 5 The lower bound described in Theorem 4 is sharp for a linear ReLU network F as in (2.13) involving only square $p \times p$ matrices. Indeed, p directional derivatives of F in directions $a_1, \ldots, a_p$, can be computed with roughly $Lc(p)$ operations, using a matrix multiplication algorithm realizing the $c(p)$ bound, for example using the forward mode of AD [103], [104]. The naive P_F algorithm for forward evaluation performs roughly $2Lp^2$ operations which results in the bound (neglecting terms of order one in numerator and denominator),

$$\frac{\mathrm{comp}(F_d,\, \mathcal{D} \cup \{\mathrm{ReLU}, \mathrm{ReLU}'\})}{\mathrm{cost}(P_F)} \leq \frac{c(p)}{2p^2},$$

for this class of networks, to be compared with (2.7). Finally, we remark that in the smooth case such complexity estimates reduce to gradient computation which can

be done using backward algorithmic differentiation with a constant multiplicative overhead ratio.

We denote by F_d, the function $F_d\colon (y, A) \mapsto F_1'(0, y, A)$ which computes p directional derivatives at a given point. Setting $\omega = \limsup_{p\to\infty} \log(c(p))/\log(p)$, since P' is an arbitrary program implementing F_d, we have shown that asymptotically, for any $\epsilon > 0$

$$\sup_{p, F=[P_F], P_F \in \mathcal{P}(\mathcal{D}\cup\{\mathrm{ReLU}\})} \frac{\mathrm{comp}(F_d,\, \mathcal{D} \cup \{\mathrm{ReLU}, \mathrm{ReLU}'\})}{\mathrm{cost}(P_F)} \times p^{2-\omega+\epsilon} = +\infty,$$

where the supremum is taken over all p and all functions $F\colon \mathbb{R}^{p\times q} \to \mathbb{R}$ implemented by a program P_F with dictionary $\mathcal{D} \cup \{\mathrm{ReLU}\}$. It is not known whether $\omega > 2$.

2.7 Proofs related to Section 2.4.3

Proof of Theorem 3: Given a program P as in Section 2.4.1, the path differentiability of $[\mathcal{P}]$ is immediate by composition and the chain rule property. The associated conservative gradient D_P is constructed in [10].

We have the following cost estimates which can be deduced from the definition of the cost of a program in Section 2.4.1.

- **Algorithm 1** forward evaluation:

$$\mathrm{cost}(P) = \mathrm{cost}(\text{Algorithm 1}) = \sum_{i=p+1}^{m} \mathrm{cost}\,(g_i) \tag{2.8}$$

- **Algorithm 1** forward evaluation with derivatives: Algorithm 1' with gd_i instead of g_i on line 2

$$\mathrm{cost}(\text{Algorithm 1'}) = \sum_{i=p+1}^{m} \mathrm{cost}\,(gd_i) \tag{2.9}$$

- **Algorithm 2** backward AD cost:

$$\begin{aligned}\mathrm{cost}(\mathrm{backprop}(P)) =&\ \mathrm{cost}(\text{Algorithm 1'}) + \sum_{i=p+1}^{m} |\mathtt{pr}(i)|(\mathrm{cost}(+) + \mathrm{cost}(\times)) \\ =&\ \sum_{i=p+1}^{m} \mathrm{cost}\,(gd_i) + |\mathtt{pr}(i)|(\mathrm{cost}(+) + \mathrm{cost}(\times)).\end{aligned} \tag{2.10}$$

- **Algorithm 2** forward AD cost:

$$\begin{aligned}\mathrm{cost}(\mathrm{forprop}(P)) =&\ \mathrm{cost}(\text{Algorithm 1'}) + \sum_{i=p+1}^{m} p|\mathtt{pr}(i)|\mathrm{cost}(\times) + p(|\mathtt{pr}(i)| - 1)\mathrm{cost}(+) \\ =&\ \sum_{i=p+1}^{m} \mathrm{cost}\,(gd_i) + p|\mathtt{pr}(i)|\mathrm{cost}(\times) + p(|\mathtt{pr}(i)| - 1)\mathrm{cost}(+).\end{aligned} \tag{2.11}$$

Let us derive the complexity bound of Algorithm 1 according to Algorithm 2.

Backward AD complexity result: Using (2.10) and the fact that cost has value in $\mathbb{R}_+^*$, we have

$$\begin{aligned}\text{cost}(\text{backprop}(P)) &= \sum_{i=p+1}^{m} \text{cost}\left(gd_i\right) + |\texttt{pr}(i)|(\text{cost}(+) + \text{cost}(\times)) \\ &= \sum_{i=p+1}^{m} \text{cost}(g_i) \times \frac{\text{cost}\left(gd_i\right) + |\texttt{pr}(i)|(\text{cost}(+) + \text{cost}(\times))}{\text{cost}(g_i)} \\ &\leq \max_{i=p+1,m} \left(\frac{\text{cost}\left(gd_i\right) + |\texttt{pr}(i)|(\text{cost}(+) + \text{cost}(\times))}{\text{cost}(g_i)} \right) \sum_{i=p+1}^{m} \text{cost}(g_i),\end{aligned}$$

where the inequality is due to factorization by the maximal value. Using (2.8), we obtain

$$\text{cost}(\text{backprop}(P)) \leq \omega_b \times \text{cost}(P)$$

where ω_b is given in (2.5). This proves point (i).

Forward AD complexity result: Using (2.11) and the fact that cost has value in $\mathbb{R}_+^*$, we have

$$\begin{aligned}\text{cost}(\text{forprop}(P)) &= \sum_{i=p+1}^{m} \text{cost}\left(gd_i\right) + p|\texttt{pr}(i)|\text{cost}(\times) + p(|\texttt{pr}(i)| - 1)\text{cost}(+) \\ &= \sum_{i=p+1}^{m} \text{cost}(g_i) \times \frac{\text{cost}\left(gd_i\right) + p|\texttt{pr}(i)|\text{cost}(\times) + p(|\texttt{pr}(i)| - 1)\text{cost}(+)}{\text{cost}(g_i)} \\ &\leq \max_{i=p+1,m} \left(\frac{\text{cost}\left(gd_i\right) + p|\texttt{pr}(i)|\text{cost}(\times) + p(|\texttt{pr}(i)| - 1)\text{cost}(+)}{\text{cost}(g_i)} \right) \times \\ &\quad \sum_{i=p+1}^{m} \text{cost}(g_i),\end{aligned}$$

where the inequality is due to factorization by the maximal value. Using (2.8), we obtain

$$\text{cost}(\text{forprop}(P)) \leq \omega_f \times \text{cost}(P)$$

where ω_f is given in (2.5).

□

2.7.1 Justification of the complexity Table 2.1 of the $\mathcal{D}_{\text{ReLU}}$-Dictionary.

The proof of Corollary 2 follows from Theorem 3 by computing the relevant constants. They are shown in Table 2.1, let us justify the proposed numbers.

Case 1 $(\text{cost}(\times), \text{cost}(+))$ *Let us define* $g(a, b) = a \times b$. *To evaluate* g, *we need one operation from* $\mathcal{D}_{\text{ReLU}}$. *The derived program* d *related to* g, *should satisfy* $d(a, b) = (b, a)$ *which does not require additional operation. Therefore, from Assumption 1 we can deduce that* $\text{cost}(g) = 1$ *and* $\text{cost}(gd) = 1$. *We get the same result for* $\text{cost}(+)$ *by applying identical reasoning.*

Case 2 ($\text{cost}(\times c), \text{cost}(+c)$) *Let us define* $g(a) = c \times a$. *To evaluate* g, *we need one operation from* $\mathcal{D}_{\text{ReLU}}$. *The derived program* d *related to* g, *should satisfy* $d(a) = c$ *which does not require additional operation from* $\mathcal{D}'_{\text{ReLU}}$. *Therefore, from Assumption 1 we can deduce that* $\text{cost}(g) = 1$ *and* $\text{cost}(gd) = 1$. *We get the same result for* $\text{cost}(+c)$ *by applying identical reasoning.*

Case 3 ($\text{cost}(\log)$) *Let us define* $g(a) = \log(a)$. *To evaluate* g, *we need one operation from* $\mathcal{D}_{\text{ReLU}}$. *The derived program* d *related to* g, *should satisfy* $d(a) = 1/a$, *which requires the inverse operation from* $\mathcal{D}'_{\text{ReLU}}$. *Therefore, from Assumption 1 we can deduce that* $\text{cost}(g) = 1$ *and* $\text{cost}(gd) = 2$.

Case 4 ($\text{cost}(\exp)$) *Let us define* $g(a) = \exp(a)$. *To evaluate* g, *we need one operation from* $\mathcal{D}_{\text{ReLU}}$. *The derived program* d *related to* g, *should satisfy* $d(a) = g(a)$ *which does not require operation from* $\mathcal{D}'_{\text{ReLU}}$. *Finally, from Assumption 1 we can deduce that* $\text{cost}(g) = 1$ *and* $\text{cost}(gd) = 1$.

Case 5 ($\text{cost}(inv)$) *Let us define* $g(a) = \frac{1}{a}$. *To evaluate* g, *we need one operation from* $\mathcal{D}_{\text{ReLU}}$. *The derived program* d *related to* g, *should satisfy* $d(a) = \frac{-1}{a^2}$ *which requires one additional multiplication to compute the square and one* (-1) *multiplication operation from* $\mathcal{D}'_{\text{ReLU}}$. *Finally, from Assumption 1 we can deduce that* $\text{cost}(g) = 1$ *and* $\text{cost}(gd) = 3$.

Case 6 ($\text{cost}(\text{ReLU})$) *Let us define* $g(x) = \text{ReLU}(x) = max(x, 0)$. *To evaluate* g, *we need to evaluate the sign of* x. *The derived program* ReLU' *can be computed also from the sign of* x *without further operation. We have* $\text{cost}(g) = 1$ *by hypothesis, but it is also reasonable to consider* $\text{cost}(gd) = 1$ *as both operations only require sign evaluation of the same object.*

Remark 6 Since $\mathcal{D}_{\text{ReLU}}$ dictionary contains the ReLU function, we can build other non-smooth functions such as the maximum and the absolute value. For example, $\max\{x, y\} = \text{ReLU}(x - y) + y = \text{ReLU}(x - y) + \text{ReLU}(y) - \text{ReLU}(-y)$.

2.7.2 An extension of Table 2.1

The justifications of the following are similar to Section 2.7.1, simply taking into consideration different types of operations. Taking $c_{\text{nonlin}} = c_{\text{ReLU}} = 1$, we recover table 2.1. We replace ReLU by $\times$ReLU which corresponds to its usage in practice and allows us to balance the cost of ReLU operations and that of multiplications.

The justification is the same as in Section 2.7.1 taking into consideration different types of operations. For the $\times$ReLU operation, the justification is as follows.

Case 7 ($\times\text{cost}(\text{ReLU})$) *The operation has two argument and requires one sign evaluation and one multiplication in the worst case, so we assign it the cost* $1 + c_{\text{ReLU}}$. *The differentiated program* d *should compute the function* $(a, b) \mapsto (\text{ReLU}(b), a \times \text{ReLU}'(b))$. *One can write a program to compute jointly* g *and* d *as follows: return* $(a \times b, b, a)$ *if* $b \geq 0$ *and* $(0, 0, 0)$ *if* $b < 0$. *This only requires a bit sign check which cost is* c_{ReLU} *and a multiplication. We therefore model this operation such that* $\text{cost}(gd) = \text{cost}(g) = 1 + c_{\text{ReLU}}$.

Further refinements could be considered including various type of computational operations, such as memory moves, these are beyond the scope of the present paper.

Table 2.2: Extension of cost table. $c_{\text{nonlin}} \geq 1$ is the cost of nonlinear operations and $c_{\text{ReLU}} \geq 0$ is the cost of sign evaluation for ReLU or ReLU$'$.

g	$(+, \times)$	$(+c, \times c)$	log	exp	inv	$\times$ReLU
$\text{cost}(g)$	1	1	c_{nonlin}	c_{nonlin}	c_{nonlin}	$1 + c_{\text{ReLU}}$
$\lvert\texttt{pr}\rvert$	2	1	1	1	1	2
$\text{cost}(gd)$	1	1	$2c_{\text{nonlin}}$	c_{nonlin}	$c_{\text{nonlin}} + 2$	$1 + c_{\text{ReLU}}$
$\frac{\text{cost}(gd)}{\text{cost}\,(g)}$	1	1	2	1	$\frac{c_{\text{nonlin}}+2}{c_{\text{nonlin}}}$	1
$\frac{\text{cost}(\times)\lvert\texttt{pr}\rvert}{\text{cost}\,(g)}$	4	2	$\frac{1}{c_{\text{nonlin}}}$	$\frac{1}{c_{\text{nonlin}}}$	$\frac{1}{c_{\text{nonlin}}}$	$\frac{2}{1+c_{\text{ReLU}}}$
$\frac{\text{cost}(gd) + 2\text{cost}(\times)\lvert\texttt{pr}\rvert}{\text{cost}\,(g)}$	5	3	≤ 4	≤ 3	≤ 5	≤ 5

2.7.3 Additional elementary nonsmooth programs and cost examples

For simplicity, we do not discuss the dictionary and its related derived dictionary as there are many possibilities, one of them being $\mathcal{D}_{\text{ReLU}}$ and $\mathcal{D}'_{\text{ReLU}}$ as all the considered operations can be equivalently expressed with ReLU. We use the same framework as in 2.7.2 and we identify the cost of comparing two real numbers with $c_{\text{ReLU}} > 0$. For each program g and associated derived program d, we let

$$\omega = \frac{\text{cost}(gd) + 2\text{cost}(\times)\lvert\texttt{pr}\rvert}{\text{cost}\,(g)}$$

Table 2.3: Extension of cost table. $c_{\text{nonlin}} \geq 1$ is the cost of nonlinear operations and $c_{\text{ReLU}} \geq 0$ is the cost of sign evaluation for ReLU or ReLU$'$. For simplicity c_{ReLU} is abbreviated c_{R} and c_{nonlin} is abbreviated c_{nl}

g	$(+, \times)$	$\lvert\cdot\rvert$	ELU	3×3-max-pool	$\lVert\cdot\rVert_\infty$	$\lVert\cdot\rVert_1$
$\text{cost}(g)$	1	$1 + c_{\text{R}}$	$2 + c_{\text{R}} + c_{\text{nl}}$	$153 + 8c_{\text{R}}$	$n + 2nc_{\text{R}} - 1$	$n(2 + c_{\text{R}}) - 1$
$\lvert\texttt{pr}\rvert$	2	1	1	9	n	n
$\text{cost}(d, g)$	1	$1 + c_{\text{R}}$	$2 + c_{\text{R}} + c_{\text{nl}}$	$153 + 8c_{\text{R}}$	$n + 2nc_{\text{R}} - 1$	$n(2 + c_{\text{R}}) - 1$
$\frac{\text{cost}(gd)}{\text{cost}\,(g)}$	1	1	1	1	1	1
$\frac{\text{cost}(\times)\lvert\texttt{pr}\rvert}{\text{cost}\,(g)}$	4	$\frac{1}{1+c_{\text{R}}}$	$\frac{1}{2+c_{\text{R}}+c_{\text{nl}}}$	$\frac{9}{153+8c_{\text{R}}}$	$\frac{n}{n+2nc_{\text{R}}-1}$	$\frac{n}{n(2+c_{\text{R}})-1}$
ω	5	≤ 3	≤ 2	≤ 1.12	≤ 3	≤ 2

Case 8 (Absolute value and Leaky-ReLU) *Recall that* $|x| = x$ *if* $x > 0$ *and* $-x$ *otherwise. Similarly Leaky-*$\text{ReLU}(x) = x$ *if* $x > 0$ *and* ax *otherwise, for some parameter* $a \in (0, 1)$ *so that both cases are exactly the same. The reasoning and result are exactly the same for both operations so we treat the absolute value. The construction is similar as what was proposed for* $\times\text{cost}(\text{ReLU})$ *treated in the previous section.*

Let g be a program to evaluate $|\cdot|$, in the worst case it requires one sign evaluation and one multiplication so that $\mathrm{cost}(g) = 1 + c_{\mathrm{ReLU}}$. *Similarly it is possible to built a program which returns $(x, 1)$ if $x > 0$ and $(-x, -1)$ otherwise, this computes (gd) and require the exact same operations so that* $\mathrm{cost}(gd) = \mathrm{cost}(g) = 1 + c_{\mathrm{ReLU}}$.

Case 9 (ELU)

$$f(x) = \begin{cases} x & \text{if} \quad x \geq 0 \\ a(e^x - 1) & \text{if} \quad x < 0 \end{cases} \text{ with } a > 0.$$

Let g be a program to evaluate the ELU function, it requires a sign evaluation and in the worst case one nonlinear operation to evaluate e^x, one multiplication to evaluate ae^x, and one substraction to evaluate $ae^x - a$. Therefore, $\mathrm{cost}(g) = c_{\mathrm{ReLU}} + c_{\mathrm{nonlin}} + 2$. *The derived program d requires the same sign and returns 1 or ae^x depending on the sign. This does not require additional operation and therefore the joint computation of g and d satisfies* $\mathrm{cost}(gd) = \mathrm{cost}(g)$.

Case 10 (max-m-linear) *Set n a number of inputs and $m \geq 2$ a number of linear functions which are parameters, represented by a matrix A and a fixed input vector of size n represented by $x \in \mathbb{R}^n$. Setting* $\max_m \colon \mathbb{R}^m$ *to $\mathbb{R}$ the function which evaluates the maximum of m numbers, we consider g a program which evaluates the function $A \mapsto \max_m(Ax)$. Recall that x is fixed so that the number of inputs is $m \times n$. The multiplication requires $m \times (2n - 1)$ multiplications and additions and the evaluation of* $\max_m$ *requires $(m-1)c_{\mathrm{ReLU}}$ as it requires $m - 1$ pairwise comparisons. We therefore have* $\mathrm{cost}(g) = m \times (2n - 1) + (m - 1)c_{\mathrm{ReLU}}$.

As for the derived program d, setting $M_i = 0$ except for row number i which attains the maximum in g which is set to x, we have an element of a conservative gradient for g. It is possible to jointly compute $g(A)$ and $d(A)$ by invoking a program which returns $((Ax)[i], M_i)$ where i is any index realizing the max and M_i is as discussed. This does not require more operations and we have therefore $\mathrm{cost}(gd) = \mathrm{cost}(g) = m \times 2n - 1 + (m - 1)c_{\mathrm{ReLU}}$

Case 11 (Two dimensional max-pooling (3×3-max-pool)) *We consider a kernel of size 3×3 for simplicity. The goal is to differentiate with respect to the kernel weights for a fixed input. Let g denote a program implementing such a function, it is of the same form as max-m-linear except that the matrix A is of size 9×25 (padding values at the boundary of the 3×3 patch, this gives $5 \times 5 = 25$ inputs and 9 outputs), but it is sparse and can be parametrized by only 9 values, and the evaluation of the linear function for a fixed 5×5 input only requires $9 \times (9 + 8) = 153$ addition and multiplications. We then take the maximum of these 9 outputs so that and* $\mathrm{cost}(g) = 153 + 8c_{\mathrm{nonlin}}$. *For the same reason as max-m-linear, we have* $\mathrm{cost}(gd) = \mathrm{cost}(g) = 153 + 8c_{\mathrm{nonlin}}$.

Case 12 (l_1-norm, $\|\cdot\|_\infty$) *Denote by g a program which evaluate the l_1 norm on $\mathbb{R}^n$. It has n inputs. In the worst case, its evaluation can be done with $n - 1$ addition, n multiplication by -1 and n pairwise comparisons. Therefore we have* $\mathrm{cost}(g) = 2n + nc_{\mathrm{ReLU}} - 1$. *For the same reasons as all examples before, it is possible to identify a derived program d without requiring additional operation so that* $\mathrm{cost}(gd) = \mathrm{cost}(g) = 2n + nc_{\mathrm{ReLU}} - 1$.

Case 13 (Median of n numbers) *Denote by g a program that evaluates the median of n numbers. This can be done by sorting the n numbers and outputting the value corresponding to $\lfloor \frac{n}{2} \rfloor$, which requires roughly $n \log(n)$ operations, depending on the algorithm used. The sorting operation is a permutation, one could apply the same permutation to the vector $(1, 2, \ldots, n)$ without additional operation required. The number at position $\lfloor \frac{n}{2} \rfloor$, call it i, is the index of the value associated with the median. Setting d to be the null vector in $\mathbb{R}^n$ with value 1 at position i only, we have a selection in a conservative gradient for the median with no additional operation required. Therefore in this case* $\mathrm{cost}(g) = \mathrm{cost}(gd)$.

Case 14 (Selection functions) *This example encompasses virtually all examples used in machine learning and extends the median example above. Assume that $f \colon \mathbb{R}^p \to \mathbb{R}$ is locally Lipschitz, given in the form*

$$f(x) = f_{s(x)}(x)$$

where $s \colon \mathbb{R}^p \to \{1, \ldots, m\}$ is an index selection function, and for each $i = 1, \ldots, m$, $f_i \colon \mathbb{R}^p \to \mathbb{R}$ is a C^2 function. Let g be a program computing f, one possibility is to first evaluate $s(x)$ at cost c_s and then evaluate $f_{s(x)}(x)$ at cost c_f. As shown in [11], under very mild restrictions on s and f (which should be expressed with logarithms, polynomials, exponentials etc ...), the function

$$x \mapsto \nabla_{s(x)} f(x)$$

is a conservative gradient for f. It can be seen that it is possible to evaluate jointly (g, gd) by first computing s, at a cost c_s, then evaluate f_s and ∇f_s jointly at a cost c_∇.

$$\begin{aligned} \mathrm{cost}(g) &= c_s + c_f \\ \mathrm{cost}(gd) &= c_s + c_\nabla \\ \frac{\mathrm{cost}(gd)}{\mathrm{cost}(g)} &= \frac{c_s + c_\nabla}{c_s + c_f} \leq \frac{c_s + 5c_f}{c_s + c_f} \end{aligned}$$

where we used $c_\nabla \leq 5c_f$, the cheap gradient principle for smooth programs. This ratio is close to 5 if c_s is negligible, we recover the usual ratio for smooth programs. It is close to 1 if c_s dominates, which is the case in the median example where f_s just corresponds to coordinate number s of the input and has a constant derivative.

2.8 Proofs of Section 2.5.1

2.8.1 Proof of the main result

Proof of Theorem 4: Let $U \in \mathbb{R}^{p \times p}$ be an orthogonal matrix with entries in $\{-1, 1\}$ which columns are denoted by $u_1, \ldots, u_p$ (with squared norm p). Assume that we have as variables a matrix $M \in \mathbb{R}^{p \times p}$ and two matrices $A, B \in \mathbb{R}^{p \times p}$ with columns $a_1, \ldots, a_p$ and $b_1, \ldots, b_p$ respectively.

Consider the function

$$F \colon (x, B, M) \mapsto \frac{1}{p} \sum_{i=1}^{p} |[UB^T Mx]_i|.$$

The pair (M, B) will be identified as y in the statement of the theorem. Considering the dictionary of elementary functions $\{+, \times, \mathrm{ReLU}, +c, \times c\}$, F has a representation as a program P_F using the identity $|t| = \mathrm{ReLU}(t) + \mathrm{ReLU}(-t)$ for all $t \in \mathbb{R}$. We may construct P_F such that $\mathrm{cost}(P_F) = 6p^2 + 2p \leq 8p^2$ where we count $2p^2 - p$ operation for each matrix vector multiplication to evaluate UB^TMx (there are three of them), p multiplication by -1 to evaluate $-UB^TMx$, $2p$ application of ReLU (on UB^TMx and $-UB^TMx$), p additions of ReLU outputs to evaluate p applications of the absolute value, $p-1$ for the outer sum and 1 for the division. Now consider the constraints

$$\mathrm{sign}(UB^TMa_i) = u_i, \quad i = 1, \dots p. \tag{2.12}$$

The set of matrices A, B, M satisfying this constraint is an open set, call it S. We now restrict our attention to this open set and argue that $\mathrm{cost}(P')$ does not change if the input variables are constrained to be in S.

We have for all $i = 1, \dots, p$ and $(A, B, M) \in S$, the following directional derivatives with respect to variable x

$$F_1'(0, B, M, a_i) = \frac{1}{p}\,\mathrm{sign}(UB^TMa_i)^T UB^TMa_i = \frac{1}{p} u_i^T UB^TMa_i = b_i^TMa_i.$$

Setting the function $G\colon (A, B, M) \mapsto \sum_{i=1}^p F_1'(0, B, M, a_i) = \mathrm{Tr}(MAB^T)$, we have that G is a polynomial and $\nabla_M G(A, B, M) = \sum_{i=1}^p b_i a_i^T = BA^T$. Note that this does not depend on M.

Fix P' any program implementing the directional derivatives function $(y, A) \mapsto F_1'(0, y, A)$ of F described above, with dictionary $\{+, \times, \mathrm{ReLU}, \mathrm{ReLU}', +c, \times c\}$, as in the statement of the theorem.

Claim 1 *There is a program P_2 on dictionary $\mathcal{D} = \{+, \times, +c, \times c\}$ such that $G = [P_2]$ (on the whole space) and $\mathrm{cost}(P_2) \leq \mathrm{cost}(P') + p$.*

We use the DAG representation of programs as in Remark 4. Therefore P' is described by a DAG which node are either input nodes or computation nodes implementing functions from $\mathcal{D}'_{\mathrm{ReLU}}$. We will modify the program by simple modifications of the computation nodes. We may obtain a program P_0 implementing G on S with dictionary $\mathcal{D}'_{\mathrm{ReLU}}$ with $\mathrm{cost}(P_0) \leq \mathrm{cost}(P') + p$ by summing the outputs of P'. The ReLU' nodes in P_0 represent a semialgebraic function [118], [119] with values in a finite set. Therefore, there is a dense open semialgebraic set on which all ReLU' nodes in P_0 are locally constant [118, Theorem 6.7]. Reducing S if necessary, we obtain a program P_1 on dictionary $\mathcal{D}_{\mathrm{ReLU}}$ such that $P_1 \sim P_0$ on S by replacing each ReLU' node in P_0 by the corresponding constants. We have $\mathrm{cost}(P_1) \leq \mathrm{cost}(P_0)$ (we replace computing nodes by constants). By Lemma 1, there is a program P_2 on $\mathcal{D}$ such that $\mathrm{cost}(P_2) = \mathrm{cost}(P_1) \leq \mathrm{cost}(P_0) \leq \mathrm{cost}(P') + p$ and $G = [P_2]$ (on the whole space). This proves the claim.

We may obtain a program D_2 implementing $\nabla_M G$ with dictionary $\mathcal{D}$ by backward algorithmic differentiation on P_2, that is $D_2 = \mathrm{backprop}(P_2)$. we have therefore

$$\begin{aligned}
\mathrm{comp}(BA^T, \mathcal{D}) &\leq \mathrm{cost}(D_2) \\
&\leq \mathrm{cost}(P_2, D_2) \\
&\leq 5\mathrm{cost}(P_2) \\
&\leq 5p + 5\mathrm{cost}(P'),
\end{aligned}$$

where the first inequality is because D_2 is a program computing BA^T for all A, B on dictionary $\mathcal{D}$, the second is because adding computation increases the cost, the third is a property of backward algorithmic differentiation on $\mathcal{D}$ and the last one is by construction of P_2. Note that $\text{comp}(BA^T, \mathcal{D}) = c(p)$ by definition, therefore we have the claimed lower bound

$$\frac{\text{cost}(P')}{\text{cost}(P_F)} \geq \frac{c(p) - 5p}{5\text{cost}(P_F)} = \frac{c(p) - 5p}{8p^2}.$$

□

2.8.2 An additional Lemma

Lemma 1 *Let $Q\colon \mathbb{R}^p \to \mathbb{R}$ be a polynomial and P_1 be a program (without loss of generality of level 1) on the dictionary $\mathcal{D}_1 = \{+, \times, \text{ReLU}, +c, \times c\}$, such that $Q = [P_1]$ for all inputs restricted to an open set $S \subset \mathbb{R}^p$. Then there is a level 1 program P_2 on the dictionary $\mathcal{D} = \mathcal{D}_1 \setminus \{\text{ReLU}\} = \{+, \times, +c, \times c\}$ such that $Q = [P_2]$ (for all inputs in $\mathbb{R}^p$). Furthermore, if $\text{cost}(\text{ReLU}) = \text{cost}(\times c)$, then, $\text{cost}(P_2) = \text{cost}(P_1)$.*

Proof : We use the DAG representation of programs as in Remark 4. Therefore P_1 is described by a DAG which node are either input nodes or computation nodes implementing functions from $\mathcal{D}_1$. The function computed by P_1 as well as each of its nodes are semi-algebraic [118]–[120]. For each ReLU node in the graph representing P_1 (assume that there are N of them) we associate a number: the function ReLU' evaluated on its input (with the convention that $\text{ReLU}'(0) = 0$). This defines a semialgebraic function $G\colon \mathbb{R}^p \to \{0, 1\}^N$. As it has values in a finite set, by semialgebraicity, there is an open subset of $S' \subset S$ such that G is constant on S [118, Theorem 6.7]. Consider P_2 which computation graph is the same as that of P_1 except that each absolute value node is replaced by multiplication by the corresponding ReLU' value (which is constant on S'). Then $Q = [P_1] = [P_2]$ for all inputs in the open set S'. All computation nodes of programs on $\mathcal{D}$ are multivariate polynomials and two polynomials which agree on an open set are equal globally. This concludes the proof. □

2.9 Proofs of Section 2.5.3

We investigate in this section the hardness of finding a Clarke subgradient for programs defined on the elementary dictionary $\mathcal{D}_0 = \{+, -, \text{ReLU}\}$. We start with an equivalent representation of these programs as linear ReLU networks with skip connections and specific weight matrices. This equivalence preserve representation size up to polynomial factors. We will then prove a hardness result on such ReLU networks. This will provide proof arguments for Theorem 5 by the polynomial time equivalence of the two representation. We proceed similarly to prove Proposition 3, using the equivalence with the two representations.

2.9.1 Polynomial time equivalence with linear ReLU networks with skip connections

Given a set of matrices $M_1 \in \{-1,0,1\}^{p_1 \times p}$, $M_2 \in \{-1,0,1\}^{p_2 \times p_1}$, $\dots M_{L-1} \in \{-1,0,1\}^{p_{L-1} \times p_{L-2}}$, $M_L \in \{-1,0,1\}^{1 \times p_{L-1}}$ we consider the function $F\colon \mathbb{R}^p \to \mathbb{R}$,

$$F\colon x \mapsto M_L \Phi_{L-1}(M_{L-1}\Phi_{L-2}(\dots \Phi_1(M_1 x))). \tag{2.13}$$

where $\Phi_i\colon \mathbb{R}^{p_i} \to \mathbb{R}^{p_i}$ are given functions which apply to each coordinate, an activation function which is either the identity or the ReLU function. There is an obvious notion of size for this representation, corresponding to the number of free parameters (matrix entries and coordinates on which ReLU or identity is applied), the size of the representation is $p_{L-1} + \sum_{i=1}^{L-1} p_i \times p_{i-1} + p_i$.

A function F given in (2.13) can be represented by a program on $\mathcal{D}_0$ of equivalent size, this correspond to a naive implementation. Similarly, any program $P \in \mathcal{P}(\mathcal{D}_0)$ on p inputs and with a single output can be represented by a network as in (2.13) which size is at most $18\text{cost}(P)^3$. Indeed, we may assume that $\text{cost}(P) \geq p/2$ without loss of generality, otherwise, the program would not perform operations on some of the input variables and it could be simplified by removing variables which do not affect the output. Recall that m in Algorithm 1 is the memory footprint of P, in our case, it is $m = p + \text{cost}(P)$, the number of inputs plus the total number of operations. Note that we have $m \leq 3\text{cost}(P)$. Each operation $+$, $-$ or ReLU in the program can be represented by a $m \times m$ matrix composed with a certain $\Phi\colon \mathbb{R}^m \to \mathbb{R}^m$ which contribution to the Relu network size is at most $(m^2+m) \leq 2m^2 \leq 18\text{cost}(P)^2$ since m is integer and $m \leq 3\text{cost}(P)$. There are $\text{cost}(P)$ such operations so that a program can be represented equivalently by linear Relu network, with $L = \text{cost}(P)$ layers which contribution to the network size is at most $18\text{cost}(P)^2$ so that the size of the resulting network is at most $18\text{cost}(P)^3$, which is the desired bound since.

We have shown that working with functions represented as in equation (2.13) is equivalent to work with programs in $\mathcal{P}(\mathcal{D}_0)$ as it is possible to switch from one to the other at a cost of an increase of the representation size which is only cubic. Therefore we will from now on work with functions represented as linear relu networks with skip connections as in (2.13), and NP-hardness or polynomial time results on such function will be valid on $\mathcal{P}(\mathcal{D}_0)$ by the construction above.

2.9.2 Further properties of Linear ReLU networks

Throughout this section F denotes a with representation as in (2.13). This function is positively homogeneous, it satisfies $F(0) = 0$ and it. By piecewise linearity, its Clarke subdifferential is a polyhedron (see e.g., [121], [122]). The Clarke subdifferential is a conservative gradient for this function, and we will associate to it a different conservative gradient, associated to Algorithm 2

Definition 4 (Autodiff conservative gradient) We consider a specific conservative gradient for F, it is given by $D_F^a(x) = \{M_1^T D_1 M_2^T D_2 \dots M_{L-1}^T D_{L-1} M_L^T\}$, where for $i = 1, \dots, L-1$, D_i is a diagonal matrix which entries respects the sign pattern of the corresponding activation function: 1 if the activation is identity, 0

if the activation is ReLU and the input is negative, 1 if the input is positive and all elements in $[0, 1]$ if the input is null. We have in particular

$$D_F^a(0) = \{M_1^T D_1 M_2^T D_2 \dots M_{L-1}^T D_{L-1} M_L^T\} \tag{2.14}$$

where in this case, diagonal entries of matrices D_i corresponding to ReLU activations are arbitrary in $[0, 1]$ and the remaining diagonal entries are 1 (corresponding to identity activations).

The autodiff conservative gradient is associated with the algorithmic differentiation of a natural numerical program implementing F as in Subsection 2.4.2. Furthermore, one can check that given a program $P \in \mathcal{P}(\mathcal{D}_0)$, after the transformation outlined in Section 2.9.1, we have that D_F^α coincides with D_P in Theorem 3. In the following definition, D_F could be,for example, the Clarke subdifferential of F or the algorithmic differentiation conservative gradient D_F^a.

We consider the following problem.

Problem 1 (Conservative gradient enumeration) Given matrices $M_1 \in \mathbb{R}^{p_1 \times p}$, $M_2 \in \mathbb{R}^{p_2 \times p_1}$, $\dots M_{L-1} \in \mathbb{R}^{p_{L-1} \times p_{L-2}}$, $M_L \in \mathbb{R}^{1 \times p_{L-1}}$, and functions $\Phi_1, \dots, \Phi_{L-1}$, consider $F \colon \mathbb{R}^p \to \mathbb{R}$ the associated linear ReLU network with skip connections in (2.13), $x \in \mathbb{R}^p$ and $D_F \colon \mathbb{R}^p \rightrightarrows \mathbb{R}^p$ a conservative gradient for F. Compute two distinct elements in $D_F(x)$ or one element if it is a singleton.

This problem enters the field of computational complexity as we have associated to it a representation size corresponding to the number of "free parameters" to be chosen: each matrix entry and the activation (ReLU or identity) corresponding to each coordinate, resulting in a number of parameters $p_{L-1} + \sum_{i=1}^{L-1} p_i \times p_{i-1} + p_i$. In what follows, we will consider integral or rational entries for matrices and input x with the common notion of bit size. [123].

Clarke enumeration is NP-hard for ReLU networks

The decision version of Problem 1, under the same assumptions, is to decide if there exists two distinct elements in $D_F(x)$, that is, decide if $D_F(x)$ is not reduced to a singleton.

Theorem 6 (Finding two Clarke subgradients is NP-Hard) *Decision version of problem* (1) *with matrix and vector entries in* $\{-1, 0, 1\}$ *and* $D_F = \partial^c F$ *is NP-hard.*

Sketch of proof: We encode a boolean formula π on p boolean variable, in a linear ReLU network with p inputs, of size proportional to that of π. We do so by replacing "or" operations by maxima, "and" operations by minima, negation by multiplication by -1 and adding ReLU operations to the result. Using Lemma 3 in appendix 2.9.5, the resulting F is represented by a linear ReLU network. By construction, 0 is a global minimum of F so $0 \in \partial^c F(0)$, and F takes positive values if and only if π is satisfiable if and only if $\partial^c F(0) \neq \{0\}$. We detail this proof in coming sections.

Theorem 6 illustrates the hardness enumerating Clarke subgradients of linear ReLU networks. For F as in (2.13) and $x \in \mathbb{R}^p$, $\partial^c F(x)$ is not a singleton if and only if F is not differentiable at x, therefore:

Corollary 3 (Deciding non-differentiability of a NN is NP-Hard) *Given a linear ReLU network as in* (2.13) *with matrices as in Theorem 6 and $x \in \mathbb{R}^p$, deciding if F is not differentiable at x is NP-hard.*

In the coming section, we will provide a proof for Theorem 6 and Corollary 3. By the polynomial time equivalence of the representation of programs in $\mathcal{P}(\mathcal{D}_0)$ and functions as in (2.13) detailed in Section 2.9.1, this proves Theorem 5.

We add a remark on lexicographic subdifferential. It follows from [95, Proposition 2.7] that, for linear ReLU network F as in (2.13), the lexicographic subdifferential [106] is the set of neighboring gradients and is contained in Clarke subdifferential.

Corollary 4 (Finding two lexicographic subgradients is NP-Hard) *If D_F is the lexicographic subdifferential, Theorem 6 remains true.*

2.9.3 Proof of the main hardness result

Preliminary on 3-SAT We will use reduction to 3-SAT problem which is among the most well known NP-complete problems. Recall that a boolean formula is built from boolean variables, and operators: AND (conjunction, denoted $\wedge$) OR (disjunction, $\vee$) and NOT (negation, $\neg$). A literal, is either a variable or the negation of a variable. A clause is a disjunction of literals (or a single literal). A formula is in conjunctive normal form (CNF), if it is a conjunction of clauses or a clause. 3-SAT is the decidability problem associated to CNF formulas with clauses containing 3 literals, such formulas are called 3-CNF formulas.

Example 3 The formula $(b_1 \vee b_2 \vee \neg b_3) \wedge (b_1 \vee b_4 \vee \neg b_5) \wedge (\neg b_2 \vee \neg b_3 \vee b_6)$ is 3-CNF with 6 boolean variables $b_1, \dots, b_6$ and 3 clauses.

Problem 2 (3-SAT) *Given $p, n \in \mathbb{N}$ and a boolean function π with p boolean arguments $b_1, \dots, b_p$ represented by a 3-CNF formula with n clauses, decide if there exists an assignment $(b_1, \dots, b_p) \in \{0,1\}^p$ such that $\pi(b_1, \dots, b_p) = 1$.*

Proof of Theorem 6:

The reduction is to 3-SAT.

Consider a 3-CNF function π in p variables $b_1, \dots, b_p$ with n clauses of size 3. We may assume without loss of generality that n is of the form 2^k for $k \in \mathbb{N}$ by adding clauses which are always true and increasing the number of clauses by a factor at most 2. We will consider p real variables $x_1, \dots, x_p$. Consider the first clause of π, say for example $(b_1 \vee b_2 \vee \neg b_3)$. We associate to each literal the corresponding variable x if the literal is equal to a variable, and $-x$ if it is the negation of the corresponding variable, for example $x_1, x_2, -x_3$. These are combined using ReLU $\circ$ max resulting in the expression $\mathrm{ReLU}(\max\{x_1, x_2, -x_3\})$.

We proceed similarly for each clause, we obtain $n = 2^k$ expressions involving ReLU $\circ$ max where the max is over three numbers. The max of 3 numbers is the same as the max of 4 numbers (by copying one of the inputs) and, according to Lemma 3, can be represented by a ReLU network with 2 ReLU layers of size at most $3 \times 2 = 6$ with weight matrices in $\{-1, 0, 1\}$.

We may therefore represent the n ReLU $\circ$ max expressions with a network with p inputs and n outputs, with 3 ReLU layers (2 for each max and one for the outer

ReLU) of size at most $6n$ (6 nodes for each max) involving matrices with entries in $\{-1, 0, 1\}$. These expressions are combined using the operator min applied to the $n = 2^k$ clause. Thanks to Lemma 3 again, using $\min\{a, b\} = -\max\{-a, -b\}$, the max over the 2^k numbers can be expressed with k layers of size at most $3 \times 2^{k-1} = \frac{3}{2}n$

We call the resulting network F. It has a representation as in (2.13), with matrices with entries in $Z_3 = \{-1, 0, 1\}$ as in Problem 1. It contains $\log_2(n) + 3$ ReLU layers of size at most $6n$ and it has therefore a description which size is polynomially bounded in n which is proportional to the bit size representation of the 3-CNF formula π.

Example 4 If the 3-CNF formula is given by $(b_1 \vee b_2 \vee \neg b_3) \wedge (b_1 \vee b_4 \vee \neg b_5) \wedge (\neg b_2 \vee \neg b_3 \vee b_6) \wedge (b_2 \vee \neg b_2 \vee b_6)$ with $p = 6$ boolean variables and $n = 4$ clauses, we will obtain a network computing the following expression in 6 real variables $x_1, \ldots, x_6$:

$$\begin{aligned} & F(x_1, \ldots, x_6) \\ = \; & \min(\mathrm{ReLU}(\max(x_1, x_2, -x_3)), \mathrm{ReLU}(\max(x_1, x_4, -x_5)), \\ & \quad \mathrm{ReLU}(\max(-x_2, -x_3, x_6)), \mathrm{ReLU}(\max(x_2, -x_2, x_6)))). \end{aligned}$$

We have the following rules for min and max over real numbers a, b, c (we use the convention $\mathrm{sign}(0) = 0$).

- $\max(a, b, c) > 0 \quad \Leftrightarrow \quad (a > 0) \vee (b > 0) \vee (c > 0).$
- $\max(a, b, c) > 0 \quad \Leftrightarrow \quad \max(\mathrm{sign}(a), \mathrm{sign}(b), \mathrm{sign}(c)) > 0.$
- $\min(a, b, c) > 0 \quad \Leftrightarrow \quad (a > 0) \wedge (b > 0) \wedge (c > 0).$
- $\min(a, b, c) > 0 \quad \Leftrightarrow \quad \min(\mathrm{sign}(a), \mathrm{sign}(b), \mathrm{sign}(c)) > 0.$
- $a > 0 \quad \Leftrightarrow \quad (-a < 0) \quad \Leftrightarrow \quad \mathrm{sign}(a) > 0.$
- $\mathrm{ReLU}(\max(\mathrm{sign}(a), \mathrm{sign}(b), \mathrm{sign}(c))) \in \{0, 1\}.$

Because of the $\min \circ \mathrm{ReLU}$ structure, we have $F(x) \geq 0$ for all x, furthermore, $F(0) = 0$, so that 0 is a global minimum of F and $0 \in \partial^c F(0)$. For any x, we have $F(x) > 0$ if and only if the output of each max is positive, if and only if each max clause contains a positive argument. We therefore have that $F(x) > 0$ if and only if $F(\mathrm{sign}(x)) > 0$ where sign is the coordinatewise application of the sign, taking value 0 at 0.

We have the following chain of equivalence

$$\begin{aligned} & \partial^c F(0) \neq \{0\} \\ \Leftrightarrow \quad & \exists x \in \mathbb{R}^p, \quad F(x) \neq 0 \\ \Leftrightarrow \quad & \exists x \in \mathbb{R}^p, \quad F(x) > 0 \\ \Leftrightarrow \quad & \exists x \in \mathbb{R}^p, \quad x_i \neq 0 \, (\forall i = 1, \ldots, p) \quad F(x) > 0 \\ \Leftrightarrow \quad & \exists x \in \mathbb{R}^p, \quad x_i \neq 0 \, (\forall i = 1, \ldots, p) \quad F(\mathrm{sign}(x)) > 0 \\ \Leftrightarrow \quad & \exists x \in \{-1, 1\}^p, F(x) > 0 \\ \Leftrightarrow \quad & \exists x \in \{-1, 1\}^p, \quad \pi(b) = 1, \quad b_i = \mathbb{I}(x_i = 1) \quad (i = 1 \ldots p), \end{aligned}$$

where $\mathbb{I}$ outputs 1 if the boolean argument is true, and 0 otherwise. The first equivalence is by Lemma 2, the second is because $F \geq 0$, the third is because F is continuous, the fourth is by the discussion above and the fifth is obvious because all possible $\{-1, 1\}$ patterns can be described as coordinatewise sign applied vectors in $\mathbb{R}^p$ with nonzero entries. For the last equivalence, for $x_i \in \{-1, 1\}$ we set $b_i = 0$ if $x_i = -1$ and $b_i = 1$ if $x_i = 1$. Each ReLU $\circ$ max applied to the sign vector corresponds to a clause and its output is in $\{0, 1\}$. The output of each ReLU $\circ$ max clause is 1 if and only if at least one of its argument is 1, if and only if one of the litteral of the corresponding disjunction is 1 if and only if the disjunction applied to the corresponding boolean variables is true. Otherwise, it is 0. Similarly, the min combination has positive output if and only if all max outputs are 1 if and only if all the disjunctions applied to variables b_i are true.

This shows that Problem 1 is NP-hard, because $0 \in \partial^c F(0)$ and $\partial^c F(0) \neq \{0\}$ if and only if there exists two distinct elements in $\partial^c F(0)$. □

2.9.4 Proof of feasibility for autodiff conservative gradient

The counterpart of Problem 1 for AD conservative gradient in Definition 4 is tractable, illustrating a major computational difference between Clarke subdifferential and AD conservative gradient. The proof is in Section 2.9.4, by reduction to a graph shortest path problem. By the polynomial time equivalence between linear ReLU network and programs on $\{+, -, \text{ReLU}\}$ proved in Section 2.9.1, this proves Proposition 3.

Proposition 4 *Problem* (1) *with matrix entries in* $\mathbb{Q}$ *and* $D_F = D_F^a$ *is polynomial time solvable.*

Proof of Proposition 4: Consider the following polynomial expression:

$$M_1^T(\bar{Q}_1 + Q_1) \dots M_{L-1}^T(\bar{Q}_{L-1} + Q_{L-1})M_L^T, \tag{2.15}$$

where we decomposed $D_i = \bar{Q}_i + Q_i$ in Definition 4, such that $\bar{Q}_i$ is constant, diagonal, with zero entries except for the 1 entries which are enforced by the network activation and sign pattern: strictly positive activation before application of ReLU when network is evaluated at x, or identity activations. Furthermore, Q_i contains $q_i \leq p_i$ diagonal variables to be chosen in $[0, 1]$ corresponding to the zero activation pattern before application of ReLU, for $i = 1, \dots, L-1$. The strictly negative values before application of ReLU do not play an additional role, they correspond diagonal entries constrained to 0 in both $\bar{Q}_i$ and Q_i, $i = 1, \dots, L-1$. Note that a polynomial is constant on a box if and only if it is constant so the polynomial expression in (2.15) is constant when diagonal entries are constrained in $[0, 1]$, if and only if it is constant. So the problem reduces to decide if the polynomial expression in (2.15) is non constant, with respect to variables $Q_1, \dots, Q_{L-1}$. We show that this reduces to a graph connectivity problem over $2 + \sum_{i=1}^{l-1} q_i$ vertices and edge weight given by partial products in (2.15).

First, the problem can be reduced to finding a non-zero value in the expression in (2.15). Indeed, one can substract the value obtained choosing $Q_i = 0$, $i =$

$1, \ldots, L-1$ and use the following block representation:

$$\begin{pmatrix} M_1^T & -M_1^T \end{pmatrix} \begin{pmatrix} \bar{Q}_1 + Q_1 & 0 \\ 0 & \bar{Q}_1 \end{pmatrix} \ldots \begin{pmatrix} M_{L-1}^T & 0 \\ 0 & M_{L-1}^T \end{pmatrix} \begin{pmatrix} \bar{Q}_{L-1} + Q_{L-1} & 0 \\ 0 & \bar{Q}_{L-1} \end{pmatrix} \begin{pmatrix} M_L^T \\ M_L^T \end{pmatrix}$$
$$= M_1^T(\bar{Q}_1 + Q_1) \ldots M_{L-1}^T(\bar{Q}_{L-1} + Q_{L-1}) M_L^T \quad - \quad M_1^T \bar{Q}_1 \ldots M_{L-1}^T \bar{Q}_{L-1} M_L^T. \tag{2.16}$$

Therefore, expression (2.15) is nonconstant if and only if expression in (2.16) takes a nonzero value for some assignment of $Q_1, \ldots, Q_{L-1}$. The number of variables in (2.15) and (2.16) is the same and they have exactly the same form. Therefore we assume without loss of generality that the problem is to decide if the polynomial expression in (2.15) is not equal to the null polynomial.

Expression (2.15) is a vector function each of its coordinates being a polynomial function. It is not uniformly null if and only if and only if there exists a coordinate which is not the null polynomial, so we may add a diagonal matrix Q_0 with $p_0 = p$ diagonal entries in $[0, 1]$ (and $\bar{Q}_0 = 0$ for the sake of symmetry) and $M_0 \in \mathbb{R}^{p \times 1}$ the vector of all ones and find a nonzero value for the product

$$M_0^T(\bar{Q}_0 + Q_0) M_1^T(\bar{Q}_1 + Q_1) \ldots M_{L-1}^T(\bar{Q}_{L-1} + Q_{L-1}) M_L^T, \tag{2.17}$$

Expression (2.17) is now real valued and therefore defines a polynomial. For each $0 = 1 \ldots L-1$, denote by $d_i \in [0, 1]^{q_i}$, the vector containing the diagonal entries of matrix Q_i, this corresponds exactly to the variable diagonal elements of D_i in Definition 4. Denote by $P(d_0, \ldots, d_L)$ the obtained polynomial, P is multilinear in $d_0, \ldots, d_{L-1}$, that is, it has an affine dependency for one block vector if the others are fixed. Therefore the hessian of P has zero diagonal blocks and the function is harmonic (hessian has zero trace), as a consequence, the maximum principle for harmonic functions entails that its maximum and minimum on any polytope are attained at vertices.

For $i = 0, \ldots, L-1$ denote by $\Delta_i \subset \mathbb{R}^{q_i}$, the convex hull of the origin and the canonical basis vectors, this is a q_i dimensional simplex with nonempty interior. The polynomial P in (2.17) is identically zero if and only if it vanishes on the product of simplices $\Delta_0 \times \ldots \times \Delta_{L-1}$ (which has non empty interior), if and only if it vanishes on the product set of the edges of these simplices by the maximum principle. In other words, P is not identically zero, if and only if it contains a nonzero element when each d_i is restricted to be an element of the canonical basis (zero vector with exactly one nonzero entry) or the null vector.

Define a graph with a layer structure:

- The source layer V_{-1} contains a single source node, $v_{-1,1}$.
- The zero-th layer V_0 contains $q_0 = p$ nodes $v_{0,1} \ldots v_{0,q_0}$.
- Recursively, the i-th layer V_i contains q_i nodes $v_{i,1} \ldots v_{i,q_i}$, for $i = 1 \ldots L-1$.
- The sink layer V_L contains a single node node $v_{L,1}$.

We connect nodes between consecutive layers, respecting the order induced by the layer structure. For $i = -1, \ldots L-1$ and $j = 0, \ldots, L$, with $j > i$, we connect layers V_i and V_j as follows

- Compute the quantity

$$R = \left(\prod_{m=i+1}^{j-1} M_m^T \bar{Q}_m \right) \times M_j^T,$$

  where if $j = i + 1$ the product reduces to the identity ($R = M_j^T$).

- For $k = 1, \ldots, q_i$ and $l = 1, \ldots, q_j$, add an edge with between $v_{i,k}$ and $v_{j,l}$ if $R_{k,l} \neq 0$.

The resulting graph has a number of nodes equal to the number of ReLU functions in F plus p additional nodes and the source and sink nodes. Computation of edges can be done in polynomial time: it requires at most $4(L+1)^2$ matrix products involving at most $2L+1$ matrices. Indeed the product of m matrices has polynomial time complexity in the representation bit size of the m input matrices.

In this graph, a directed path from the source to the sink visits each layer at most once, and in that case it visits a single node. Each such path corresponds to a monomial with nonzero coefficient appearing in the polynomial P in (2.17) by construction of the graph structure. Conversely each nonzero coefficient of a given monomial in (2.17) is uniquely associated to a path in the graph which corresponds to the nodes associated to variables in the monomial. Therefore, the source is connected to the sink if and only if there is a nonzero monomial in (2.17), if and only if the corresponding polynomial is nonzero. Furthermore, each path corresponds to the evaluation of the program at an edge of the product $\Delta_0 \times \ldots \times \Delta_{L-1}$. Therefore finding a path connecting the source to the sink allows to compute a nonzero element in the product and if no such path exists, the polynomial is identically zero.

So we have shown that the truth value of problem 1 with $D_F = D_F^q$, is the same as the source being connected to the sink by a directed path in the graph we defined, which has size polynomialy bounded compared to network size. Connectivity can be solved, for example using Dijkstra's algorithm, in time $O(|V|^2)$ where $|V|$ is the number of nodes (or vertices). A path represents a nonzero element of $D_f(0)$ and if no such path exists, we conclude that $D_F(0) = \{0\}$. This shows that the problem is solvable in polynomial time and concludes the proof.

□

2.9.5 Additional lemmas

The following lemma provides a characterization of singleton subgradient for linear ReLU networks.

Lemma 2 *Let F be a linear ReLU network, then $\partial^c F(0) = \{0\}$ if and only if F is constant.*

Proof : If F is constant, the result is immediate because $F \equiv 0$. Now, suppose that $\partial^c F(0) = \{0\}$. We know that F is piecewise linear and there exists a finite set of polyhedron whose union is $\mathbb{R}^p$, where F is affine linear over each polyhedron. Furthermore, F is positively homogeneous, therefore for each $x \in \mathbb{R}^p, \partial^c F(x) =$

$\partial^c F(\lambda x)$ with $\lambda > 0$. Setting $R \subset \mathbb{R}^p$, the full measure set where F is differentiable, one has that for all $x \in \mathbb{R}^p$ and

$$\partial^c F(x) = \operatorname{conv}\left\{v \in \mathbb{R}^p,\ \exists y_k \underset{k\to\infty}{\to} 0 \text{ with } y_k \in R,\ v_k = \nabla F(y_k) \underset{k\to\infty}{\to} v\right\} = \{0\}.$$

Therefore, each affine part has zero derivative on each polyhedra and by continuity we conclude that F is constant. □

The next lemma describes an explicit representation of maximum of finitely many numbers using a ReLU network with weights in $\{-1, 0, 1\}$.

Lemma 3 *Given $k \in \mathbb{N}$, $k > 0$, there exists F, a* ReLU *network with k* ReLU *layers of size at most $3 \times 2^{k-1}$ and weight matrices with entries in $\{-1, 0, 1\}$, with $p = 2^k$ inputs such that for any $x \in \mathbb{R}^p$,*

$$F(x) = \max_{i=1,\dots,2^k} x_i.$$

Proof : We proceed by recursion on k. Note that for any $x_1, x_2 \in \mathbb{R}$

$$\max\{x_1, x_2\} = \mathrm{ReLU}(x_1 - x_2) + x_2 = \mathrm{ReLU}(x_1 - x_2) + \mathrm{ReLU}(x_2) - \mathrm{ReLU}(-x_2).$$

Set the matrices

$$A = \begin{pmatrix} 1 & -1 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 1 & -1 \end{pmatrix}.$$

The function $F_1 \colon \mathbb{R}^2 \to \mathbb{R}$ given by

$$F_1(x) = B\mathrm{ReLU}(Ax)$$

satisfies $F_1(x) = \max\{x_1, x_2\}$. This proves the result for $k = 1$.

Now assume that for $k \geq 1$, we have a network with k ReLU layers of size at most 3×2^k represented by matrices $M_1, \dots, M_{k+1}$ with entries in $\{-1, 0, 1\}$, such that the corresponding ReLU network, as in (2.13) $F_k \colon \mathbb{R}^{2^k} \to \mathbb{R}$ satisfies for all $x \in \mathbb{R}^{2^k}$,

$$F_k(x) = \max_{i=1,\dots,2^k} x_i.$$

Set $\tilde{F}_k$ the concatenation of two copies of F_k, that is $\tilde{F}_k \colon \mathbb{R}^{2^{k+1}} \to \mathbb{R}^2$, such that for all $x, y \in \mathbb{R}^{2k}$,

$$\tilde{F}_k(x, y) = \begin{pmatrix} \max_{i=1,\dots,2^k} x_i \\ \max_{i=1,\dots,2^k} y_i \end{pmatrix}.$$

The matrices representing $\tilde{F}_k$ can be described in block form

$$\tilde{M}_i = \begin{pmatrix} M_i & 0 \\ 0 & M_i \end{pmatrix} \in \mathbb{R}^{(2p_i)\times(2p_{i-1})}$$

for $i = 1, \ldots, k+1$, where $p_0 = 2^k$ and $p_k = 1$. This network is made of k layers of size at most $3 \times 2^{k+1}$, it has 2^{k+1} inputs and two outputs and its weight matrices have elements in $\{-1, 0, 1\}$. The block representation of the last matrix of this network is of the form

$$\begin{pmatrix} M_{k+1} & 0 \\ 0 & M_{k+1} \end{pmatrix} \in \mathbb{R}^{2 \times l}$$

where l is the size of the row vector M_{k+1}. We have

$$\begin{aligned} & A \times \tilde{M_{k+1}} \\ = \quad & \begin{pmatrix} 1 & -1 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \times \begin{pmatrix} M_{k+1} & 0 \\ 0 & M_{k+1} \end{pmatrix} = \begin{pmatrix} M_{k+1} & -M_{k+1} \\ 0 & M_{k+1} \\ 0 & -M_{k+1} \end{pmatrix} \in \mathbb{R}^{3 \times (2l)}. \end{aligned}$$

We set $F_{k+1}(x, y) = F_1(F_k(x), F_k(y)) = F_1(\tilde{F}_k(x, y))$ for all $x, y \in \mathbb{R}^{2k}$. In matrix notation we have

$$F_{k+1}(x, y) = B\mathrm{ReLU}(A\tilde{F}_k(x, y)).$$

The involved matrices are $M_{k+2} = B$, $A \times \tilde{M}_{k+1}$ and $\tilde{M}_k \ldots \tilde{M}_1$. They all have entries in $\{-1, 0, 1\}$ and the corresponding network has layers of size at most $3 \times 2^{k+1}$. The result then holds by recursion. □

Chapter 3

On the numerical reliability of nonsmooth automatic differentiation

Abstract

This section investigates the reliability of automatic differentiation (AD) for neural networks using nonsmooth operations (e.g., MaxPool, ReLU) across varying precision levels (16, 32, 64 bits), architectures (LeNet, VGG, ResNet), and datasets (MNIST, CIFAR10, SVHN, ImageNet). While AD computes derivatives that are almost everywhere correct, even for nonsmooth operations, it relies on floating-point arithmetic, which introduces the potential for numerical inaccuracies.
Bertoin et al. [41] analyzed the impact of $\mathrm{ReLU}'(0)$ on AD output, identifying a numerical bifurcation zone where $\mathrm{ReLU}'(0) = 0$ differs from $\mathrm{ReLU}'(0) = 1$. We generalize this concept to a broader range of nonsmooth operations by introducing:

(1) Bifurcation zone: AD is incorrect over real numbers.

(2) Compensation zone: AD is incorrect in floating-point arithmetic but correct over reals.

Our experiments with SGD show that lower-norm MaxPool Jacobians preserve training stability and performance, while higher-norm Jacobians lead to instability and reduced accuracy. Techniques such as batch normalization, Adam-like optimizers, or higher precision effectively mitigate the adverse effects of high-norm MaxPool Jacobians.

This Part is organized as follows:

- In Section 3.1, we present the background, our motivation, and related work.
- In Section 3.2, we discuss the elements of nonsmooth backpropagation and define three subsets of network parameters - the bifurcation zone, compensation zone, and regular zone. We also examine the implications of nonsmooth

MaxPool Jacobians for backpropagation, based on Bolte and Pauwels [10], [11].

- In Section 3.3, we focus on the numerical bifurcation and compensation zones, and the factors that affect their importance.
- In Section 3.4, we present detailed experiments on neural network training.

3.1 Introduction

As seen in Chapter 1 and Chapter 2, nonsmooth neural networks are trained using optimization algorithms [27], [124] based on autodiff [2]–[4]. AD is a crucial tool in contemporary learning architectures as it allows for fast differentiation [8], [55]. It is implemented in popular machine learning libraries such as TensorFlow [5], PyTorch [6], and Jax [7]. Although the validity domain of AD is theoretically limited to smooth functions [8], it is commonly used for nonsmooth functions [41], [55], [84]. In this part, we examine the reliability of autodiff for neural networks with nonsmooth operations (MaxPool, ReLU) operating with floating point numbers.

MaxPool: a nonsmooth operation The MaxPool operation, introduced by Yamaguchi et al. [125], is commonly used in convolutional neural networks (CNN) for image classification [20], [23], [50], [80]. MaxPool reduces the spatial dimensions of a feature map by selecting the maximum value within specific patches. However, when applied to uniform pixel values, MaxPool can cause nonsmoothness, especially at image edges where identical pixels can be chosen arbitrarily (see Figure 3.1 for an illustration). In such cases, different nonsmooth MaxPool Jacobians bear a variational sense as it corresponds to a subgradient [108].

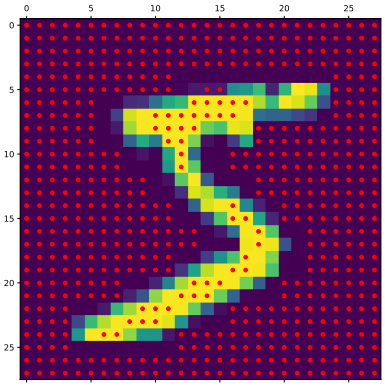


Figure 3.1: Image segment post-convolution, spotlighting equal pixel values (marked in red) within a 2x2 MaxPool window.

3.1.1 Various types of nonsmooth AD errors

We carry out a PyTorch [6] experiment to investigate the autodiff behavior of the nonsmooth max function, defined as $\max\colon x \mapsto \max_{1\leq i\leq 4} x_i \in \mathbb{R}$. We implement two max programs ($\max_1$ and $\max_2$) with different derivative implementations as in Figure 3.2. For example, the max function is not differentiable at $x = (1,1,1,1)$ and autodiff returns $(1,0,0,0)$ for $\max_1$ and $(0.25, 0.25, 0.25, 0.25)$ for $\max_2$. Let zero be a program as follows: $\text{zero}\colon t \mapsto \max_1(t\times x) - \max_2(t\times x)$. The AD output of zero is denoted by zero'.

As mathematical functions, both $\max_1$ and $\max_2$ output the same value and zero always outputs 0. However, we observe an unexpected behavior when using AD and floating-point numbers: $\text{zero}'(t) \neq 0$ for some $t \in \mathbb{R}$.

```
def max1(x):
    res = x[0]
    for i in range(1, 4):
        if x[i] > res:
            res = x[i]
    return res
```

```
def max2(x):
    return torch.max(x)

def zero(t):
    z = t * x
    return max1(z) - max2(z)
```

Figure 3.2: Implementation of programs $\max_1$, $\max_2$ and zero using PyTorch. Programs $\max_1$ and $\max_2$ are an equivalent implementation of max, but with different derivatives due to the implementation.

	$\text{zero}'(t)$						
t	-10^{-3}	-10^{-2}	-10^{-1}	0	10^{1}	10^{2}	10^{3}
$x_1 = (1, 2, 3, 4)$	0.0	0.0	0.0	-1.5	0.0	0.0	0.0
$x_2 = (1.4, 1.4, 1.4, 1.4)$	10^{-7}	10^{-7}	10^{-7}	10^{-7}	10^{-7}	10^{-7}	10^{-7}

Table 3.1: Overview of numerical AD errors for the zero program with 32 bits precision.

For x_1 in Table 3.1, we observe a significant error at $t = 0$ where the computed derivative, $\text{zero}'(0)$, is -1.5, deviating from the correct derivative value. In contrast, for x_2, which often appears in tasks such as image classification (refer to Figure 3.1), theoretical calculations predict $\text{zero}'(t) = 0$ for any $t \in \mathbb{R}$. However, discrepancies emerge when using floating-point arithmetic, as illustrated by AD results. Specifically, across all t values listed in Table 3.1, $\text{zero}'(t)$ approximates to 5.96×10^{-8} (rounded to 10^{-7} in the table), which is near the computational precision limit of 32-bit systems. This phenomenon occurs due to numerical arithmetic limits. Typically, t denotes a neural network parameter and x represents an input image pixel area where the pixel values are identical.

The behavior observed in Table 3.1 are not due to the nonsmooth multivariate nature of the max function. Similar phenomena can also be seen when the univariate ReLU operation computes max. In contrast, we observe no errors near machine precision using NormPool—a nonsmooth multivariate function that computes the Euclidean norm. Furthermore, reproducing Table 3.1 with $\text{zero}\colon t \mapsto \text{ReLU}_1(t) - \text{ReLU}_2(t)$ where $\text{ReLU}_1'(0) = 0$ and $\text{ReLU}_2'(0) = 1$, also shows no AD errors near machine precision. These observations suggest that minor AD errors may stem from the intrinsic properties of the max function. Thus, our study primarily examines max and MaxPool operations. For more details, please refer to Appendix 3.5.1 and Appendix 3.5.2.

3.1.2 Reals vs floating-point numbers

Over the real numbers, AD computes derivatives for nondifferentiable functions, except on a Lebesgue measure-zero subset of inputs [10], [11]. However, as in-

dicated in Table 3.1, the use of floating-point arithmetic can expand the subsets where AD yields incorrect results. In Section 3.3, we propose two subsets of network parameters where AD numerically fails: a new *bifurcation zone*, characterized by significant AD amplitude variations, and a *compensation zone*, where minor amplitude variations occur near machine precision due to rounding schemes in inexact arithmetic over reals (e.g., non-associativity). Our experiments show that in a 64-bit network using MaxPool, the compensation zone covers the entire parameter space. In a 32-bit network, both compensation and bifurcation zones exist, while in a 16-bit setting, the bifurcation zone dominates the entire parameter space.

3.1.3 Related works and contributions

Recent work shows that for a broad class of programs using nonsmooth functions, AD is mostly incorrect on a Lebesgue measure-zero subset of the input domain [10], [126]. However, practical inputs are machine-representable. In this context, [126] examined AD correctness in neural networks with machine-representable parameters, excluding MaxPool. Additionally, Bertoin et al. [41] studied how $\mathrm{ReLU}'(0)$ affects AD and training, identifying a *bifurcation zone* in ReLU networks where AD is incorrect. These studies don't address cases where AD is incorrect for floating-point numbers but correct over reals, as seen in Table 3.1. To fill this gap, we propose a *new bifurcation zone* and introduce the concept of a *compensation zone*. Our research evaluates AD reliability in MaxPool neural networks at different precision levels, examining how the compensation zone's impact varies with network architecture. We also explore how nonsmooth MaxPool Jacobians affect training stability and performance.

3.2 Nonsmooth AD for MaxPool neural networks

3.2.1 Preliminaries and notations

As discussed in Chapter 1, supervised training uses a dataset $(x_i, y_i)_{i=1}^N$, where each x_i is an input and y_i its corresponding label. A neural network, represented by the function f with parameters θ, generates predictions $\hat{y}_i = f(x_i, \theta)$. The difference between these predictions and the true labels is measured by a loss function ℓ. The goal is to minimize this discrepancy over the training set by minimizing an empirical loss function L, such as:

$$\min_{\theta \in \mathbb{R}^p} \quad L(\theta) := \frac{1}{N} \sum_{i=1}^{N} \ell(\hat{y}_i, y_i). \tag{3.1}$$

For all $i \in \{1, \ldots, N\}$ and $\theta \in \mathbb{R}^p$, Equation (3.1) can be expressed with $\ell(\hat{y}_i, y_i) = l_i(\theta)$, where $l_i : \mathbb{R}^p \to \mathbb{R}$ represents a composition of H elementary functions as follows:

$$l_i(\theta) = g_{i,H} \circ g_{i,H-1} \circ \ldots \circ g_{i,1}(\theta). \tag{3.2}$$

Equation (3.2) models common neural network types, including feed-forward [127], convolutional [49], and recurrent networks [128]. For a more concrete example, please refer to Appendix A.2 in [41]. We focus on elementary functions that are locally Lipschitz and semialgebraic, commonly found in nonsmooth neural

networks [10], [11]. Functions $g_{i,j}$ include operations such as linear transformations, ReLU, MaxPool, convolution with filters, and softmax for classification.

3.2.2 Nonsmooth AD framework

Training nonsmooth neural networks [10], [37], [55], [84], [129] is challenging due to the need to compute subgradients from Equation (3.1). Major machine learning tools such as TensorFlow [5], PyTorch [6], and Jax [7] address this issue using automatic differentiation, referred to here as backprop [2], [3]. They apply differential calculus to nonsmooth items, often replacing derivatives with *Clarke Jacobians* [107] (see Section 2.1).

Definition 5 (Calculus model, programs and nonsmooth AD) Let l be a composition function evaluated at $\theta \in \mathbb{R}^p$, as specified in Equation (3.2). A program P that executes l can be described through a sequence of subprograms such as:

- Elementary programs: $\{g_j\}_{j=1}^H$ such that $l(\theta) = g_H \circ g_{H-1} \circ \ldots \circ g_1(\theta)$.
- Derived programs: $\{v_j\}_{j=1}^H$ where each $v_j(w) \in \mathrm{Jac}^{\mathrm{c}}\, g_j(w)$ at point $w = g_{j-1} \circ \cdots \circ g_1(\theta)$.

Then, the backprop algorithm automates applying differential calculus rules as follows:

$$\mathrm{backprop}[P](\theta) = v_H\left(g_{H-1} \circ \ldots \circ g_1(\theta)\right) \cdot v_{H-1}\left(g_{H-2} \circ \ldots \circ g_1(\theta)\right) \cdot \ldots \cdot v_1(\theta). \tag{3.3}$$

In practice, AD libraries [5]–[7] implement dictionaries (see for e.g. [8], [55] and Section 2.4) containing conjointly elementary programs and derived programs which efficiently computes the quantities defined in Equation (3.3).

Remark 7 As seen in Section 3.1.1 with the zero program, various programs can implement a unique composition function l. Each nonsmooth elementary program g_j in the composition (see Definition 5) can be associated with different derived programs v_j. Specifically, for any $j = 1, \ldots, H$ and $w = g_{j-1} \circ \cdots \circ g_1(\theta)$, all selections $v_j(w)$ from the Clarke Jacobian of $g_j(w)$ bear a variational sense.

Example 5 The Clarke subdifferential of $\mathrm{ReLU}(t) = \max(0, t)$ at t is 0 for $t < 0$, 1 for $t > 0$, and the interval $[0, 1]$ for $t = 0$. All ReLU-derived program that implements $\mathrm{ReLU}'(0) = s$ with $s \in [0, 1]$ can be used for backprop.

Definition 6 (Backprop set) Let l denote a composition function evaluated at $\theta \in \mathbb{R}^p$, as specified in Equation (3.2). We define $J(\theta)$ as the function that encompasses the set of all possible backprop outputs through all programs implementing $l(\theta)$ as in Definition 5:

$$J(\theta) = \left\{\mathrm{backprop}[P](\theta) : P \text{ is a program implementing } l(\theta)\right\}. \tag{3.4}$$

Remark 8 For a composition function l composed by C^1 elementary programs $\{g_j\}_{j=1}^H$, $J(\theta)$ is a singelton for all $\theta \in \mathbb{R}^p$. For locally Lipchitz semialgebraic (or definable) elementary programs $\{g_j\}_{j=1}^H$: Equation (3.3) returns an element within the backprop set.

3.2.3 Network parameters subsets

Bertoin et al. [41] studied the bifurcation zone in ReLU networks, where AD output diverges between $\text{ReLU}'(0) = 0$ and $\text{ReLU}'(0) = 1$. However, they did not explore cases where backprop should compute a singleton, but AD produces errors due to floating-point arithmetic. We observe this issue when comparing AD outputs from different MaxPool derivative implementations. To address this, we introduce the concept of the compensation zone.

Definition 7 (Compensation, bifurcation and regular zones) For each $i = 1, \ldots, N$, let l_i denote a composition function evaluated at $\theta \in \mathbb{R}^p$ and $J_i(\theta)$ denote the backprop set associated as detailed in Definition 6. We define the following network parameters subsets of $\mathbb{R}^p$:

$$\Theta_R = \left\{\theta \in \mathbb{R}^P : \forall i, j \in \{1, \ldots, N\} \times \{1, \ldots, H\}, \text{Jac}^{\text{c}}\, g_{i,j}(w) \text{ a singleton}\right\} \quad (3.5)$$

$$\Theta_C = \left\{\theta \in \mathbb{R}^P \backslash \Theta_R : \forall i \in \{1, \ldots, N\}, J_i(\theta) \text{ is a singleton}\right\} \quad (3.6)$$

$$\Theta_B = \{\theta \in \mathbb{R}^p \backslash \Theta_R : \exists i \in \{1, \ldots, N\} \text{ such that } J_i(\theta) \text{ is not a singleton}\} \quad (3.7)$$

where $w = g_{i,j-1} \circ \ldots \circ g_{i,1}(\theta)$, Θ_R is the regular zone, Θ_C the compensation zone and Θ_B the bifurcation zone.

The mathematical tools of Proposition 5 are conservative fields developed in [10]. This proposition implies that theoretically (assuming exact arithmetic over the reals), the backprop set is almost everywhere a singleton. The proof is given in Appendix 3.6.

Proposition 5 Given subsets Θ_R, Θ_B, and Θ_C in $\mathbb{R}^p$ as defined in Definition 7, the following properties hold:

- Θ_R, Θ_B, and Θ_C form a partition of $\mathbb{R}^p$.
- Θ_B is a Lebesgue null measure subset.

Remark 9 (Backprop returns a gradient a.e.) Let $\theta \in \mathbb{R}^p$ and P be a program implementing a composition function $l(\theta)$ as in Definition 5. Then $\text{backprop}[P](\theta) = \nabla l(\theta)$ almost everywhere.

3.2.4 MaxPool-derived programs

Definition 8 (Clarke Jacobian of matrix's maximum function) Let X be a $m \times n$ real matrix and F_s be a function such that $F_s(X) = \max_{1 \le i \le m, 1 \le j \le n} X_{ij} \in \mathbb{R}$, where $s := m \times n$ denotes the size of X. The Clarke Jacobian of F_s at the point X is:

$$\text{Jac}^{\text{c}}\, F_s(X) = \text{conv}\left(\bigcup_{(i,j) \in A(X)} E_{ij}\right), \quad (3.8)$$

where $A(X) := \{(i, j) \in \{1, \ldots, m\} \times \{1, \ldots, n\} : F_s(X) = X_{ij}\}$ is the active set and E_{ij} is an $m \times n$ matrix with all entries equal to 0 except for the (i, j)-th entry which is 1.

Definition 9 (MaxPool operation) Let $X \in \mathbb{R}^{p\times q}$ be a real matrix, and $s := m \times n$ be the size of a pooling window such that $p \geq m$ and $q \geq n$. For each $i \in \{0, \ldots, \left\lfloor \frac{p}{m} \right\rfloor - 1\}$ and $j \in \{0, \ldots, \left\lfloor \frac{q}{n} \right\rfloor - 1\}$, we define a submatrix $X_{i,j}$ of X, of size $m \times n$ as follows:

$$X_{i,j} := \{X_{kl} : m \times i \leq k < m \times (i+1), n \times j \leq l < n \times (j+1)\}, \tag{3.9}$$

where k and l are the indices of the entries in X, in the lexicographic order. The MaxPool operation output a matrix $Y \in \mathbb{R}^{\left\lfloor \frac{p}{m} \right\rfloor \times \left\lfloor \frac{q}{n} \right\rfloor}$ where $Y_{ij} = F_s(X_{i,j})$ for all $i \in \{0, \ldots, \left\lfloor \frac{p}{m} \right\rfloor - 1\}$ and $j \in \{0, \ldots, \left\lfloor \frac{q}{n} \right\rfloor - 1\}$. Finally, the MaxPool Clarke Jacobian at point X, denoted as $\mathrm{Jac}^{\mathrm{c}}\,\mathrm{MaxPool}(X)$, can be obtained by replacing each submatrix $X_{i,j}$ in X with $\mathrm{Jac}^{\mathrm{c}}\,F_s(X_{i,j})$.

Definition 10 (MaxPool-derived programs) Define $X_{i,j} \in \mathbb{R}^{m\times n}$ as a submatrix of X (Definition 9), from which we derive MaxPool programs based on the Clarke Jacobian:

- **Native:** Chooses the first index (i_1, j_1) from the active set $A(X_{i,j})$ and outputs $E_{i_1 j_1}$. Autograd libraries use this implementation.
- **Minimal:** Takes all indices from $A(X_{i,j})$, averaging them as $\frac{1}{|A(X_{i,j})|} \sum_{(k,l)\in A(X_{i,j})} E_{kl}$. We called it "minimal" as it yields the smallest norm element within Equation (3.8).
- **Hybrid:** A blend of native and minimal, parameterized by $\beta > 0$:

$$(1-\beta) \cdot E_{i_1 j_1} + \beta \cdot \left(\frac{1}{|A(X_{i,j})|} \sum_{(k,l)\in A(X_{i,j})} E_{kl} \right),$$

Remark 10 The hybrid MaxPool-derived program is a selection of the MaxPool Clarke Jacobian for $\beta \in [0, 1]$ and a selection of a conservative Jacobian approach for other β values, as outlined in [10].

In the following section, we will examine the impact of using a conservative Jacobian ($\beta > 1$) on learning and training. This is important because conservative Jacobians have a variational interpretation in nonsmooth AD [10].

3.3 A new numerical bifurcation zone

In this section, we examine numerical subsets of network parameters for neural networks with MaxPool operations across different floating-point precisions. We find that the numerical bifurcation zone identified by Bertoin et al. [41] does not apply to MaxPool-based programs. Specifically, both the max function and MaxPool cause minor AD errors in floating-point computations, but not in real numbers. To address this, we propose a new numerical bifurcation and define a compensation zone using two methods: one with nondeterministic GPU computations and another with ReLU-derived programs, based on the framework in Section 3.2.

3.3.1 A numerical criteria for the bifurcation and compensation zone

Numerical bifurcation zone for ReLU **networks:** Recently, Bertoin et al. [41] investigated a numerical bifurcation zone S_{01} specific to ReLU-derived programs. For each $i = 1, \ldots, N$, two programs implement a same function l_i: R_i^0 (using $\text{ReLU}'(0) = 0$) and R_i^1 (using $\text{ReLU}'(0) = 1$). The bifurcation zone S_{01} is defined as:

$$S_{01} = \left\{\theta \in \mathbb{R}^P : \ \exists i \in \{1, \ldots, N\}, \text{backprop}[R_i^0](\theta) \neq \text{backprop}[R_i^1](\theta)\right\}. \quad (3.10)$$

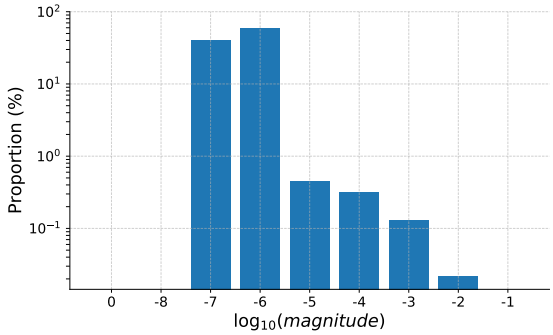
Definition 11 (Backprop variation) Let $(B_q)_{q\in\mathbb{N}}$ be a sequence of mini-batches, where each batch size $|B_q|$ falls within $\{1, \ldots, N\}$. Consider $P = \{P_i\}_{i=1}^N$ and $Q = \{Q_i\}_{i=1}^N$ as two neural network implementations using different nonsmooth-derived programs (e.g., ReLU or MaxPool). Each P_i and Q_i computes a composition function l_i. The backprop variation between P and Q over M experiments with random parameters $\{\theta_m\}_{m=1}^M$ is defined as:

$$D_{m,q}(P, Q) = \left\| \text{backprop}\left[\sum_{i\in B_q} P_i(\theta_m)\right] - \text{backprop}\left[\sum_{i\in B_q} Q_i(\theta_m)\right] \right\|_1. \quad (3.11)$$

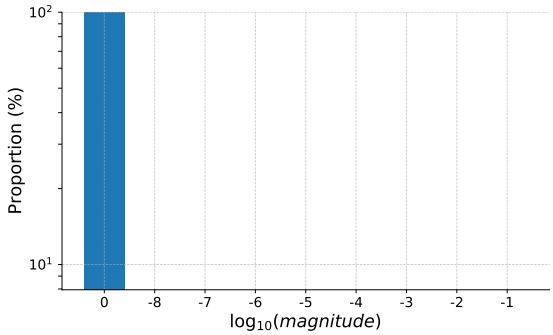
A 32 bits MNIST experiment: Let P and Q be programs for a LeNet-5 network (see Figure 1.3) on MNIST, using native and minimal MaxPool programs, respectively. For a sanity check, let $\tilde{P}$ be a copy of P. We compute the backprop variation (as in Definition 11) between P and $\tilde{P}$ and between P and Q. We control all sources of divergence in our implementation using deterministic computation. Results are reported in Figure 3.3, and the experiment was run on a CPU under 32 bits precision. See also Section 3.5.3 for a practical example.

We observe no variation in backpropagation between P and $\tilde{P}$, indicating controlled sources of divergence. This observation contrasts with the expectations from Proposition 5, which predicts no variation between P and Q; we find $D_{m,q}(P, Q) > 0$ across all m, q. We identify two types of variations: minor ones, comprising 98.78% of parameters, which align with machine precision in 32 bits (between 10^{-8} and 10^{-7}), and major ones, peaking at 10^{-3} and accounting for 1.22% of parameters. Consequently, this section focuses on analyzing backprop variations in MaxPool-derived programs to propose a new bifurcation zone.

An heuristic for the numerical bifurcation zone: In Figure 3.3, we identify two types of backpropagation variations: one potentially arising from numerical bifurcations and another due to floating-point arithmetic errors, which we refer to as compensation errors. To establish criteria for proposing a new numerical bifurcation zone for nonsmooth-derived programs, we compare these observed backpropagation variations with known sources, such as GPU nondeterminism and variations from ReLU-derived programs at 16 and 32-bit precision. This method allows us to differentiate between numerical bifurcations and compensation errors without presuming distinct zones. We denote floating-point precision by ω and consider various neural networks like LeNet-5, VGG, or ResNet for our analysis.



(a) $D_{m,q}(P, Q)$



(b) $D_{m,q}(P, \tilde{P})$

Figure 3.3: Histogram of backprop variation $D_{m,q}$ for LeNet-5 on MNIST (128 mini-batch size) at 32-bit precision, comparing P with $\tilde{P}$ and P with Q over $M = 1000$ experiments.

A threshold with nondeterministic GPU computations: We set a threshold $\tau^1_{f,\omega}$ for the maximum backprop variation due to nondeterministic GPU computations (refer to Appendix 3.5.4 for more details):

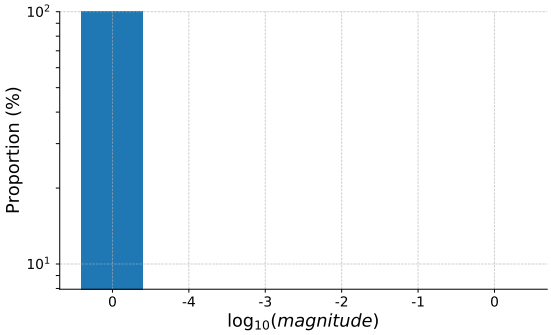
$$\tau^1_{f,\omega} = \max_{1 \le m \le M, 1 \le q} D_{m,q}(P, \tilde{P}) \tag{3.12}$$

where P and $\tilde{P}$ compute a neural network f using the same nonsmooth-derived program, for example, $\mathrm{ReLU}'(0) = 0$ or minimal MaxPool. See Figure 3.4 for an illustration. We observe no variation at $\omega = 16$, with PyTorch's nondeterministic GPU operations disabled. Additionally, τ^1 can be interpreted as an upper bound on the expected backprop variation when repeatedly running the same program under nondeterministic conditions.

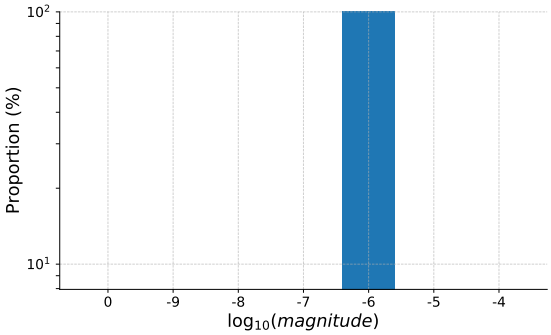
A threshold with ReLU**-derived programs:** For ReLU-derived programs, we define R^0 (with $\mathrm{ReLU}'(0) = 0$) and R^1 (with $\mathrm{ReLU}'(0) = 1$) as two programs implementing a same network f under deterministic GPU operations. We introduce threshold $\tau^2_{f,\omega}$ for backprop variation:

$$\tau^2_{f,\omega} = \min_{1 \le m \le M, 1 \le q} \left\{ D_{m,q}(R^0, R^1) : D_{m,q}(R^0, R^1) > 0 \right\}, \tag{3.13}$$

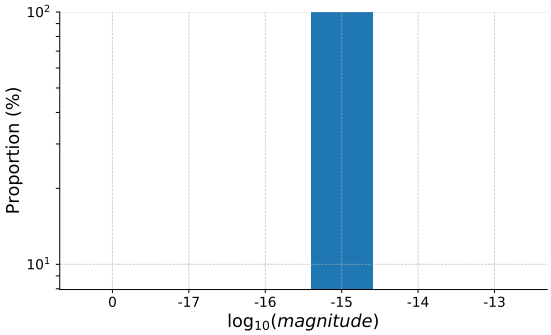
τ^2 can be interpreted as a lower bound on the error we expect to make when running two different ReLU-derived programs of the same function.



(a) $\tau^1_{f,16} = 0$



(b) $\tau^1_{f,32} = 1.11 \times 10^{-7}$

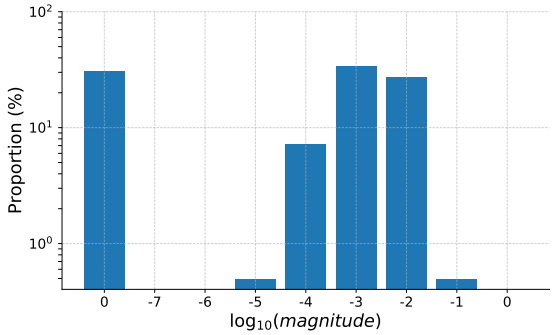


(c) $\tau^1_{f,64} = 1.55 \times 10^{-16}$

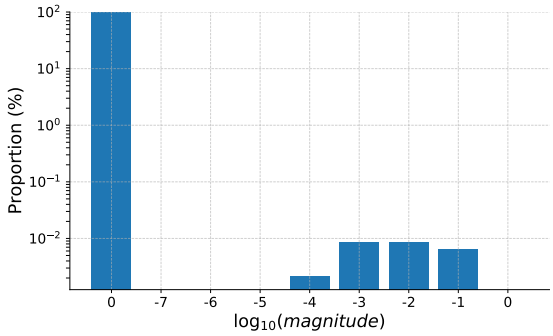
Figure 3.4: Histogram of backprop variation under nondeterministic GPU operations, where f is a LeNet-5 network on MNIST with batch size 128 for $M = 1000$ experiments.

Remark 11 We do not consider τ^2 in the context of MaxPool-derived programs, as it is anticipated that τ_2 will approximate machine precision values as in Figure 3.3.

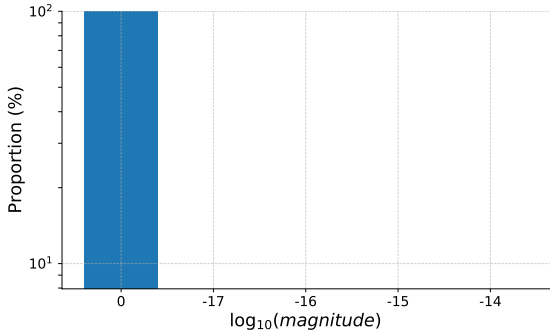
Figure 3.5 illustrates two types of backpropagation variations with ReLU-derived programs: significant divergences or none at all, contrasting with phenomena observed with MaxPool-derived programs. These divergences may suggest the presence of a numerical bifurcation zone. Additionally, variations from nondeterministic GPU computations, as shown in Figure 3.4, correspond to minor variations near machine precision, similar to those seen in Figure 3.3. We propose establishing a numerical bifurcation zone, applying different thresholds for various precisions to accommodate hardware constraints.



(a) $\tau^2_{f,16} = 3.39 \times 10^{-6}$



(b) $\tau^2_{f,32} = 4.81 \times 10^{-5}$



(c) $\tau^2_{f,64} = 0$

Figure 3.5: Histogram of backprop variation with ReLU-derived programs, where f is a LeNet-5 network on MNIST with batch size 128 for $M = 1000$ experiments.

Criteria 1 (Numerical bifurcation zone) For a neural network f and a floating-point precision ω, let $\tau_{f,\omega}$ be a fixed threshold (for e.g $\tau^1_{f,\omega}$, $\tau^2_{f,\omega}$). The numerical bifurcation zone is defined as:

$$S(\tau_{f,\omega}) = \left\{\theta \in \mathbb{R}^P : \exists i \in \{1, \ldots, N\}, \|\text{backprop}[P_i](\theta) - \text{backprop}[Q_i](\theta)\|_1 > \tau_{f,\omega}\right\}. \tag{3.14}$$

Here, P_i and Q_i are programs implementing f using different nonsmooth-derived programs.

Table 3.4 in Appendix 3.5.4 lists threshold values for various networks and datasets across 16-bit, 32-bit, and 64-bit precisions. These thresholds are numerical guides and fluctuate based on the initial network parameters, datasets, and architecture. The characteristics of the compensation zone depend on the

neural network's structure rather than the nature (univariate or multivariate) of the nonsmooth elementary programs defined in Definition 5. For example, computing MaxPool using ReLU programs can lead to compensation errors, as detailed in Table 3.1 (see Appendix 3.5.1). In convolutional networks like VGG or ResNet, computing MaxPool with ReLU functions using the formula $2\max(x, y) = (x + y) + (\text{ReLU}(x) - \text{ReLU}(-y)) + (\text{ReLU}(y) - \text{ReLU}(-x)$ does not align with the bifurcation zone proposed by [41]. Conversely, using NormPool—a nonsmooth multivariate function calculating the Euclidean norm—avoids such compensation errors. Further details can be found in Appendix 3.5.2.

3.3.2 Volume of the numerical bifurcation zone

We employed Monte Carlo sampling to estimate the volume of the numerical bifurcation zone for various networks, adhering to Criteria 1. Thresholds $\tau^2_{f,16}$, $\tau^1_{f,32}$, and $\tau^1_{f,64}$ were consistently applied across all networks, as detailed in Appendix 3.5.4 with reference to Equations (3.12) and (3.13).

Experimental Setup: We generated a set of network parameters $\{\theta_m\}_{m=1}^M$ randomly using Kaiming-Uniform initialization [130], with $M = 1000$ experiments conducted. Subsequently, we iterated over the entire CIFAR10 dataset to estimate the proportion of θ_m within the numerical bifurcation zone S as defined in Criteria 1 (referenced in Equation (3.17)) and the proportion of affected mini-batches (detailed in Equation (3.18)).

Impact of floating-point precision: Using the VGG11 model on the CIFAR10 dataset, we evaluated the volume of S across different precision levels. The results revealed that at 16-bit and 32-bit precision, all parameters resided within S, whereas at 64-bit precision, none did. This variance demonstrates the significant role of precision in the effects of backprop with MaxPool-derived programs. Notably, the impact on mini-batches was substantial, with 46% at 32 bits and 100% at 16 bits, underscoring the influence of precision on the computational outcomes.

Table 3.2: Impact of S according to floating-point precision using a VGG11, on CIFAR10 dataset and $M = 1000$ experiments. The first line represents network parameters θ_m in S, while the second measured the proportion of affected mini-batches falling in S.

Floating-point precision	16 bits	32 bits	64 bits
Proportion of $\{\theta_m\}_{m=1}^M$ in S	100%	100%	0%
Proportion of impacted mini-batches	100%	46.67%	0%

We also investigated the influence of mini-batch size on the proportion of affected mini-batches within the numerical bifurcation zone S using the VGG11 model on the CIFAR10 dataset. Our findings indicate that larger mini-batch sizes correlate with an increased proportion of impacted mini-batches at 32-bit precision. However, at 64-bit precision, no parameters were observed to fall into S

(as shown in Figure 3.6). Additionally, variations in network depth—examined across VGG variants 11, 13, 16, and 19—did not significantly alter the impact on mini-batches at 16-bit and 32-bit precisions. Notably, the introduction of batch normalization markedly increased the number of affected mini-batches at 32-bit precision.

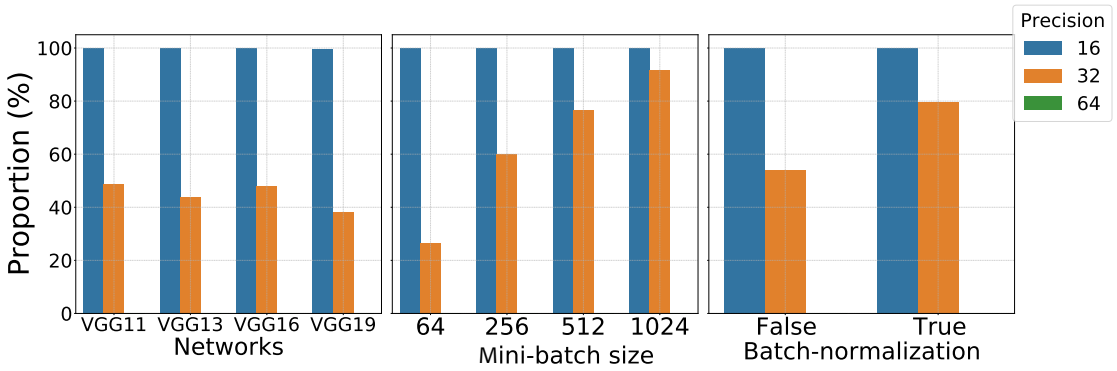


Figure 3.6: Impact of different size parameters on the proportion of affected mini-batches (see Equation (3.18) using CIFAR10 dataset. First: Different VGG network sizes. Second: VGG11 with varying mini-batch sizes. Third: VGG11 with and without batch normalization.

3.4 Experiments on learning

3.4.1 Benchmarks and implementation

Datasets and architectures: We train neural networks to investigate the impact of numerical effects outlined in Section 3.3. Our experiments used CIFAR10 [23], MNIST [49] and ImageNet [22] datasets. We test various network architectures including VGG11 [131], ResNet [45], and LeNet [49]. Details are available in Appendix 3.7.1.

Training settings: The default optimizer is SGD. Conducted on PyTorch and Nvidia V100 GPUs, we define mini-batch sequences $(B_q)_{q\in\mathbb{N}}$ with sizes $|B_q| \subset \{1,\dots,N\}$, where $\alpha_q > 0$ is the learning rate for each mini-batch q. Each program P_i in $P = \{P_i\}_{i=1}^N$ implements a function l_i (as in Definition 5). The SGD algorithm updates network parameters $\theta_{q,P}$ by:

$$\theta_{q+1,P} = \theta_{q,P} - \gamma \frac{\alpha_q}{|B_q|} \sum_{i\in B_q} \text{backprop}[P_i](\theta_{q,P}) \tag{3.15}$$

with $\gamma > 0$ indicating the step-size parameter.

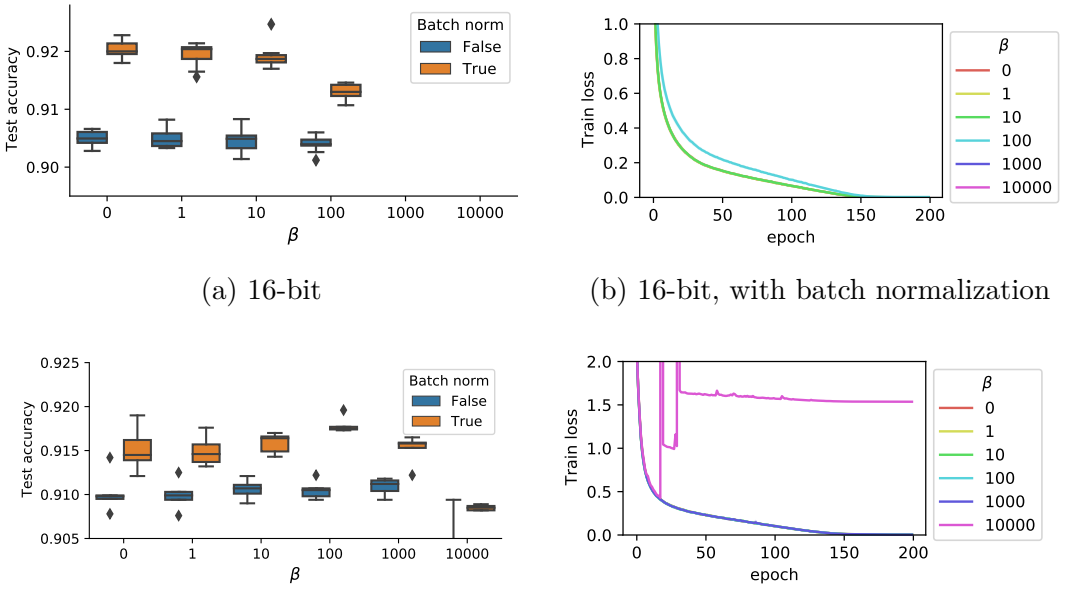
3.4.2 Effect on training and test errors

We further investigate the phenomenon from Section 3.3 using the CIFAR10 dataset [23] and the VGG11 architecture [131], at 16-bit and 32-bit precision with various β values (as defined in Definition 10). Each configuration was repeated

ten times with random initialization, and the results are shown in Figure 3.7. To validate our findings, we also tested with MNIST [49] and ImageNet [20] using ResNet18 and ResNet50 architectures [45]. Additional details on the experiments are in Appendix 3.7.

Training effect with 16-bit: For β values greater than 10^3, we observe training instability characterized by exploding gradients, unaffected by the presence of batch normalization. Conversely, stable and efficient test accuracies are maintained for β values within the set $\{0, 1, 10, 100\}$.

Training effect with 32-bit: When β values are large, such as 10^4, the training process may become unstable, exhibiting oscillations and sudden fluctuations in the learning curve. This instability can occur if batch normalization is not used. However, incorporating batch normalization with high β values helps stabilize the training dynamics, enhances test data accuracy, and prevents gradient explosion.



(a) 16-bit

(b) 16-bit, with batch normalization

(c) 32-bit

(d) 32-bit, without batch normalization

Figure 3.7: Training a VGG network on CIFAR10 with SGD. We performed ten random initializations for each experiment, depicted by the boxplots and the filled contours (standard deviation).

Training and weight differences: We trained seven VGG11 networks $\{P_i\}_{i=0}^{6}$ at 32-bit precision on CIFAR10 for 200 epochs, using 128-size mini-batches, fixed learning rate for each mini-batch q: $\alpha_q = 1.0$. All networks started with the same parameters and varied by hybrid MaxPool values $\{\beta_i\}_{i=0}^{6}$. Using non-deterministic GPU computation, we measured backpropagation differences between P_0 and the others, tracking parameter variations and test accuracies. As shown in Figure 3.8, for $\beta \leq 10^3$, results were consistent, showing minimal impact from β. However, at $\beta = 10^4$, significant divergences and a drop in test accuracy occurred, indicating that high β values can destabilize training due to exploding gradients.

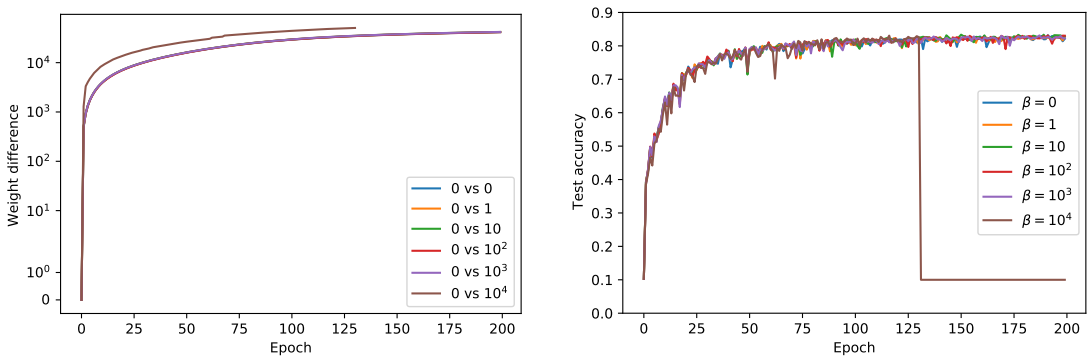


Figure 3.8: Left: Difference between network parameters (L^1 norm) at each epoch. "0 vs 0" indicates $\|\theta_{k,P_0} - \theta_{k,P_7}\|_1$ where P_7 is a second run of P_0 for sanity check, "0 vs 1" indicates $\|\theta_{k,P_0} - \theta_{k,P_1}\|_1$. Right: test accuracy of each $\{P_i\}_{i=0}^5$ for 200 epochs.

Recommendation for practitioners: Our findings show that large β values can destabilize training and reduce test accuracy, making them impractical for real-world use. For realistic β values, we observed no impact on training loss or accuracy. We recommend using low-norm Jacobians to ensure stable training, with $\beta = 1$ yielding the minimal MaxPool Jacobian norm. Additionally, using the Adam optimizer at 32-bit precision, as suggested by [41], helps mitigate the negative effects of large β values and stabilizes training (see Appendix 3.7.2).

Connexion with the choice of $\mathrm{ReLU}'(0)$**:** Initially, [41] reported that the choice of $\mathrm{ReLU}'(0)$ significantly impacts learning, with vanilla SGD training showing $\mathrm{ReLU}'(0) = 0$ as the most efficient option. A recent erratum published by the same authors [132] revises this finding, indicating that the impact of $\mathrm{ReLU}'(0)$ on learning outcomes is considerably less pronounced than initially stated.

B Appendix of Chapter 3

3.5 Further comments, discussion, and technical elements

3.5.1 AD errors with ReLU-derived programs

We conduct a small PyTorch experiment using the nonsmooth function ReLU: $x \mapsto \max(x, 0)$. Consider two programs $\max_1$ and $\max_2$ implementing the max: $x \mapsto \max_{1 \leq i \leq 4} x_i \in \mathbb{R}$ function using different ReLU-derived programs. Note that $2\max(x, y) = (x + y) + (\mathrm{ReLU}(x) - \mathrm{ReLU}(-y)) + (\mathrm{ReLU}(y) - \mathrm{ReLU}(-x))$. Let $\mathrm{zero}_2 : t \mapsto \max_1(t \times x) - \max_2(t \times x)$ be a program implementing the null function as described in Figure 3.9. Let zero_2' denote the backward AD algorithm for the zero program. As mathematical functions, $\max_1$ and $\max_2$ are equal and the program zero outputs constantly 0. However, for some $t \in \mathbb{R}$, AD can return $\mathrm{zero}_2'(t) \neq 0$. Results are reported in Table 3.3 and similar to Table 3.1.

```
def relu(x):
    return torch.relu(x)

def relu2(x):
    return torch.where(x >= 0, x, torch.tensor(0.0))

def max01(x):
    return (x[0] + x[1]) / 2 + relu((x[0] - x[1]) / 2) + relu((x[1] - x[0]) /
    ↪  2)

def max02(x):
    return (x[0] + x[1]) / 2 + relu2((x[0] - x[1]) / 2) + relu((x[1] - x[0]) /
    ↪  2)

def max1(x):
    return max01(torch.stack([max01(x[0:2]), max01(x[2:4])]))

def max2(x):
    return max02(torch.stack([max02(x[0:2]), max02(x[2:4])]))

def zero_2(t):
    z = t * x
    return max1(z) - max2(z)
```

Figure 3.9: Implementation of $\max_1$, $\max_2$ and zero_2 using Pytorch. Programs $\max_1$ and $\max_2$ are an equivalent implementation of max, but implemented using different ReLU-derived programs.

					$\text{zero}_2'(t)$						
t					-10^{-3}	-10^{-2}	-10^{-1}	0	10^{1}	10^{2}	10^{3}
$x =$	1.0	2.0	3.0	4.0	0.0	0.0	0.0	1.5	0.0	0.0	0.0
$x =$	1.4	1.4	1.4	1.4	10^{-7}	10^{-7}	10^{-7}	10^{-7}	10^{-7}	10^{-7}	10^{-7}

Table 3.3: Summary of various types of AD errors with zero_2 program using PyTorch for different combinations of t and x.

3.5.2 NormPool : a nonsmooth multivariate operation without compensation errors

We conducted an experiment to show that compensation errors are not caused by the multivariate nature of nonsmooth elementary functions when using floating-point arithmetic. In this experiment, we used the NormPool operation, which is similar to the MaxPool operation but replaces the maximum with the Euclidian norm. Two programs, P and Q, were used to implement a LeNet-5 network on the MNIST dataset with two different NormPool-derived programs. We computed the backprop variation (see Definition 11) between P and Q, while controlling all sources of divergence in our implementation using deterministic computation. The results are presented in Figure 3.10. The experiment was conducted on a CPU with 16-bit floating-point precision.

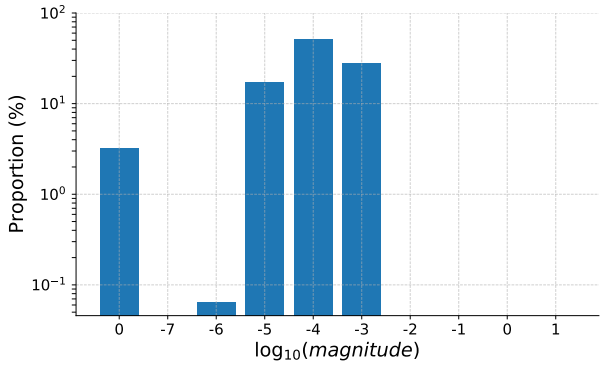


Figure 3.10: Histogram of backprop variation between P and Q for a LeNet-5 network on MNIST (128 mini-batch size) with 16-bit. We run $M = 1000$ experiments.

In contrast to our findings with MaxPool, we obtained similar results to those reported in [41] with ReLU-based programs. Specifically, for NormPool-based programs, we observed either significant divergence of backprop or none.

3.5.3 Bifurcation zone: a practical example

This section presents an example that demonstrates cases where AD can be incorrect. Calculating the accurate derivative for all inputs might be impossible, particularly when the function is nondifferentiable. This is because the derivative does not exist for inputs where the function is nondifferentiable.

Network configuration

Consider an input matrix X of size 4×4 given by:

$$X = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad \text{(Input)}$$

Let k be a positive number and W be a convolution kernel of size 3×3 given by:

$$W = k \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \qquad \text{(Convolution kernel)}$$

Let's consider a composition function l such that:

$$l(W) = \text{MaxPool} \circ (X * W) = k \tag{3.16}$$

where the convolution operation $X * W$ produces an output matrix Z of size 2×2, followed by the application of a MaxPool with a pooling window of size 2×2.

Backprop computation: native vs minimal

Let P (resp. Q) be a program implementing the composition function l in Equation (3.16) using the native (resp. minimal) MaxPool-derived program. Then, we have:

$$\text{backprop}[P](W) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad \text{backprop}[Q](W) = \begin{pmatrix} 0.5 & 0 & 0.5 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The convolutional kernel W falls within the bifurcation zone defined in Definition 7.

3.5.4 Comments on Section 3.3

Non-determinism in GPU computation

Graphics Processing Units (GPUs) are designed for parallel processing, which can result in unpredictable behaviors.

- **Floating-point operations:** The non-associative nature of floating-point arithmetic can lead to discrepancies. These differences might become significant as they accumulate across operations.
- **Reduction operations:** Functions like sum or maximum, especially in GPUs, can exhibit variability between runs. This variability can result in divergent accumulated rounding errors.

Network f	**Dataset**	$\tau^1_{f,16}$	$\tau^2_{f,16}$	$\tau^1_{f,32}$	$\tau^2_{f,32}$	$\tau^1_{f,64}$	$\tau^2_{f,64}$
LeNet-5	MNIST	0	10^{-5}	10^{-6}	10^{-5}	10^{-14}	0
VGG-11	CIFAR-10	0	10^{-1}	10^{-8}	10^{-7}	10^{-14}	0
VGG-11	SVHN	0	10^{-1}	10^{-8}	10^{-7}	10^{-15}	0
VGG-13	CIFAR-10	0	10^{-1}	10^{-9}	10^{-9}	10^{-14}	0
VGG-16	CIFAR-10	0	10^{-2}	10^{-10}	10^{-9}	10^{-15}	0
VGG-19	CIFAR-10	0	10^{-3}	10^{-11}	10^{-10}	10^{-15}	0
ResNet-18	CIFAR-10	10^{-2}	1	10^{-3}	10^{-4}	10^{-13}	0
DenseNet-121	CIFAR-100	0	10^{-2}	10^{-6}	10^{-1}	10^{-14}	0

Table 3.4: Threshold values of various neural networks f across different datasets.

Threshold values for various networks in Section 3.3.1

Table 3.4 presents threshold values for various neural networks on different datasets, computed under different floating-point precisions (16-bit, 32-bit, and 64-bit) and as explained in Section 3.3.1. For simplicity, thresholds are approximated as powers of 10.

Details on Monte Carlo sampling in Section 3.3.2

Recall that, for a neural network f and a floating-point precision ω, we want to estimate the volume of the set

$$S(\tau_{f,\omega}) = \left\{\theta \in \mathbb{R}^P : \exists i \in \{1,\dots,N\}, \|\text{backprop}[P_i](\theta) - \text{backprop}[Q_i](\theta)\|_1 > \tau_{f,\omega}\right\},$$

and we have $S(\tau_{f,\omega}) \subset \Theta_B$.

Our experiments divide a dataset into R mini-batches. Each r-th mini-batch is represented by the index set $B_r \subset \{1,\dots,N\}$. The programs P_r and Q_r are associated with the neural network f and implement a composition function l_r for each r. Specifically, P_r uses the native MaxPool-derived program, whereas Q_r uses the minimal one. For every precision level $\omega \in \{16, 32, 64\}$, we establish a threshold $\tau_{f,\omega}$ as in Section 3.3. Using the Kaiming-Uniform [130] initialization in PyTorch, we randomly generate a parameter set $\{\theta_j\}_{j=1}^M$, with $M = 1000$. The first line of Table 3.2 is given by the formula

$$\frac{1}{M}\sum_{j=1}^{K} \mathbb{1}\left(\exists r \in \{1,\dots,R\}, \left\|\text{backprop}\left[\sum_{j\in B_r} P_j(\theta)\right] - \text{backprop}\left[\sum_{j\in B_r} Q_j(\theta)\right]\right\|_1 > \tau_{f,\omega}\right), \tag{3.17}$$

where $\mathbb{1}$ represents the indicator function, returning either 1 or 0 depending on the truth value of its argument's condition. Similarly, the second line of Table 3.2 is given by the formula

$$\frac{1}{MR}\sum_{j=1}^{M}\sum_{r=1}^{R} \mathbb{1}\left(\left\|\text{backprop}\left[\sum_{j\in B_r} P_j(\theta)\right] - \text{backprop}\left[\sum_{j\in B_r} Q_j(\theta)\right]\right\|_1 > \tau_{f,\omega}\right), \tag{3.18}$$

Using the formula

$$\sqrt{\frac{\ln\left(\frac{2}{\alpha}\right)}{2n}},$$

and setting $\alpha = 0.05$, we compute the error margin of the Hoeffding confidence interval as $n = M$ for Table 3.2's first line and $n = MR$ for its second. The first line adheres to a 95% confidence interval under the *iid* assumption due to Hoeffding's inequality.

Using McDiarmid's inequality at risk level $\alpha = 0.05$, we compute the error margin of the second line in Table 3.2 by the formula

$$\sqrt{\frac{1}{2}\left(\frac{1}{M}+\frac{1}{R}\right)\ln\left(\frac{2}{\alpha}\right)}.$$

3.6 Proof related to Section 3.2.3

Proof of Proposition 5:

1 The three subsets have unique definitions, indicating that they are separate. For instance, a parameter cannot belong to the regular and bifurcation zones since the regular zone is defined as the area where each program $g_{i,j}$ is assessed at differentiable points. On the other hand, the bifurcation zone is defined as the region where the set of all possible backprop outputs is not a singleton, indicating non-differentiability at some points. Additionally, the union of these zones covers the entire parameter space Θ as every parameter must be assigned to one of the three subsets: resulting in differentiable points when evaluated, resulting in nondifferentiable points but having a singleton backprop set, or resulting in nondifferentiable points with a non-singleton backprop set. Therefore, $\Theta_R \cup \Theta_B \cup \Theta_C = \Theta$.

2 As we consider locally Lipchitz semialgebraic (or definable) functions, see [Theorem 1, [10]] for the proof arguments.

□

3.7 Complements on experiments

3.7.1 Benchmark datasets and architectures

Datasets: In this work, we utilized various well-known image classification benchmarks. Below are the datasets, including their characteristics and original references.

Dataset	Dimensionality	Training set	Test set
MNIST	28×28 (grayscale)	60K	10K
CIFAR10	32×32 (RGB)	60K	10K
SVHN	32×32 (RGB)	600K	26K
ImageNet	224×224 (RGB)	1.3M	50K

The corresponding references for these datasets are [23], [49], [133].

Neural network architectures: We evaluated various CNN neural network architectures, with details as follows:

Name	Layers	Loss function
LeNet-5	5	Cross-entropy
VGG11	11	Cross-entropy
VGG13	13	Cross-entropy
VGG16	16	Cross-entropy
VGG19	19	Cross-entropy
ResNet18	18	Cross-entropy
ResNet50	50	Cross-entropy
DenseNet121	125	Cross-entropy

The corresponding references for these architectures are [45], [49], [131], [134].

LeNet-5: The implementation for LeNet-5 was sourced from the following GitHub repository: `https://github.com/ChawDoe/LeNet5-MNIST-PyTorch/blob/master/model.py`.

VGG: We used the PyTorch repository's implementation for the VGG models. It can be accessed at the following link: `https://github.com/PyTorch/vision/blob/main/torchvision/models/vgg.py`.

ResNet: For ResNet models, we utilized the PyTorch repository's implementation available at: `https://github.com/PyTorch/vision/blob/main/torchvision/models/resnet.py`. We made minor adjustments to the output layer's size (changing from 1000 to 10 classes) and the kernel size in the primary convolutional, varying from 7 to 3). When batch normalization was not used, we replaced the batch normalization layers with identity mappings.

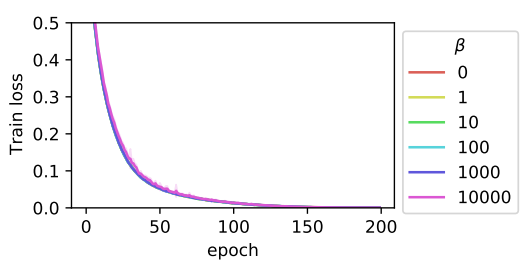
DenseNet: The implementation for DenseNet was taken from the PyTorch repository, available at: `https://github.com/PyTorch/vision/blob/main/torchvision/models/densenet.py`.

3.7.2 Mitigating factor: Adam optimizer

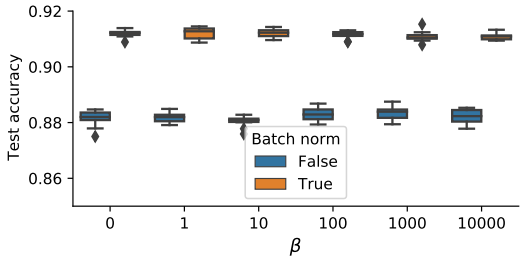
After training a VGG11 network on CIFAR-10 using the Adam optimizer, we obtained results shown in Figure 3.11. Our findings are consistent with those presented in Section 3.4, but the network exhibits reduced sensitivity to β, resulting in improved stability of both test errors and training loss.

3.7.3 Additional experiments with MNIST and LeNet-5 networks

We repeated the experiments in Section 3.4.2 using a LeNet-5 network on the MNIST dataset. The results are depicted in Figure 3.12. We found that for 16 bits, the test accuracies were similar when training was possible, but $\beta = \{10^3, 10^4\}$ caused chaotic training behavior. For 32 bits, the test accuracies were mostly



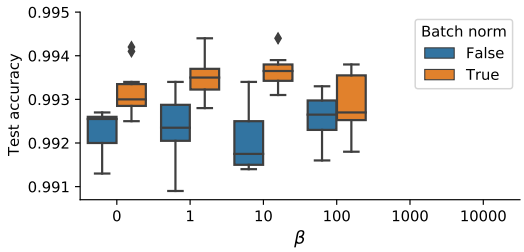
(a) 32-bit without batch normalization



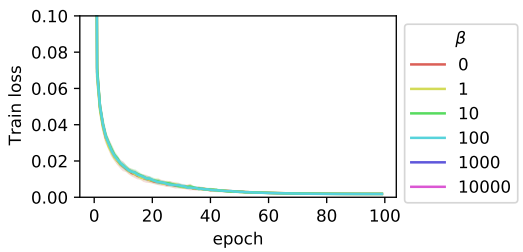
(b) 32-bit

Figure 3.11: Training losses on CIFAR10 (left) and test accuracy (right) on VGG network trained with Adam optimizer and without batch normalization.

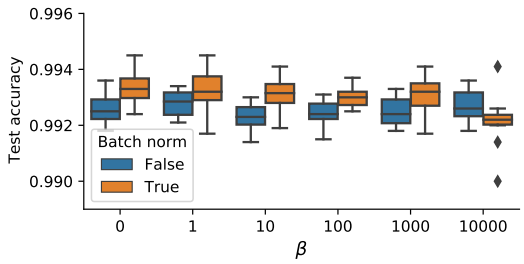
similar, except for $\beta = 10^4$. We noticed that the chaotic oscillations had completely disappeared.



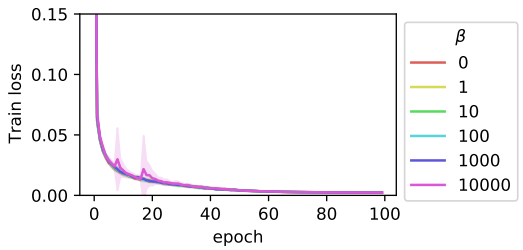
(a) 16-bit



(b) 16-bit, with batch normalization



(c) 32-bit

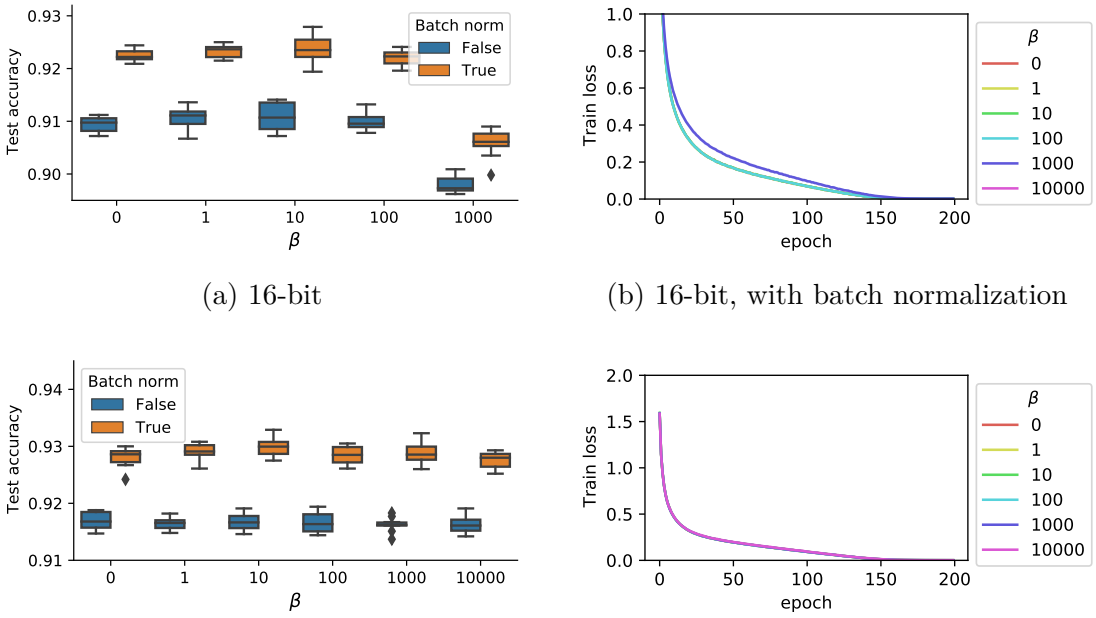


(d) 32-bit, without batch normalization

Figure 3.12: Training a LeNet-5 network on MNIST with SGD. We performed ten random initializations for each experiment, depicted by the boxplots and the filled contours (standard deviation).

3.7.4 Additional experiments with ResNet18

We performed the same experiments described in Section 3.4.2 using ResNet18 architecture trained on CIFAR 10. Figure 3.13 represents the test errors with or without batch normalization. For 16 bits, test accuracies are similar, but $\beta = 10^4$ induces chaotic training behavior. For 32 bits, test accuracies are identical, and the chaotic oscillations phenomena have entirely disappeared.



(a) 16-bit

(b) 16-bit, with batch normalization

(c) 32-bit

(d) 32-bit, with batch normalization

Figure 3.13: Training a ResNet18 network on CIFAR10 with SGD. We performed ten random initializations for each experiment, depicted by the boxplots and the filled contours (standard deviation).

3.7.5 Additional experiments with ResNet50 on ImageNet

We performed the same experiments described in Section 3.4.2 using a ResNet50 architecture trained on ImageNet. The test accuracy is represented in Figure 3.14. We employ mixed precision [57], [135], utilizing 16 and 32 bits precision to balance computational speed and information retention. Test accuracies are similar when training is possible, but $\beta = 10^3$ induces chaotic training behavior.

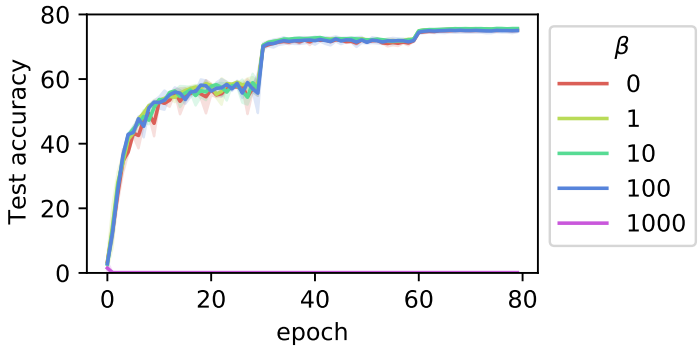


Figure 3.14: Test accuracy during training a Resnet50 on ImageNet with SGD using mixed precision. The shaded area represents three runs. We have a chaotic test accuracy behavior for $\beta = 10^3$.

Chapter 4

A second-order-like optimizer with adaptive gradient scaling for deep learning

Abstract

In this empirical part, we introduce INNAprop, an optimization algorithm that combines the INNA method with the RMSprop adaptive gradient scaling. It leverages second-order information and rescaling while keeping the memory requirements of standard DL methods as AdamW or SGD with momentum. After giving geometrical insights, we evaluate INNAprop on CIFAR-10, Food101, and ImageNet with ResNets, VGG, DenseNet, and ViT, and on GPT-2 (OpenWebText) train from scratch and with LoRA fine-tuning (E2E). INNAprop consistently matches or outperforms AdamW both in training speed and accuracy, with minimal hyperparameter tuning in large-scale settings.

This Part is organized as follows:

- In Section 4.1, we present the background and related work.
- In Section 4.2, we describe our algorithm and its derivation.
- In Section 4.3, we provide hyperparameter tuning recommendations and our experimental results.

4.1 Introduction

4.1.1 Motivations

As deep learning models grow in size, massive computational resources are needed for training, representing significant challenges in terms of financial costs, energy consumption, and processing time [136], [137]. According to the UN's Environment Programme Training, the Big Tech sector produced between two and three percent of the world's carbon emissions in 2021; some estimations for the year 2023 go beyond 4%, see the latest Stand.earth reports, and also [28], [29], [138] for related issues. For instance, training GPT-3 is estimated to require 1,287 megawatt-hours (MWh) of electricity, equivalent to the annual usage of over 100 U.S. households [29], [139]. Similarly, the financial cost of specialized hardware and cloud computing is extremely high. OpenAI claimed that the training cost for GPT-4 [140] exceeded 100 million dollars. The PaLM model developed by Google AI was trained for two months using 6144 TPUs for 10 million dollars [141]. All this implies a need for faster and more cost-efficient optimization algorithms. It also suggests that early stopping [142], [143] in the training phase is a desirable feature whenever possible.

We focus in this work on computational efficiency during the training phase and consider the problem of unconstrained minimization of a loss function $\mathcal{J}\colon \mathbb{R}^p \to \mathbb{R}$, as follows

$$\min_{\theta\in\mathbb{R}^p} \mathcal{J}(\theta). \tag{4.1}$$

4.1.2 Continuous dynamical systems as optimization models

To achieve higher efficiency, it is necessary to deeply understand how algorithms work and how they relate to each other. A useful way to do this is by interpreting optimization algorithms as discrete versions of continuous dynamical systems [144], further developed in [145]–[149]. In deep learning, this approach is also quite fruitful; it has, in particular, been used to provide convergence proofs or further geometric insights [10], [37], [150], [151].

In the spirit of [12], we consider the following continuous-time dynamical system introduced in [152] and referred to as DIN (standing for "dynamical inertial Newton"):

$$\underbrace{\ddot{\theta}(t)}_{\text{Inertial term}} + \underbrace{\alpha\,\dot{\theta}(t)}_{\text{Friction term}} + \underbrace{\beta\,\nabla^2\mathcal{J}(\theta(t))\dot{\theta}(t)}_{\text{Newtonian effects}} + \underbrace{\nabla\mathcal{J}(\theta(t))}_{\text{Gravity effect}} = 0, \qquad t \geq 0, \tag{4.2}$$

where t is the time, $\mathcal{J}\colon \mathbb{R}^p \to \mathbb{R}$ is a loss function to be minimized (e.g., empirical loss in DL applications) as in Equation (4.1), assumed C^2 with gradient $\nabla\mathcal{J}$ and Hessian $\nabla^2\mathcal{J}$. A key aspect of Equation (4.2) that places it between first- and second-order optimization is that a change of variables allows to describe it using only the gradient $\nabla\mathcal{J}$, since $\nabla^2\mathcal{J}(\theta(t))\dot{\theta}(t) = \frac{d}{dt}\nabla\mathcal{J}(\theta(t))$ (see Section 4.2.2 for details). INNA is a discretized version of Equation (4.2), specifically adapted for deep learning applications. This greatly reduces computational costs, as it can be discretized as a difference of gradients which does not require Hessian vector

product, making it possible to design more practical algorithms, as shown in [12], [153], [154].

We recover the continuous-time heavy ball system by assuming $\alpha > 0$, and removing the geometrical "damping" term in Equation (4.2) through the choice $\beta = 0$. A discrete version of this system corresponds to the Heavy Ball method [62], which is at the basis of SGD solvers with momentum in deep learning [63], [155]. By allowing both α and β to vary, we recover Nesterov acceleration [64], [156], [157].

Adaptive methods. Adaptive optimization methods, such as RMSprop [13] and AdaGrad [31], modify the update dynamics by introducing coordinate-wise scaling of the gradient based on past information. These methods can be modeled by continuous-time ODEs of the following form, expressed here for the simple gradient system:

$$\dot{\theta}(t) + \frac{1}{\sqrt{G(t,\theta(t)) + \epsilon}} \odot \nabla \mathcal{J}(\theta(t)) = 0, \quad t \geq 0, \tag{4.3}$$

where $\epsilon > 0$, $G(t,\theta(t)) \in \mathbb{R}^p$ represents accumulated information. The scalar addition, square root, and division are understood coordinatewise and $\odot$ denotes the coordinate-wise product for vectors in $\mathbb{R}^p$. In AdaGrad or RMSprop, $G(t,\theta(t))$ is defined as an accumulation of squared gradient coordinates of the form:

$$G(t,\theta(t)) := \int_0^t \nabla \mathcal{J}(\theta(\tau))^2 \, d\mu_t(\tau), \tag{4.4}$$

for different choices of μ_t (uniform for AdaGrad and moving average for RMSprop). Both approaches scale the gradient based on accumulated information on past gradient magnitudes, improving performance, particularly in settings with sparse or noisy gradients [13], [31].

Our approach. We combine the "dynamical inertial Newton" method (DIN) from Equation (4.2) with an RMSprop adaptive gradient scaling. This allows us to take into account second-order information for the RMSProp scaling. Computationally, this second-order information is expressed using a time derivative. In discrete time, this will provide a second-order intelligence with the same computational cost as gradient evaluation. The resulting continuous time ODE is given as follows:

$$\ddot{\theta}(t) + \alpha\,\dot{\theta}(t) + \beta\,\frac{d}{dt}\mathrm{RMSprop}(\mathcal{J}(\theta(t))) + \mathrm{RMSprop}(\mathcal{J}(\theta(t))) = 0, \qquad t \geq 0$$
$$\text{where } \mathrm{RMSprop}(\mathcal{J}(\theta(t))) = \frac{1}{\sqrt{G(t,\theta(t)) + \epsilon}} \odot \nabla \mathcal{J}(\theta(t)) \tag{4.5}$$

with G of the form (4.4) with an adequate time-weight distribution μ_t corresponding to the RMSProp scaling. A discretization of this continuous time system, combined with careful memory management, results in our new optimizer INNAprop, see Section 4.2.1.

4.1.3 Related works and contributions

Relation with existing work. To improve the efficiency of stochastic gradient descent (SGD), two primary strategies are used: leverage local geometry for having clever directions and incorporate momentum to accelerate convergence. These approaches include accelerated methods (e.g., Nesterov's acceleration [64], [158], momentum SGD [62], [63], [155], and adaptive methods (e.g., Adagrad [31], RMSProp [13]), which adjust learning rates per parameter.

Adam remains the dominant optimizer in deep learning. It comes under numerous variants proposed to improve its performance or to adapt it to specific cases [32], [66], [68], [158], [159]. Adafactor [159] improves memory efficiency, Lamb [160] adds layerwise normalization, and Lion [33] uses sign-based momentum updates. AdEMAMix [161] combines two EMAs, while Defazio et al. [162] introduced a schedule-free method incorporating Polyak-Ruppert averaging with momentum.

One of the motivations of our work is the introduction of second-order properties in the dynamics akin to Newton's method. Second-order optimizers are computationally expensive due to frequent Hessian computations [73], [163] and their adaptation to large scale learning settings require specific developments [164], [165]. For example, the Sophia optimizer [74], designed for large language models, uses a Hessian-based pre-conditioner to penalize high-curvature directions. In this work, we draw inspiration from the INNA optimizer [12], based on the DIN continuous time dynamics introduced by [152], which combines gradient descent with a Newtonian mechanism for first-order stochastic approximations.

Our proposed method, INNAprop, integrates the algorithm INNA, which features a Newtonian effect with cheap computational cost, with the gradient scaling mechanism of RMSprop. This framework preserves the efficiency of second-order methods and the adaptive features of RMSprop while significantly reducing the computational overhead caused by Hessian evaluation. Specific hyperparameter choices for our method allow us to recover several existing optimizers as special cases.

Contributions. They can be summarized as follows:

- We introduce INNAprop, a new optimization algorithm that combines the Dynamical Inertial Newton (DIN) method with RMSprop's adaptive gradient scaling, efficiently using second-order information for large-scale machine learning tasks. We obtain a second-order optimizer with computational requirements similar to first-order methods like AdamW, making it suitable for deep learning (see Section 4.2.2 and Appendix 4.5).
- We provide a continuous-time explanation of INNAprop, connecting it to second-order ordinary differential equations (see Section 4.2 and Equation (4.5)). We discuss many natural possible discretizations and show that INNAprop is empirically the most efficient. Let us highlight a key feature of our method: it incorporates second-order terms in space without relying on Hessian computations or inversions of linear systems which are both prohibitive in deep learning.

- We show through extensive experiments that INNAprop matches or outperforms AdamW in both training speed and final accuracy on benchmarks such as image classification (CIFAR-10, ImageNet) and language modeling (GPT-2) (see Section 4.3).

4.2 INNAprop: a second-order method in space and time based on RMSProp

4.2.1 The algorithm

Our method is built on the following Algorithm 3, itself derived from a combination of INNA [12] and RMSprop [13] (refer to Section 4.2.2 for more details). The following version of the method is the one we used in all experiments. It includes the usual ingredients of deep-learning training: mini-batching, decoupled weight-decay, and scheduler procedure. For a simpler, "non-deep learning" version, refer to Algorithm 4 in Appendix 4.5.

Algorithm 3: Deep learning implementation of INNAprop

1: **Objective function:** $\mathcal{J}(\theta) = \frac{1}{n}\sum_{n=1}^{N} \mathcal{J}_n(\theta)$ for $\theta \in \mathbb{R}^p$.
2: **Learning step-sizes:** $\gamma_k := \{\text{SetLrSchedule}(k)\}_{k\in\mathbb{N}}$ where γ_0 is the initial learning rate.
3: **Hyper-parameters:** $\sigma \in [0,1]$, $\alpha \geq 0$, $\beta > \sup_{k\in\mathbb{N}} \gamma_k$, $\lambda \geq 0$, $\epsilon = 10^{-8}$.
4: **Mini-batches:** $(\mathsf{B}_k)_{k\in\mathbb{N}}$ of nonempty subsets of $\{1,\dots,N\}$.
5: **Initialization:** time step $k \leftarrow 0$, parameter vector θ_0, $v_0 = 0$, $\psi_0 = (1-\alpha\beta)\theta_0$.
6: **for** $k = 1$ **to** K **do**
7: $\quad \boldsymbol{g}_k = \frac{1}{|\mathsf{B}_k|}\sum_{n\in\mathsf{B}_k} \nabla\mathcal{J}_n(\boldsymbol{\theta}_k)$ ▷ select batch B_k and return the corresponding gradient
8: $\quad \gamma_k \leftarrow \text{SetLrSchedule}(k)$ ▷ see above and Remark 12
9: $\quad \boldsymbol{\theta}_k \leftarrow (1-\lambda\gamma_k)\boldsymbol{\theta}_k$ ▷ decoupled weight decay
10: $\quad \boldsymbol{v}_{k+1} \leftarrow \sigma\boldsymbol{v}_k + (1-\sigma)\boldsymbol{g}_k^2$
11: $\quad \hat{\boldsymbol{v}}_{k+1} \leftarrow \boldsymbol{v}_{k+1}/(1-\sigma^k)$
12: $\quad \boldsymbol{\psi}_{k+1} \leftarrow \left(1-\frac{\gamma_k}{\beta}\right)\boldsymbol{\psi}_k + \gamma_k\left(\frac{1}{\beta}-\alpha\right)\boldsymbol{\theta}_k$
13: $\quad \boldsymbol{\theta}_{k+1} \leftarrow \left(1+\frac{\gamma_k(1-\alpha\beta)}{\beta-\gamma_k}\right)\boldsymbol{\theta}_k - \frac{\gamma_k}{\beta-\gamma_k}\boldsymbol{\psi}_{k+1} - \gamma_k\beta\left(\boldsymbol{g}_k/(\sqrt{\hat{\boldsymbol{v}}_{k+1}}+\epsilon)\right)$
14: **return** $\boldsymbol{\theta}_{K+1}$

In Algorithm 3, SetLrSchedule is the "scheduler" for step-sizes; it is defined as a custom procedure for handling learning rate sequences for different networks and databases. To provide a full description of our algorithm, we provide detailed explanations of the scheduler procedures used in our experiments (Section 4.3) in Appendix 4.7, along with the corresponding benchmarks.

Remark 12 (Well posedness) Observe that, for all schedulers $\gamma_k < \beta$ for $k \in \mathbb{N}$, so that INNAprop is well-posed (line 13 in Algorithm 3, the division is well defined).

4.2.2 Derivation of the algorithm

There are several ways to combine RMSprop and INNA, or DIN its second-order form, as there exist several ways to do so with the heavy ball method and RMSprop. We opted for the approach below because of its mechanical and geometrical appeal and its numerical success (see Appendix 4.5 for further details). Consider the following dynamical inertial Newton method [152]:

$$\ddot{\theta}(t) + \alpha\,\dot{\theta}(t) + \beta\,\frac{d}{dt}\nabla\mathcal{J}(\theta(t)) + \nabla\mathcal{J}(\theta(t)) = \ 0, \quad t \geq 0, \tag{4.6}$$

as in Equation (4.2) and replacing $\nabla^2\mathcal{J}(\theta(t))\dot{\theta}(t)$ by $\frac{d}{dt}\nabla\mathcal{J}(\theta(t))$. We use finite differences with a fixed time step γ to discretize this system, replacing in particular the gradient derivatives by gradient differences:

$$\frac{d}{dt}\nabla\mathcal{J}(\theta(t)) \simeq \frac{\nabla\mathcal{J}(\theta_{k+1}) - \nabla\mathcal{J}(\theta_k)}{\gamma},$$

where θ_k, θ_{k+1} correspond to two successive states around the time t.

Setting $\nabla\mathcal{J}(\theta_k) = g_k$, we obtain

$$\frac{\theta_{k+1} - 2\theta_k + \theta_{k-1}}{\gamma} + \alpha\frac{\theta_k - \theta_{k-1}}{\gamma} + \beta\frac{g_k - g_{k-1}}{\gamma} + g_{k-1} = 0. \tag{4.7}$$

To provide our algorithm with an extra second-order geometrical intelligence, we use the proxy of RMSprop direction in place of the gradient.

Choose $\sigma > 0$ and $\epsilon > 0$, and consider:

$$v_{k+1} = \sigma v_k + (1-\sigma)g_k^2 \tag{4.8}$$

$$\frac{\theta_{k+1} - 2\theta_k + \theta_{k-1}}{\gamma} + \alpha\frac{\theta_k - \theta_{k-1}}{\gamma} + \beta\frac{\frac{g_k}{\sqrt{v_{k+1}}+\epsilon} - \frac{g_{k-1}}{\sqrt{v_k}+\epsilon}}{\gamma} + \frac{g_{k-1}}{\sqrt{v_k}+\epsilon} = 0. \tag{4.9}$$

Although this system has a natural mechanical interpretation, its memory footprint is abnormally important for this type of algorithm: for one iteration of the system (4.8)-(4.9), it culminates at 6 full dimension memory slots, namely g_{k-1}, g_k, θ_{k-1}, θ_k, v_k, and v_{k+1} before the evaluation of (4.9).

Therefore, we proceed to rewrite the algorithm in another system of coordinates. The computations and the variable changes are provided in Appendix 4.5. We eventually obtain:

$$\begin{aligned}
v_{k+1} &= \sigma v_k + (1-\sigma)g_k^2 \\
\psi_{k+1} &= \psi_k\left(1 - \frac{\gamma}{\beta}\right) + \gamma\left(\frac{1}{\beta} - \alpha\right)\theta_k, \\
\theta_{k+1} &= \left(1 + \frac{\gamma(1-\beta\alpha)}{\beta-\gamma}\right)\theta_k - \frac{\gamma}{\beta-\gamma}\psi_{k+1} - \gamma\beta\frac{g_k}{\sqrt{v_{k+1}}+\epsilon}
\end{aligned}$$

which only freezes 3 full dimension memory slots corresponding to v_k, ψ_k, θ_k. As a result, the memory footprint is equivalent to that of the Adam optimizer (see Table 4.5).

Remark 13 (A family of algorithms indexed by α, β) INNAprop can be seen as a family of methods indexed by the hyperparameters α and β. When $\beta = 0$, we recover a modified version of RMSprop with momentum [166] (see Appendix 4.5.1). For $\alpha = \beta = 1$, INNAprop with its default initialization, boils down to AdamW without momentum ($\beta_1 = 0$), see Appendix 4.5.1 and Table 4.5. By setting $\alpha = \beta = 1$, we empirically recover the behavior of AdamW. Experiments demonstrate that this consistently aligns with AdamW, suggesting that AdamW can be seen as a special case within the broader INNAprop family. See Appendix 4.5.1 for further details and illustrations.

Remark 14 (On other possible discretizations) (a) If we use the proxy of RMSprop directly with INNA [12], we recover indeed INNAprop through a rather direct derivation (see Appendix 4.6.1 for more details). Our motivation to start from the "mechanical" version of the algorithm is to enhance our understanding of the geometrical features of the algorithm.
(b) RMSprop with momentum [166] is obtained by a discretization of the heavy ball continuous time system, using a momentum term and an RMSprop proxy. It would be natural to proceed that way in our case, and it indeed leads to a different method (see Appendix 4.6.2). However, the resulting algorithm appears to be numerically unstable (see Figure 4.10 for an illustration).
(c) Incorporating RMSprop as it is done in Adam using momentum leads to a third method (see Appendix 4.6.3), which appears to be extremely similar to NAdam [158]; it was thus discarded. We now explain how these hyperparameters (α, β) have been tuned on "small size" problems.

4.3 Empirical evaluation of INNAprop

We conduct extensive comparisons of the proposed algorithm and the AdamW optimizer, which is dominantly used in image classification [66], [167]–[169] and language modeling tasks [74], [170], [171]. See Section 4.10 for more details. Hyperparameter tuning [172] is a crucial issue for this comparison, and we start with this. As a general rule, we strive to choose the hyperparameters that give a strong baseline for AdamW (based on literature or using grid search). Unless stated differently, our experiments use the AdamW optimizer [1] with its default settings as defined in widely-used libraries [5]–[7]: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\lambda = 0.01$ and $\epsilon = 1e-8$. For INNAprop, unless otherwise specified, the default settings for the RMSprop component align with those of AdamW: $\sigma = 0.999$ and $\epsilon = 1e-8$.

The INNAprop method and the AdamW optimizer involve different classes of hyperparameters; some of them are common to both algorithms, and some are specific. Our hyperparameter tuning strategy for both algorithms is summarized in Table 4.1.

We begin this section with the tuning of parameters α, β for INNAprop on CIFAR10 with VGG and ResNet architectures and then use these parameters on larger datasets and models. We use as much as possible the step size scheduler

[1] `https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html`

and weight decay settings reported in the literature for the AdamW optimizer, which we believe to be well-adjusted and provide adequate references for each experiment. These are used both for AdamW and INNAprop. With this protocol, we only perform minimal hyperparameter tuning for INNAprop for larger-scale experiments. This is due to constrained computational resources. We aim to demonstrate the typical performance of the Algorithm 3, rather than its peak performance with extensive tuning.

Table 4.1: Hyperparameter tuning strategy for INNAprop and AdamW: AdamW is systematically favored.

Parameters	**AdamW tuning**	**INNAprop tuning**	**Comparative advantage**
Learning rate	Literature or grid search tuning	Reused from AdamW	AdamW favored
Step size scheduler	Literature	Reused from AdamW	N/A
Weight decay λ	Literature or grid search tuning	Reused from AdamW	AdamW favored
RMSprop parameter (β_2, ϵ)	Default or literature	Reused from AdamW	AdamW favored
Inertial parameters (α, β)	N/A	Tuned on CIFAR-10	N/A

4.3.1 Tuning INNAprop on CIFAR-10 with VGG11 and ResNet18

Hyperparameter tuning: We tune (α, β) using VGG11 [131] and ResNet18 [45] models trained on CIFAR10 [23], together with the initial learning rate γ_0 to ensure proper training. We fix a cosine scheduler where $T_{\max} = 200$ and $\gamma_{\min} = 0$ (see Section 4.7 for more details) and we consider two weight decay parameters $\lambda = 0$ or $\lambda = 0.01$. Our experiment suggests using an initial learning rate $\gamma_0 = 10^{-3}$, which is the baseline value reported for AdamW in this experiment (see Section 4.8). For INNAprop, we optimize only the hyperparameters α and β, using test accuracy and training loss as the optimization criteria. A grid search is performed over $(\alpha, \beta) \in \{0.1, 0.5, 0.9, \ldots, 3.5, 4.0\}$ using `optuna` [173]. In Figure 4.1, we detail the obtained training loss and test accuracy for various (α, β) configurations over short training durations (20 epochs) and long training durations (200 epochs) for VGG11 with weight decay $\lambda = 0.01$. Our criteria (short and long training duration) are chosen to find parameters (α, β) that provide a rapid decrease in training loss in the early stages and the best test accuracy for long training duration.

These results highlight a tendency for efficient couples; we choose for further experiments the values $(\alpha, \beta) = (0.1, 0.9)$ which correspond to aggressive optimization of the training loss for short training durations, and $(\alpha, \beta) = (2.0, 2.0)$ which provides very good results for longer training durations. Additional results for VGG11 and ResNet18 with and without weight decay are in Appendix 4.8.2, which are qualitatively similar.

Validation and comparison with AdamW: We confirm our hyperparameter choices ($\gamma_0 = 10^{-3}$, $\lambda = 0.01$) by reproducing the experiment with 8 random seeds and comparing with AdamW using the same settings. Based on hyperparameter tuning, we select two pairs of (α, β) with different training speeds. As shown in

(a) 20 epochs

(b) 200 epochs

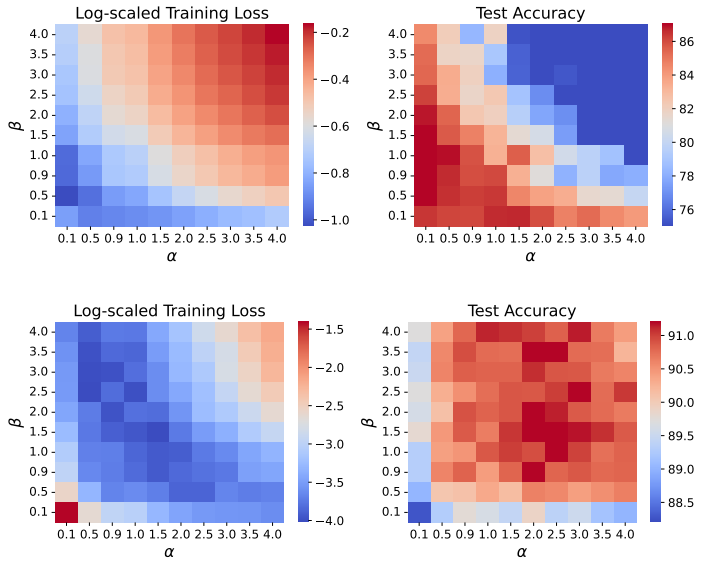


Figure 4.1: Log-scale training loss and test accuracies for hyperparameters (α, β) with VGG11 on CIFAR10 at 20 and 200 epochs. Optimal learning rate $\gamma_0 = 10^{-3}$ and weight decay $\lambda = 0.01$, with one random seed.

Figure 4.2 (and Section 4.9 for ResNet18), with $(\alpha, \beta) = (0.1, 0.9)$, INNAprop improves training loss and test accuracy rapidly by the 100th epoch, maintaining the highest training accuracy. With $(\alpha, \beta) = (2.0, 2.0)$, INNAprop trains more slowly but achieves higher final test accuracy. This is aligned with the experiments described in Figure 4.1. In Table 4.2, we compare the performance of different networks on CIFAR-10 using INNAprop and AdamW optimizers.

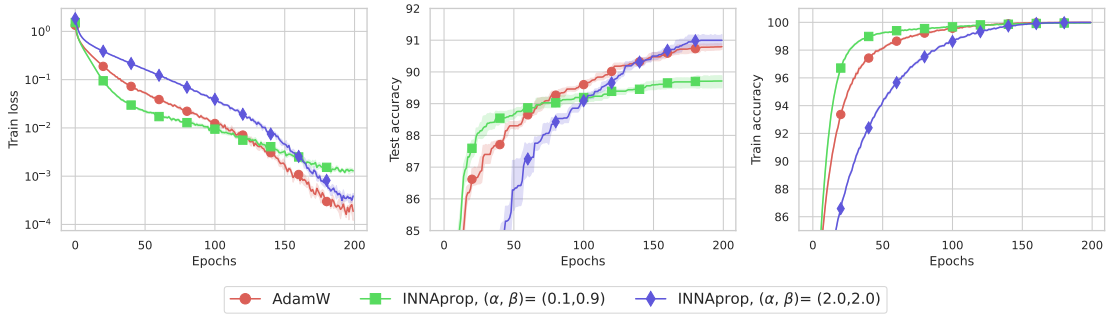


Figure 4.2: Training VGG11 on CIFAR10. Left: train loss, middle: test accuracy (%), right: train accuracy (%), with 8 random seeds.

Remark 15 (Trade-off between fast learning and good generalization) For CIFAR-10 experiments, INNAprop offers flexibility in adjusting convergence speed through (α, β). Faster training configurations generally lead to weaker generalization compared to slower ones, highlighting the trade-off between quick convergence and generalization [174], [175].

4.3.2 Extensive experiments on large-scale vision models

We present experiments on large-scale vision benchmarks with the hyperparameters of Section 4.3.1.

Table 4.2: Test accuracy (%) of ResNet-18, VGG11, and DenseNet121 on CIFAR-10 using AdamW optimized weight decay and learning rate. Results are averaged over eight runs.

Model	**Optimizer**	**Test accuracy**
Training on CIFAR-10 over 200 epochs		
ResNet18	AdamW	91.14
	INNAprop ($\alpha = 2.0, \beta = 2.0$)	**91.58**
VGG11	AdamW	90.79
	INNAprop ($\alpha = 2.0, \beta = 2.0$)	**90.99**
DenseNet121	AdamW	86.19
	INNAprop ($\alpha = 0.1, \beta = 0.9$)	**86.91**

Resnets on ImageNet: We consider the larger scale ImageNet-1k benchmark [20]. We train a ResNet-18 and a ResNet-50 [45] for 90 epochs with a mini-batch of size of 256 as in [33], [66]. We used the same cosine scheduler for both AdamW and INNAprop with initial learning rate $\gamma_0 = 10^{-3}$ as reported in [33], [66], [167]. The weight decay of AdamW is set to $\lambda = 0.01$ for the ResNet18, instead of $\lambda = 0.05$ reported in [66], [167] because it improved the test accuracy from 67.93 to 69.43. The results of the ResNet18 experiment are presented in Figure 4.17 in Section 4.9. The figure shows that our algorithm with $(\alpha, \beta) = (0.1, 0.9)$ outperforms AdamW in test accuracy (70.12 vs 69.34), though the training loss decreases faster initially but slows down towards the end of training.

For the ResNet50, we kept the value $\lambda = 0.1$ as reported in [66], [167]. For INNAprop, we tried two weight decay values $\{0.1, 0.01\}$ and selected $\lambda = 0.01$ as it resulted in a faster decrease in training loss. We report the results in Figure 4.3, illustrating the advantage of INNAprop. As discussed in Section 4.3.1, INNAprop with $(\alpha, \beta) = (0.1, 0.9)$ leads to a faster decrease in training loss but yields lower test accuracy compared to either AdamW or INNAprop with $(\alpha, \beta) = (2.0, 2.0)$. For $(\alpha, \beta) = (2.0, 2.0)$, the loss decrease is similar to AdamW, with no clear advantage for either method. This obviously suggests developing scheduling strategies for damping parameters (α, β). This would require a much more computation-intensive tuning, far beyond the numerical resources used in the current work. In Table 4.3, we present the performance of INNAprop achieved using minimal hyperparameter tuning, as explained in Table 4.1.

Vision transformer (ViT) on ImageNet: We performed the same experiment with a ViT-B/32 architecture over 300 epochs with a mini-batch size of 1024, following [169], [176]. For AdamW, we used a cosine scheduler with a linear warmup (30 epochs) and the initial learning rate and weight decay from [176]. For INNAprop, we tested weight decay values of $\{0.1, 0.01\}$, selecting $\lambda = 0.1$ for better test accuracy. Results in Figure 4.3 show the advantage of INNAprop. For faster convergence using INNAprop $(0.1, 0.9)$, we recommend a weight decay of $\lambda = 0.01$ (see Figure 4.18 in the Appendix).

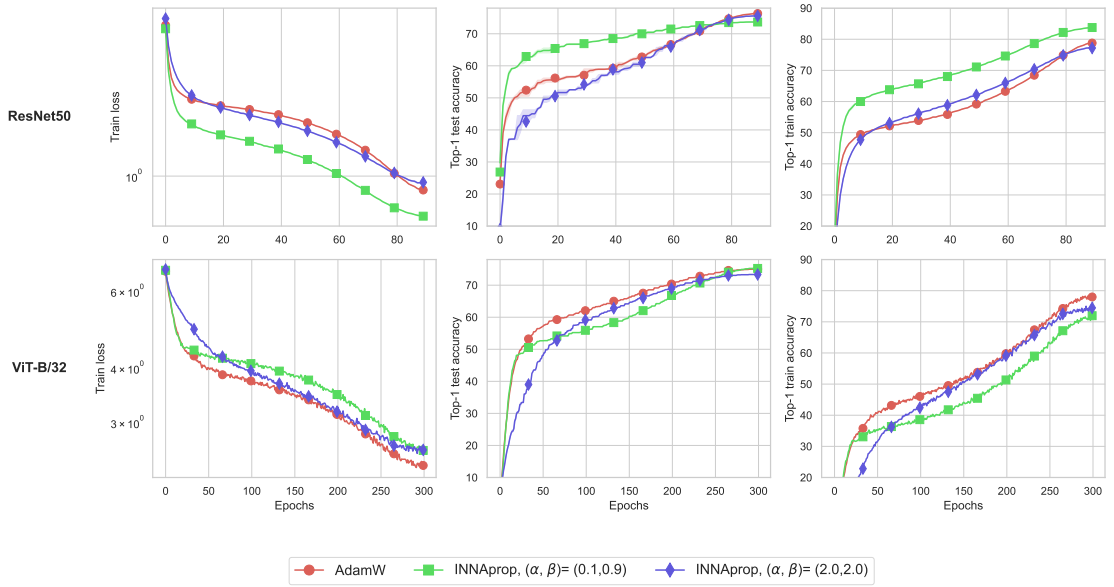


Figure 4.3: Training a ResNet50 (top) and ViT-B/32 (bottom) on ImageNet. Left: train loss, middle: Top-1 test accuracy (%), right: Top-1 train accuracy (%). 3 random seeds.

In the ImageNet experiments, we evaluated INNAprop for rapid early training and optimal final test accuracy without tuning $(\gamma_0, \alpha, \beta)$. For ViT-B/32 with $\lambda = 0.1$, INNAprop achieved lower training loss and higher final test accuracy than AdamW (75.23 vs. 75.02).

Table 4.3: Top-1 and Top-5 accuracy (%) of ResNet-18, ResNet-50, and ViT-B/32 on ImageNet. Results are averaged from three runs for ResNets and one run for ViT-B/32. AdamW favored as in Table 4.1.

Model	**Optimizer**	**Top-1 accuracy**	**Top-5 accuracy**
Train from scratch on ImageNet			
ResNet18	AdamW	69.34	88.71
	INNAprop ($\alpha = 0.1, \beta = 0.9$)	**70.12**	**89.21**
ResNet50	AdamW	76.33	93.04
	INNAprop ($\alpha = 1.0, \beta = 1.0$)	**76.43**	**93.15**
ViT-B/32	AdamW	75.02	91.52
	INNAprop ($\alpha = 0.1, \beta = 0.9$)	**75.23**	**91.77**

Fintetuning VGG11 and ResNet18 models on Food101: We fine-tuned ResNet-18 and VGG-11 models on the Food101 dataset [24] for 20 epochs, using pre-trained models on ImageNet-1k. Since weight decay and learning rate values for AdamW were not found in the literature, we chose the default AdamW weight decay value, $\lambda = 0.01$. We used a cosine scheduler and tried one run for each initial

learning rate value in $\{10^{-5}, 5\times10^{-5}, 10^{-4}, 5\times10^{-4}, 10^{-3}\}$. The best result for AdamW was obtained for $\gamma_0 = 10^{-4}$, and we kept the same setting for INNAprop. See for this Figure 4.4, where INNAprop performs no worse than AdamW on three random seeds.

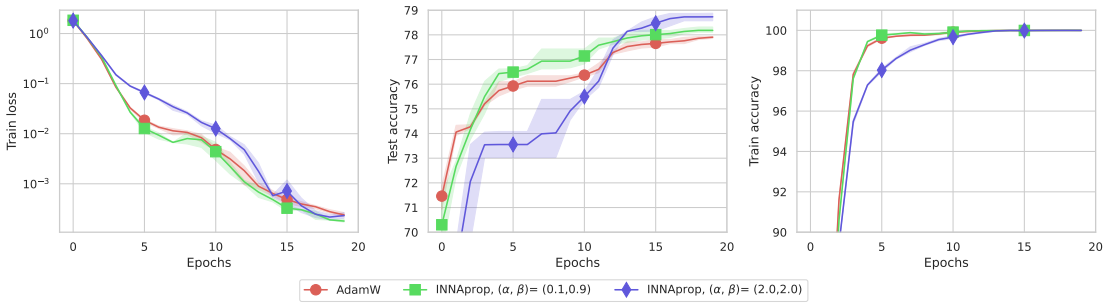


Figure 4.4: Finetuning a VGG11 on Food101. Left: train loss, middle: test accuracy (%), right: train accuracy (%). Qualitatively similar results for ResNet18 are in Figure 4.16 in Section 4.9. 3 random seeds.

Conclusion and recommendation for image classification: Tuning (α, β) significantly impacts training. Based on heatmaps in Section 4.3.1 and in Section 4.3.2, we recommend using $\alpha = 0.1$ and $\beta \in [0.5, 1.5]$ for shorter training (e.g., fine-tuning). For longer training, $\alpha, \beta \geq 1$ is preferable. In both cases, our algorithm either matches or outperforms AdamW.

4.3.3 Pre-training and fine-tuning GPT2

We present experimental results on LLMs using the hyperparameters selected as in Section 4.3.1.

Training GPT-2 from scratch: We train various GPT-2 transformer models from scratch [177] using the nanoGPT repository[2] on the OpenWebText dataset. For all models, gradients are clipped to a norm of 1, following [74], [169], [170]. We use AdamW with hyperparameters from the literature [74], [170], the standard configuration for LLM pre-training. The reported RMSprop parameter $\beta_2 = 0.95$ is different from AdamW's default (0.999), the weight decay is $\lambda = 0.1$ and γ_0 depending on the network size (see [74], [170]). For example, GPT-2 small works with an initial learning rate $\gamma_0 = 6\times10^{-4}$. For INNAprop, we keep the same values for λ and γ_0 as AdamW, and use the RMSprop parameter $\sigma = 0.99$ (corresponding to β_2 for AdamW), which provides the best results among values $\{0.9, 0.95, 0.99\}$ on GPT-2 mini. We use this setting for all our GPT-2 experiments with $(\alpha, \beta) = (0.1, 0.9)$. The results are in Figure 4.5. INNAprop leads to a faster decrease in validation loss during the early stages compared to the baseline for GPT-2 models of Mini (30M), Small (125M), and Medium (355M) sizes. Its performance could be further improved with more thorough tuning of hyperparameters $(\alpha, \beta, \sigma, \lambda)$. For GPT-2 small, we also include a comparison with Sophia-G, using the hyperparameters provided in the literature [3] [74].

[2] `https://github.com/karpathy/nanoGPT`
[3] `https://github.com/Liuhong99/Sophia`

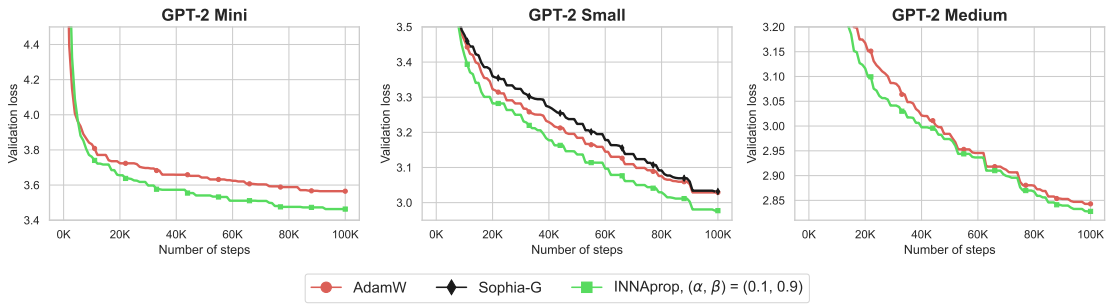


Figure 4.5: GPT-2 training from scratch on OpenWebText (Sophia-G unstable on mini and medium).

Fine-tune GPT-2 with LoRA: Using LoRA [171], we fine-tune the same GPT-2 models on the E2E dataset, consisting of roughly 42000 training 4600 validation, and 4600 test examples from the restauration domain. We compare AdamW and INNAprop for 5 epochs, as recommended in [171]. We use for both algorithms the same linear learning rate schedule, the recommended mini-batch size, and the RMSprop parameter ($\beta_2 = \sigma = 0.999$); these are listed in Table 11 in [171]. The results are displayed in Figure 4.6 and Table 4.4, where we see the perplexity mean result over 3 random seeds. INNAprop with $(\alpha, \beta) = (0.1, 0.9)$ consistently achieves lower perplexity loss compared to AdamW across all GPT-2 fine-tuning experiments.

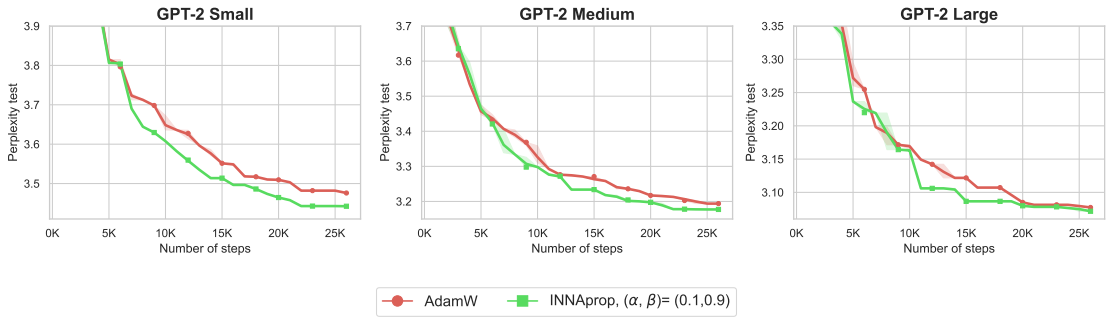


Figure 4.6: Perplexity test with GPT-2 E2E Dataset with LoRA finetuning on five epochs. Three random seeds.

We synthetize the performance of our algorithm on LLMs below and we emphasize the capabilities of INNAprop compared to AdamW in the context of early training where gains are considerable.

Table 4.4: Performance comparison for GPT-2 training from scratch on OpenWebText (validation loss) and fine-tuning with LoRA on the E2E dataset (perplexity).

Model	**AdamW best**	**INNAprop best**	**Steps to match AdamW**
GPT-2 Training from scratch (Validation loss)			
GPT-2 mini	3.57	**3.47**	51,000 (1.96× faster)
GPT-2 small	3.03	**2.98**	79,000 (1.26× faster)
GPT-2 medium	2.85	**2.82**	83,000 (1.2× faster)
GPT-2 with LoRA (Perplexity test)			
GPT-2 small	3.48	**3.44**	19,000 (1.31× faster)
GPT-2 medium	3.20	**3.17**	20,000 (1.25× faster)
GPT-2 large	3.09	**3.06**	20,000 (1.25× faster)

C Appendix of Chapter 4

4.4 A reminder on optimization algorithms

Considering the problem in Equation (4.1) and setting $\nabla \mathcal{J}(\theta_k) = g_k$, we outline several well-known update rule optimizers.

Table 4.5: Update rules considered for known optimizers. SGD is due to [26], Momentum to [62], Nesterov to [64], RMSprop + Momentum to [166], Adam to [65], NAdam to [158] and INNA to [12].

SGD(γ_k)	Momentum(γ_k, β_1)
$\theta_{k+1} = \theta_k - \gamma_k g_k$	$v_0 = 0$
	$v_{k+1} = \beta_1 v_k + (1-\beta_1) g_k$
Adam($\gamma_k, \beta_1, \beta_2, \epsilon$)	$\theta_{k+1} = \theta_k - \gamma_k v_{k+1}$
$m_0 = 0, v_0 = 0$	RMSprop + Momentum($\gamma_k, \beta_1, \beta_2, \epsilon$)
$m_{k+1} = \beta_1 m_k + (1-\beta_1) g_k$	$v_0 = 1, m_0 = 0$
$v_{k+1} = \beta_2 v_k + (1-\beta_2) g_k^2$	$v_{k+1} = \beta_2 v_k + (1-\beta_2) g_k^2$
$\theta_{k+1} = \theta_k - \gamma_k \frac{m_{k+1}}{\sqrt{v_{k+1}} + \epsilon}$	$m_{k+1} = \beta_1 m_k + \frac{g_k}{\sqrt{v_{k+1} + \epsilon}}$
	$\theta_{k+1} = \theta_k - \gamma_k m_{k+1}$
NAdam($\gamma_k, \psi, \beta_1, \beta_2, \epsilon$)	INNA(γ_k, α, β)
$m_0 = 0, v_0 = 0$	$\psi_0 = (1 - \alpha\beta)\theta_0$
$\mu_k = \beta_1 (1 - \frac{1}{2} 0.96^{k\psi})$	$\psi_{k+1} = \psi_k + \gamma_k \left((\frac{1}{\beta} - \alpha)\theta_k - \frac{1}{\beta}\psi_k \right)$
$m_{k+1} = \beta_1 m_k + (1-\beta_1) g_k$	$\theta_{k+1} = \theta_k + \gamma_k \left((\frac{1}{\beta} - \alpha)\theta_k - \frac{1}{\beta}\psi_k - \beta g_k \right)$
$v_{k+1} = \beta_2 v_k + (1-\beta_2) g_k^2$	
$\theta_{k+1} = \theta_k - \gamma_k \frac{\mu_{k+1} m_{k+1} + (1-\mu_k) g_k}{\sqrt{v_{k+1}} + \epsilon}$	

4.5 Derivation of INNAprop from DIN

We consider (4.9) which was a discretization of (4.6), namely:

$$v_{k+1} = \sigma_2 v_k + (1-\sigma_2) g_k^2 \tag{4.10}$$

$$\frac{\theta_{k+1} - 2\theta_k + \theta_{k-1}}{\gamma^2} + \alpha \frac{\theta_k - \theta_{k-1}}{\gamma} + \beta \frac{\frac{g_k}{\sqrt{v_{k+1}}+\epsilon} - \frac{g_{k-1}}{\sqrt{v_k}+\epsilon}}{\gamma} + \frac{g_{k-1}}{\sqrt{v_k}+\epsilon} = 0. \tag{4.11}$$

This gives

$$\begin{aligned} &\frac{1}{\gamma}\left(\left(\frac{\theta_{k+1}-\theta_k}{\gamma} + \beta \frac{g_k}{\sqrt{v_{k+1}}+\epsilon}\right) - \left(\frac{\theta_k - \theta_{k-1}}{\gamma} + \beta \frac{g_{k-1}}{\sqrt{v_k}+\epsilon}\right)\right) \\ = &-\alpha \frac{\theta_k - \theta_{k-1}}{\gamma} - \frac{g_{k-1}}{\sqrt{v_k}+\epsilon} \end{aligned}$$

and thus

$$\begin{aligned} &\frac{1}{\gamma}\left(\left(\frac{\theta_{k+1}-\theta_k}{\gamma} + \beta \frac{g_k}{\sqrt{v_{k+1}}+\epsilon}\right) - \left(\frac{\theta_k - \theta_{k-1}}{\gamma} + \beta \frac{g_{k-1}}{\sqrt{v_k}+\epsilon}\right)\right) \\ = &\left(\frac{1}{\beta} - \alpha\right) \frac{\theta_k - \theta_{k-1}}{\gamma} - \frac{1}{\beta}\left(\frac{\theta_k - \theta_{k-1}}{\gamma} + \beta \frac{g_{k-1}}{\sqrt{v_k}+\epsilon}\right). \end{aligned}$$

Multiplying by β, we obtain

$$\begin{aligned} &\frac{1}{\gamma}\left(\left(\beta\frac{\theta_{k+1}-\theta_k}{\gamma} + \beta^2 \frac{g_k}{\sqrt{v_{k+1}}+\epsilon}\right) - \left(\beta\frac{\theta_k - \theta_{k-1}}{\gamma} + \beta^2 \frac{g_{k-1}}{\sqrt{v_k}+\epsilon}\right)\right) \\ = &\left(1-\alpha\beta\right) \frac{\theta_k - \theta_{k-1}}{\gamma} - \frac{\theta_k - \theta_{k-1}}{\gamma} - \beta \frac{g_{k-1}}{\sqrt{v_k}+\epsilon} \end{aligned}$$

after rearranging all terms

$$\begin{aligned} &\frac{1}{\gamma}\Bigg(\left(\beta\frac{\theta_{k+1}-\theta_k}{\gamma} + \beta^2 \frac{g_k}{\sqrt{v_{k+1}}+\epsilon} + (\alpha\beta - 1)\theta_k\right) \\ &\qquad - \left(\beta\frac{\theta_k - \theta_{k-1}}{\gamma} + \beta^2 \frac{g_{k-1}}{\sqrt{v_k}+\epsilon} + (\alpha\beta-1)\theta_{k-1}\right)\Bigg) \\ = &-\frac{\theta_k - \theta_{k-1}}{\gamma} - \beta \frac{g_{k-1}}{\sqrt{v_k}+\epsilon}. \end{aligned}$$

Setting $\psi_{k-1} = -\beta\frac{\theta_k - \theta_{k-1}}{\gamma} - \beta^2 \frac{g_{k-1}}{\sqrt{v_k}+\epsilon} - (\alpha\beta-1)\theta_{k-1}$, we obtain the recursion

$$v_{k+1} = \sigma_2 v_k + (1-\sigma_2) g_k^2 \tag{4.12}$$

$$\frac{\psi_k - \psi_{k-1}}{\gamma} = -\frac{\psi_{k-1}}{\beta} - \left(\alpha - \frac{1}{\beta}\right)\theta_{k-1} \tag{4.13}$$

$$\frac{\theta_{k+1} - \theta_k}{\gamma} = \frac{-1}{\beta}\psi_k - \beta \frac{g_k}{\sqrt{v_{k+1}}+\epsilon} - \left(\alpha - \frac{1}{\beta}\right)\theta_k \tag{4.14}$$

We can also rewrite the above as follows:

$$
\begin{aligned}
v_{k+1} &= \sigma_2 v_k + (1-\sigma_2) g_k^2 \\
\psi_{k+1} &= \psi_k \left(1 - \frac{\gamma}{\beta}\right) + \gamma \left(\frac{1}{\beta} - \alpha\right) \theta_k, \\
\theta_{k+1} &= \theta_k \left(1 + \gamma \left(\frac{1}{\beta} - \alpha\right)\right) - \frac{\gamma}{\beta} \psi_k - \gamma\beta \frac{g_k}{\sqrt{v_{k+1}} + \epsilon}.
\end{aligned}
$$

We can save a memory slot by avoiding the storage of ψ_k:

$$
\psi_{k+1} = \psi_k \left(1 - \frac{\gamma}{\beta}\right) + \gamma \left(\frac{1}{\beta} - \alpha\right) \theta_k, \tag{4.15}
$$

$$
\begin{aligned}
\Leftrightarrow \quad \psi_k &= \frac{\beta}{\beta - \gamma} \left(\psi_{k+1} - \gamma \left(\frac{1}{\beta} - \alpha\right) \theta_k\right) = \frac{\beta}{\beta - \gamma} \psi_{k+1} - \frac{\beta}{\beta - \gamma} \gamma \left(\frac{1}{\beta} - \alpha\right) \theta_k \\
\theta_{k+1} &= \theta_k \left(1 + \gamma \left(\frac{1}{\beta} - \alpha\right)\right) - \frac{\gamma}{\beta} \psi_k - \gamma\beta \frac{g_k}{\sqrt{v_{k+1}} + \epsilon} \\
&= \theta_k + \gamma \left(\frac{1}{\beta} - \alpha\right) \theta_k - \frac{\gamma}{\beta - \gamma} \psi_{k+1} + \frac{\gamma}{\beta - \gamma} \gamma \left(\frac{1}{\beta} - \alpha\right) \theta_k - \gamma\beta \frac{g_k}{\sqrt{v_{k+1}} + \epsilon} \\
&= \theta_k + \left(1 + \frac{\gamma}{\beta - \gamma}\right) \gamma \left(\frac{1}{\beta} - \alpha\right) \theta_k - \frac{\gamma}{\beta - \gamma} \psi_{k+1} - \gamma\beta \frac{g_k}{\sqrt{v_{k+1}} + \epsilon} \\
&= \theta_k + \left(\frac{\beta}{\beta - \gamma}\right) \gamma \left(\frac{1}{\beta} - \alpha\right) \theta_k - \frac{\gamma}{\beta - \gamma} \psi_{k+1} - \gamma\beta \frac{g_k}{\sqrt{v_{k+1}} + \epsilon} \\
&= \theta_k + \left(\frac{\gamma(1 - \beta\alpha)}{\beta - \gamma}\right) \theta_k - \frac{\gamma}{\beta - \gamma} \psi_{k+1} - \gamma\beta \frac{g_k}{\sqrt{v_{k+1}} + \epsilon} \\
&= \left(1 + \frac{\gamma(1 - \beta\alpha)}{\beta - \gamma}\right) \theta_k - \frac{\gamma}{\beta - \gamma} \psi_{k+1} - \gamma\beta \frac{g_k}{\sqrt{v_{k+1}} + \epsilon}
\end{aligned} \tag{4.16}
$$

Finally, we merely need to use 3 memory slots having the underlying dimension size p:

$$
\begin{aligned}
v_{k+1} &= \sigma_2 v_k + (1-\sigma_2) g_k^2 \\
\psi_{k+1} &= \psi_k \left(1 - \frac{\gamma}{\beta}\right) + \gamma \left(\frac{1}{\beta} - \alpha\right) \theta_k, \\
\theta_{k+1} &= \left(1 + \frac{\gamma(1 - \beta\alpha)}{\beta - \gamma}\right) \theta_k - \frac{\gamma}{\beta - \gamma} \psi_{k+1} - \gamma\beta \frac{g_k}{\sqrt{v_{k+1}} + \epsilon}
\end{aligned}
$$

4.5.1 Equivalence between a special case of INNAprop and Adam without momentum

In this section, we demonstrate that INNAprop with $\alpha = 1$ and $\beta = 1$ is equivalent to Adam [65] without momentum ($\beta_1 = 0$). To illustrate this, we analyze the update rules of both algorithms. We assume that the RMSprop parameter β_2 (for Adam) and σ (for INNAprop) are equal. Starting with INNAprop, we initialize $\psi_0 = (1 - \alpha\beta)\theta_0$. For $\alpha = 1$ and $\beta = 1$, this simplifies to $\psi_0 = 0$. The update for

Algorithm 4: INNAprop

Objective function: $\mathcal{J}(\theta)$ for $\theta \in \mathbb{R}^p$.
Constant step-size: $\gamma > 0$
Hyper-parameters: $\sigma \in [0, 1]$, $\alpha \geq 0$, $\beta > \gamma$, $\epsilon = 10^{-8}$.
Initialization: θ_0, $v_0 = 0$, $\psi_0 = (1 - \alpha\beta)\theta_0$.
for $k = 1$ **to** K **do**
  $\boldsymbol{g}_k = \nabla \mathcal{J}(\boldsymbol{\theta}_k)$
  $\boldsymbol{v}_{k+1} \leftarrow \sigma \boldsymbol{v}_k + (1 - \sigma)\boldsymbol{g}_k^2$
  $\boldsymbol{\psi}_{k+1} \leftarrow \left(1 - \frac{\gamma}{\beta}\right) \boldsymbol{\psi}_k + \gamma \left(\frac{1}{\beta} - \alpha\right) \boldsymbol{\theta}_k$
  $\boldsymbol{\theta}_{k+1} \leftarrow \left(1 + \frac{\gamma(1-\alpha\beta)}{\beta-\gamma}\right) \boldsymbol{\theta}_k - \frac{\gamma}{\beta-\gamma} \boldsymbol{\psi}_{k+1} - \gamma\beta \frac{\boldsymbol{g}_k}{\sqrt{v_{k+1}}+\epsilon}$
return $\boldsymbol{\theta}_{K+1}$

ψ becomes:

$$\psi_{k+1} = \left(1 - \frac{\gamma}{\beta}\right) \psi_k + \gamma \left(\frac{1}{\beta} - \alpha\right) \theta_k = (1 - \gamma)\psi_k$$

Given that $\psi_0 = 0$, it follows that $\psi_k = 0$ for all k. The parameter update rule for INNAprop is:

$$\theta_{k+1} = \left(1 + \frac{\gamma(1 - \alpha\beta)}{\beta - \gamma}\right) \theta_k - \frac{\gamma}{\beta - \gamma} \psi_{k+1} - \gamma\beta \frac{g_k}{\sqrt{v_{k+1}} + \epsilon}$$

Replacing $\alpha = 1$, $\beta = 1$, and $\psi_k = 0$, we get:

$$\theta_{k+1} = \theta_k - \gamma \frac{g_k}{\sqrt{v_{k+1}} + \epsilon}$$

Here, g_k is the gradient, and v_{k+1} is the exponential moving average of the squared gradients:

$$v_{k+1} = \sigma v_k + (1 - \sigma) g_k^2$$

The Adam optimizer uses two moving averages, m_k (momentum term) and v_k (squared gradients):

$$m_k = \beta_1 m_{k-1} + (1 - \beta_1) g_k$$
$$v_k = \sigma v_{k-1} + (1 - \sigma) g_k^2$$

Setting $\beta_1 = 0$, the momentum term m_k simplifies to $m_k = g_k$. The update rule becomes:

$$\theta_{k+1} = \theta_k - \gamma \frac{g_k}{\sqrt{v_k} + \epsilon}$$

This matches the form of Adam's update rule without the momentum term, confirming that INNAprop with $\alpha = 1$ and $\beta = 1$ is equivalent to Adam with $\beta_1 = 0$.

Algorithm 5: INNAprop with $(\alpha, \beta) = (1, 1)$

Objective function: $\mathcal{J}(\theta)$ for $\theta \in \mathbb{R}^p$.
Constant step-size: $\gamma > 0$
Hyper-parameters: $\sigma \in [0, 1]$, $\alpha \geq 0$, $\beta > \gamma$, $\epsilon = 10^{-8}$.
Initialization: time step $k \leftarrow 0$, parameter vector θ_0, $v_0 = 0$.
repeat
  $k \leftarrow k + 1$
  $\boldsymbol{g}_k = \nabla \mathcal{J}(\boldsymbol{\theta}_k)$
  $\boldsymbol{v}_{k+1} \leftarrow \sigma \boldsymbol{v}_k + (1 - \sigma)\boldsymbol{g}_k^2$
  $\hat{\boldsymbol{v}}_{k+1} \leftarrow \boldsymbol{v}_{k+1}/(1 - \sigma^k)$
  $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \gamma_k \left(\boldsymbol{g}_k/(\sqrt{\hat{\boldsymbol{v}}_{k+1}} + \epsilon\right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{\theta}_{k+1}$

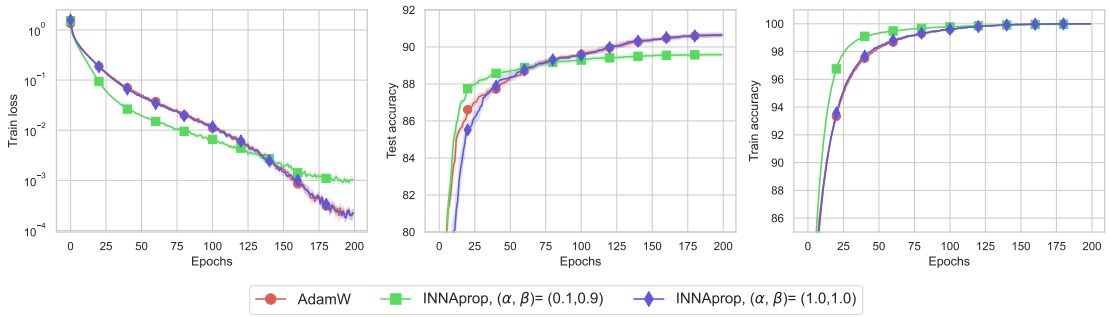


Figure 4.7: Training VGG11 on CIFAR10. Left: train loss, middle: test accuracy (%), right: train accuracy (%), with 8 random seeds.

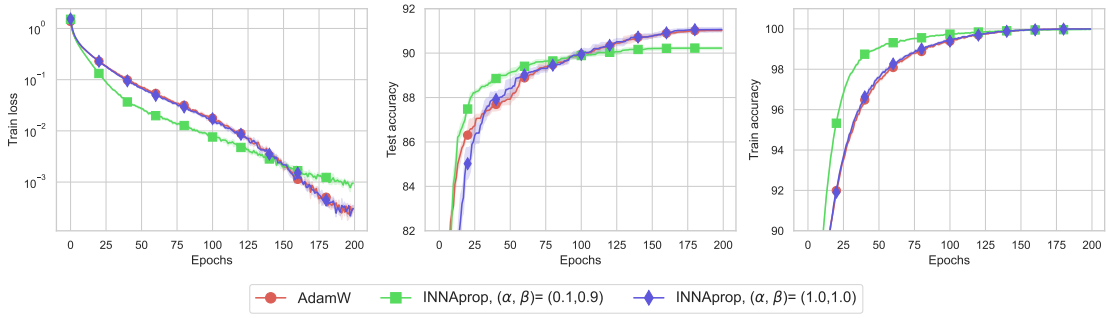


Figure 4.8: Training ResNet18 on CIFAR10. Left: train loss, middle: test accuracy (%), right: train accuracy (%), with 8 random seeds.

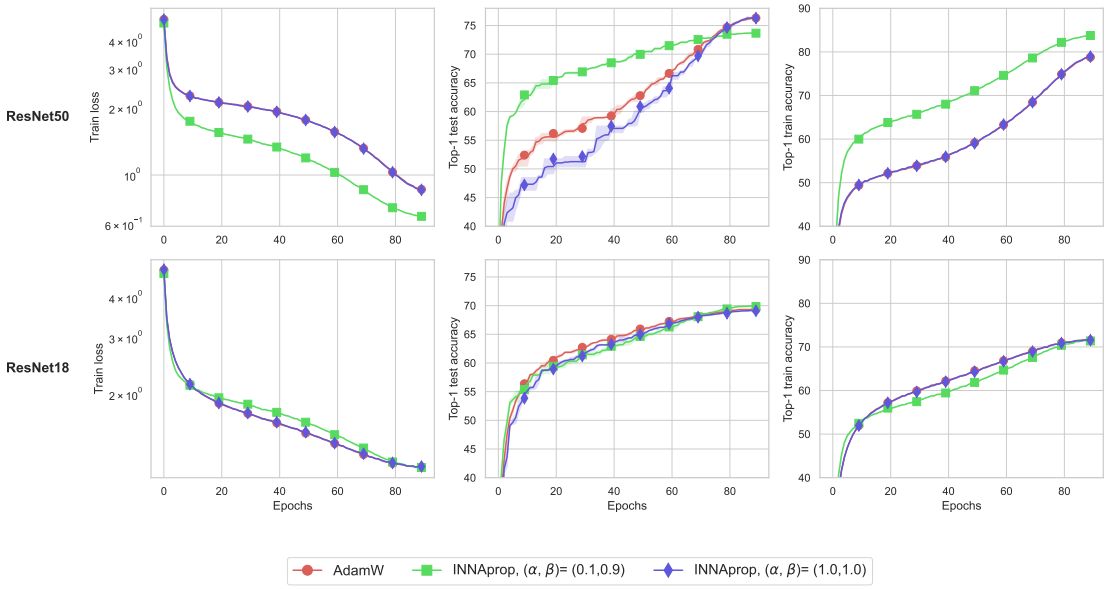


Figure 4.9: Training a ResNet50 (top) and ResNet18 (bottom) on ImageNet. Left: train loss, middle: Top-1 test accuracy (%), right: Top-1 train accuracy (%). 3 random seeds.

4.6 Alternative discretizations

4.6.1 An alternative derivation of INNAprop

As mentioned in Remark 14, we can obtain INNAprop easily from INNA [12]. The algorithm INNA writes (see Table 4.5):

$$\psi_{k+1} = \psi_k + \gamma\left((\frac{1}{\beta} - \alpha)\theta_k - \frac{1}{\beta}\psi_k\right)$$
$$\theta_{k+1} = \theta_k + \gamma\left((\frac{1}{\beta} - \alpha)\theta_k - \frac{1}{\beta}\psi_k - \beta g_k\right)$$

Rearranging the terms and saving a memory slot — use ψ_{k+1} in the second equation instead of ψ_k, (see Equation (4.16) for details)— yields

$$\psi_{k+1} = \psi_k\left(1 - \frac{\gamma}{\beta}\right) + \gamma\left(\frac{1}{\beta} - \alpha\right)\theta_k$$
$$\theta_{k+1} = \left(1 + \frac{\gamma(1-\beta\alpha)}{\beta - \gamma}\right)\theta_k - \frac{\gamma}{\beta-\gamma}\psi_{k+1} - \gamma\beta g_k$$

Now, use the RMSprop proxy directly within INNA. Using the usual RMSprop constants $\sigma \in [0,1]$ and $\epsilon > 0$, we obtain:

$$v_{k+1} = \sigma v_k + (1-\sigma)g_k^2$$
$$\psi_{k+1} = \psi_k\left(1 - \frac{\gamma}{\beta}\right) + \gamma\left(\frac{1}{\beta} - \alpha\right)\theta_k$$
$$\theta_{k+1} = \left(1 + \frac{\gamma(1-\beta\alpha)}{\beta - \gamma}\right)\theta_k - \frac{\gamma}{\beta-\gamma}\psi_{k+1} - \gamma\beta\frac{g_k}{\sqrt{v_{k+1}} + \epsilon}$$

This is INNAprop and the derivation is much more direct, although less illustrative of the geometric features.

4.6.2 A variant of INNAprop with momentum

The algorithm. We follow the rationale behind the algorithm RMSprop with momentum [166]. We therefore start with Equation (4.9) using the RMSprop proxy for the gradient:

$$\begin{aligned}
&v_{k+1} = \sigma v_k + (1-\sigma) g_k^2 \\
&\frac{\theta_{k+1} - 2\theta_k + \theta_{k-1}}{\gamma} + \alpha \frac{\theta_k - \theta_{k-1}}{\gamma} + \beta \frac{\frac{g_k}{\sqrt{v_{k+1}}+\epsilon} - \frac{g_{k-1}}{\sqrt{v_k}+\epsilon}}{\gamma} + \frac{g_{k-1}}{\sqrt{v_k}+\epsilon} = 0.
\end{aligned}$$

Rearranging terms, we have

$$\begin{aligned}
&v_{k+1} = \sigma v_k + (1-\sigma) g_k^2 \\
&\theta_{k+1} = \theta_k + (1-\alpha\gamma)(\theta_k - \theta_{k-1}) - \beta\gamma \left(\frac{g_k}{\sqrt{v_{k+1}}+\epsilon} - \frac{g_{k-1}}{\sqrt{v_k}+\epsilon} \right) - \gamma^2 \frac{g_{k-1}}{\sqrt{v_k}+\epsilon}
\end{aligned}$$

Let us introduce a momentum variable $m_k = \theta_{k-1} - \theta_k$ to obtain:

$$v_{k+1} = \sigma v_k + (1-\sigma) g_k^2 \tag{4.17}$$

$$m_{k+1} = (1-\alpha\gamma) m_k + \gamma^2 \frac{g_{k-1}}{\sqrt{v_k}+\epsilon} + \beta\gamma \left(\frac{g_k}{\sqrt{v_{k+1}}+\epsilon} - \frac{g_{k-1}}{\sqrt{v_k}+\epsilon} \right) \tag{4.18}$$

$$\theta_{k+1} = \theta_k - m_{k+1} \tag{4.19}$$

As previously need now to optimize the dynamics in terms of storage. For this we rewrite Equation (4.18) as

$$m_{k+1} = a m_k + b g_k - c g_{k-1}. \tag{4.20}$$

where $a = (1-\alpha\gamma)$, $b = \beta\gamma$ and $c = \gamma(\beta - \gamma)$. Writing $\tilde{m}_k = m_k - \frac{c}{a} g_{k-1}$, we have

$$\begin{aligned}
\tilde{m}_{k+1} &= m_{k+1} - \frac{c}{a} g_k \\
&= a m_k + b g_k - c g_{k-1} - \frac{c}{a} g_k \\
&= a \left(m_k - \frac{c}{a} g_{k-1} \right) + \left(b - \frac{c}{a} \right) g_k \\
&= a \tilde{m}_k + \left(b - \frac{c}{a} \right) g_k.
\end{aligned}$$

Therefore, using this identity, we may rewrite the following

$$\begin{aligned}
m_{k+1} &= a m_k + b g_k - c g_{k-1}, \\
\theta_{k+1} &= \theta_k - m_{k+1}
\end{aligned}$$

as

$$\begin{aligned}
\tilde{m}_{k+1} &= a \tilde{m}_k + \left(b - \frac{c}{a} \right) g_k, \\
\theta_{k+1} &= \theta_k - \tilde{m}_{k+1} - \frac{c}{a} g_k.
\end{aligned}$$

Recalling that $a = (1 - \alpha\gamma)$, $b = \beta\gamma$ and $c = \gamma(\beta - \gamma)$. Finally, we get the following recursion which is an alternative way to integrate RMSprop to INNA:

$$v_{k+1} = \sigma v_k + (1 - \sigma) g_k^2 \tag{4.21}$$

$$\tilde{m}_{k+1} = (1 - \alpha\gamma)\tilde{m}_k + \gamma^2 \left(\frac{1 - \alpha\beta}{1 - \alpha\gamma} \right) \frac{g_k}{\sqrt{v_{k+1}} + \epsilon} \tag{4.22}$$

$$\theta_{k+1} = \theta_k - \tilde{m}_{k+1} - \frac{\gamma(\beta - \gamma)}{1 - \alpha\gamma} \frac{g_k}{\sqrt{v_{k+1}} + \epsilon} \tag{4.23}$$

but as shown below through numerical experiments, the factor γ^2 is poorly scaled for 32 bits or lower machine precision.

Numerical experiments. Using CIFAR-10 dataset, we train a VGG11 network with the momentum version of INNAprop with the hyperparameters $(\alpha, \beta) = (0.1, 0.9)$ above. We used a cosine annealing scheduler with $\gamma_0 = 10^{-3}$ and no weight decay. As seen in Figure 4.10, the training loss stops decreasing between the 125th and 150th epochs. Upon closely examining the algorithm in this regime, we observe that at the end of training, γ_k^2 falls below the numerical precision, resulting in unstable behavior in Equation (4.22).

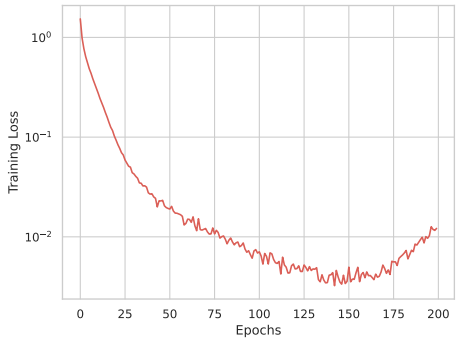


Figure 4.10: The version of INNA with momentum of Section 4.6.2 is an unstable method.

4.6.3 An approach à la Adam

In this section, we mimic the process for deriving Adam from the heavy ball with a RMSprop proxy, see, e.g., [65], [178], by simply replacing the heavy ball by DIN[4]. We call this optimizer DINAdam.

From (4.6), we infer the discretization:

$$\frac{\theta_{k+1} - 2\theta_k + \theta_{k-1}}{\gamma^2} + \alpha \frac{\theta_{k+1} - \theta_k}{\gamma} + \beta \frac{g_k - g_{k-1}}{\gamma} + g_k = 0. \tag{4.24}$$

Rearranging terms, we have

$$\theta_{k+1} = \theta_k - \frac{\gamma^2}{1 + \alpha\gamma} g_k + \frac{1}{1 + \alpha\gamma}(\theta_k - \theta_{k-1}) - \frac{\beta\gamma}{(1 + \alpha\gamma)}(g_k - g_{k-1}) \tag{4.25}$$

[4]Note that DIN with $\beta = 0$ boils down to the heavy ball method.

By introducing the new variable $m_k = (\theta_{k-1} - \theta_k)/\eta$ and setting $\eta > 0$, we can rewrite equation (4.25) as:

$$m_{k+1} = \frac{1}{(1+\alpha\gamma)} m_k + \frac{\gamma^2}{(1+\alpha\gamma)\eta} g_k + \frac{\beta\gamma}{(1+\alpha\gamma)\eta}(g_k - g_{k-1}) \tag{4.26}$$

$$\theta_{k+1} = \theta_k - \eta m_{k+1} \tag{4.27}$$

To follow the Adam spirit, we set $\sigma_1 = \frac{1}{(1+\alpha\gamma)}$ and $(1-\sigma_1) = \frac{\gamma^2}{(1+\alpha\gamma)\eta}$. Solving for γ, we get

$$\frac{\alpha\gamma}{1+\alpha\gamma} = \frac{\gamma^2}{(1+\alpha\gamma)\eta} \Rightarrow \gamma = \frac{\eta}{\alpha}$$

Then, we find the following recursion:

$$m_{k+1} = \sigma_1 m_k + (1-\sigma_1) g_k + \beta\alpha\sigma_1(g_k - g_{k-1}) \tag{4.28}$$

$$\theta_{k+1} = \theta_k - \eta m_{k+1} \tag{4.29}$$

From Equation (4.28), we make a change of variable $\tilde{m}_k = m_k - \alpha\beta g_{k-1}$ to save a memory cell.

$$\tilde{m}_{k+1} = \sigma_1 \tilde{m}_k + (1 - \sigma_1 + \beta\alpha\sigma_1 - \beta\alpha) g_k \tag{4.30}$$

$$\theta_{k+1} = \theta_k - \eta(\tilde{m}_{k+1} - \alpha\beta g_k) \tag{4.31}$$

Using the usual RMSprop constants $\sigma_2 \in [0,1]$ and $\epsilon > 0$, we obtain:

$$v_{k+1} = \sigma_2 v_k + (1-\sigma_2) g_k^2 \tag{4.32}$$

$$\tilde{m}_{k+1} = \sigma_1 \tilde{m}_k + (1 - \sigma_1 + \beta\alpha\sigma_1 - \beta\alpha) g_k \tag{4.33}$$

$$\theta_{k+1} = \theta_k - \eta \frac{\tilde{m}_{k+1} - \alpha\beta g_k}{\sqrt{v_{k+1}} + \epsilon} \tag{4.34}$$

Algorithm 6: DINAdam

Objective function: $\mathcal{J}(\theta)$ for $\theta \in \mathbb{R}^p$.
Constant step-size: $\gamma > 0$
Hyper-parameters: $(\sigma_1, \sigma_2) \in [0,1]^2$, $\alpha, \beta > 0$, $\epsilon = 10^{-8}$.
Initialization: θ_0, $v_0 = 0$, $\tilde{m}_0 = 0$.
repeat
  $\boldsymbol{g}_k = \nabla\mathcal{J}(\boldsymbol{\theta}_k)$
  $\boldsymbol{v}_{k+1} \leftarrow \sigma_2 \boldsymbol{v}_k + (1-\sigma_2)\boldsymbol{g}_k^2$
  $\tilde{\boldsymbol{m}}_{k+1} \leftarrow \sigma_1 \tilde{\boldsymbol{m}}_k + (1 - \sigma_1 + \beta\alpha\sigma_1 - \beta\alpha)\boldsymbol{g}_k$
  $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \gamma \frac{\tilde{\boldsymbol{m}}_{k+1} - \alpha\beta\boldsymbol{g}_k}{\sqrt{\boldsymbol{v}_{k+1}} + \epsilon}$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{\theta}_k$

Remark 16 The way RMSprop is added in INNAprop and DINAdam is different. In INNAprop, RMSprop is incorporated directly during the discretization process of Equation (4.9) for all gradients. However, in DINAdam, RMSprop is added only at the last step, as shown in Equation (4.32), and only on the gradient in the θ_{k+1} update. This is how RMSprop was combined with heavy ball to obtain Adam.

Remark 17 After setting $\alpha = 1$ and $\beta = 0$, we obtain Adam update rules. If $\beta \neq 0$, DINAdam is very close to NAdam algorithm. Hence, we did not investigate this algorithm numerically.

4.7 Scheduler procedures

Cosine annealing [60]. Let γ_k represent the learning rate at iteration k, $T_{\max}$ be the maximum number of iterations (or epochs), and $\gamma_{\min}$ be the minimum learning rate (default value is 0). The learning rate γ_k at iteration k is given by:

$$\gamma_k = \gamma_{\min} + \frac{1}{2}(\gamma_0 - \gamma_{\min})\left(1 + \cos\left(\frac{k}{T_{\max}}\pi\right)\right)$$

This scheduler was employed in all image classification experiments except for ViT.

Cosine annealing with linear warmup [179]. Let γ_k represent the learning rate at iteration k, $\gamma_{\min}$ the minimum learning rate, γ_0 the initial learning rate, T_{warmup} the number of iterations for the warmup phase, and T_{decay} the iteration number after which the learning rate decays to $\gamma_{\min}$. The learning rate is defined as follows:

$$\gamma_k = \begin{cases} \gamma_0 \cdot \frac{k}{T_{\text{warmup}}}, & \text{if } k < T_{\text{warmup}} \\ \gamma_{\min} + \frac{1}{2}(\gamma_0 - \gamma_{\min})\left(1 + \cos\left(\pi \cdot \frac{k - T_{\text{warmup}}}{T_{\text{decay}} - T_{\text{warmup}}}\right)\right), & \text{if } T_{\text{warmup}} \leq k \leq T_{\text{decay}} \\ \gamma_{\min}, & \text{if } k > T_{\text{decay}} \end{cases}$$

This scheduler was applied in experiments involving training GPT-2 from scratch and for ViT.

Linear schedule with linear warmup [171]. Let γ_k represent the learning rate at iteration k and $T_{\max}$ be the maximum number of iterations, T_{warmup} be the number of warmup steps, and $\gamma_{\min}$ be the minimum learning rate after warmup (default value is typically set to the initial learning rate, γ_0). The learning rate γ_k at iteration k is given by:

$$\gamma_k = \begin{cases} \gamma_0 \cdot \frac{k}{T_{\text{warmup}}} & \text{if } k < T_{\text{warmup}}, \\ \gamma_0 \cdot \left(1 - \frac{k - T_{\text{warmup}}}{T_{\max} - T_{\text{warmup}}}\right) & \text{otherwise.} \end{cases}$$

This scheduler was used for fine-tuning GPT-2 with LoRA.

4.8 Choosing hyperparameters α and β for INNAprop

4.8.1 Comparison with AdamW

For VGG and ResNet training on CIFAR10, the literature suggest using initial learning rate $\gamma_0 = 10^{-3}$ with a learning rate schedule [66], [169], [176], [180]. Our

experiment fix a cosine scheduler where $T_{\max} = 200$ and $\gamma_{\min} = 0$ as it achieves a strong baseline for AdamW [60], [169]. We set weight decay $\lambda = 0.1$. Then, we tune the initial learning rate γ_0 among $\{10^{-4}, 5 \times 10^{-4}, 10^{-3}, 5 \times 10^{-3}, 10^{-2}\}$. In Figure 4.11, we report the performance in terms of training loss and test accuracy for AdamW. These results confirm the usage of $\gamma_0 = 10^{-3}$.

(a) Performance rankings with VGG11.

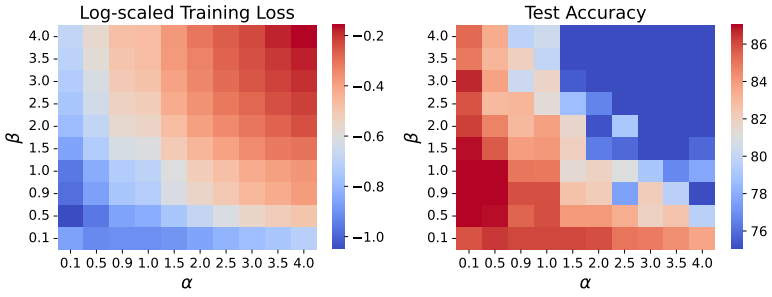
γ_0	**Train loss**	**Test accuracy (%)**
10^{-3}	0.00041	91.02
5×10^{-3}	0.00047	90.86
5×10^{-4}	0.00048	90.79
10^{-2}	0.00057	90.41
10^{-4}	0.00081	88.49

(b) Performance rankings with ResNet18.

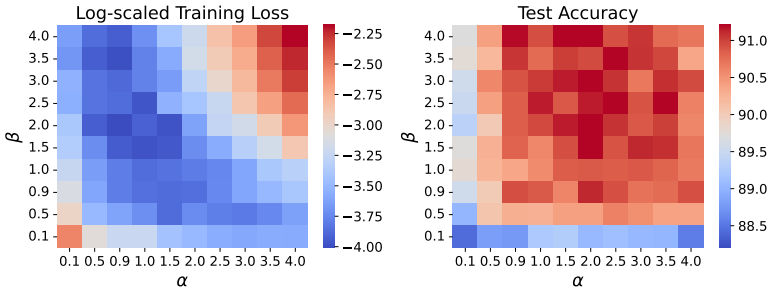
γ_0	**Train loss**	**Test accuracy (%)**
2×10^{-3}	0.00040	92.10
5×10^{-3}	0.00049	91.84
5×10^{-4}	0.00094	92.32
10^{-2}	0.00057	90.41
10^{-4}	0.0018	87.85

Figure 4.11: Comparative performance of the training loss and test accuracy according to γ_0. We trained VGG11 and ResNet18 models on CIFAR10 for 200 epochs.

4.8.2 Heatmap for preliminary tuning of α and β

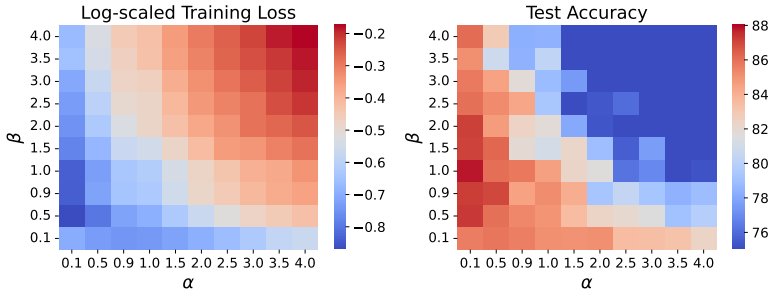


(a) 20 epochs

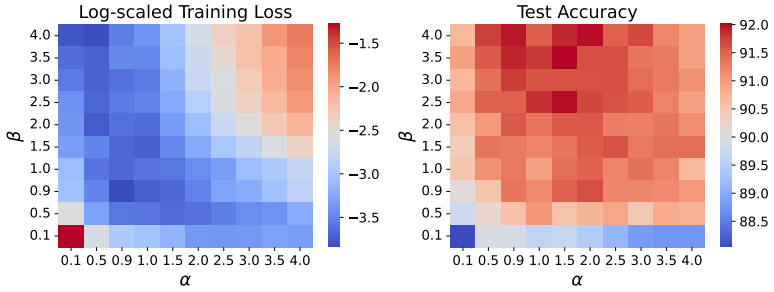


(b) 200 epochs

Figure 4.12: Log-scale training loss and test accuracies for (α, β) hyperparameters with VGG11 on CIFAR10 at different epochs. Optimal learning rate $\gamma_0 = 10^{-3}$, weight decay $\lambda = 0$.

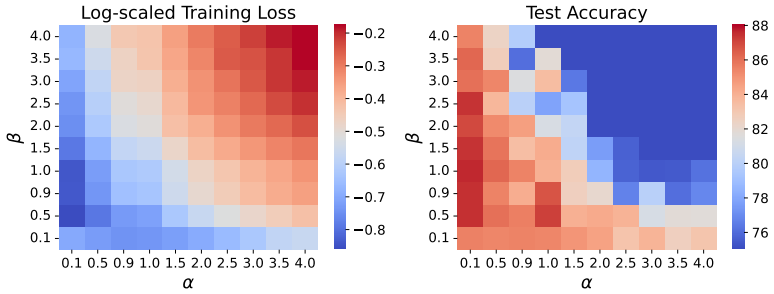


(a) 20 epochs

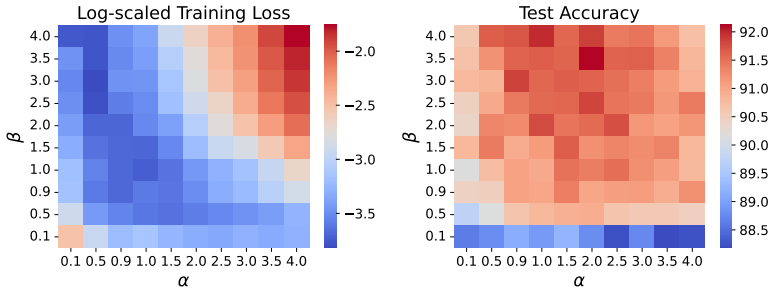


(b) 200 epochs

Figure 4.13: Log-scale training loss and test accuracies for (α, β) hyperparameters with ResNet18 on CIFAR10 at different epochs. Optimal learning rate $\gamma_0 = 10^{-3}$, weight decay $\lambda = 0.01$.



(a) 20 epochs



(b) 200 epochs

Figure 4.14: Log-scale training loss and test accuracies for (α, β) hyperparameters with ResNet18 on CIFAR10 at different epochs. Optimal learning rate $\gamma_0 = 10^{-3}$, weight decay $\lambda = 0$.

4.9 Additional experiments

4.9.1 CIFAR10 experiments

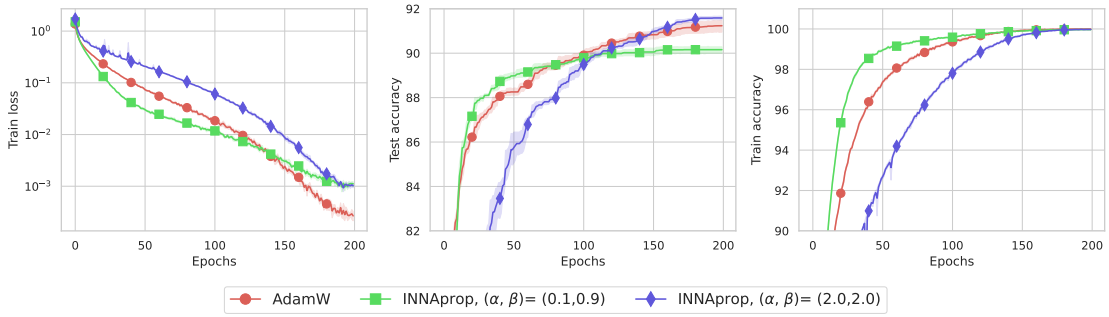


Figure 4.15: Training ResNet18 on CIFAR10. Left: train loss, middle: test accuracy (%), right: train accuracy (%), with 8 random seeds.

4.9.2 Food101 experiments

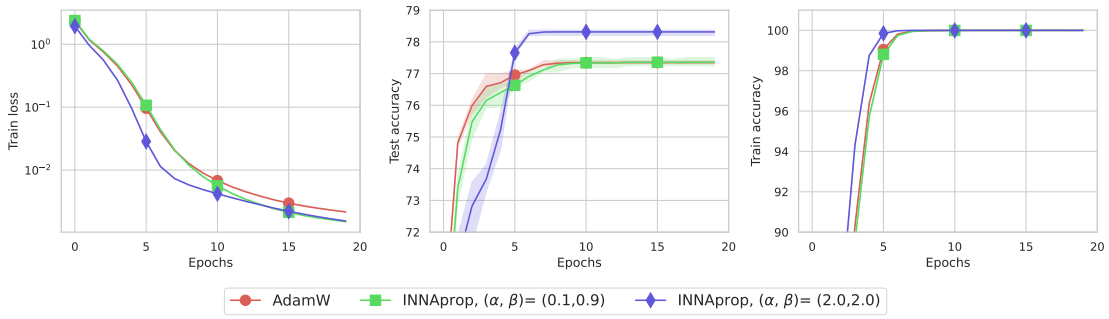


Figure 4.16: Finetuning a ResNet18 on Food101, same as Figure 4.4 for ResNet18. Left: train loss, middle: test accuracy (%), right: train accuracy (%), with 3 random seeds.

4.9.3 ImageNet

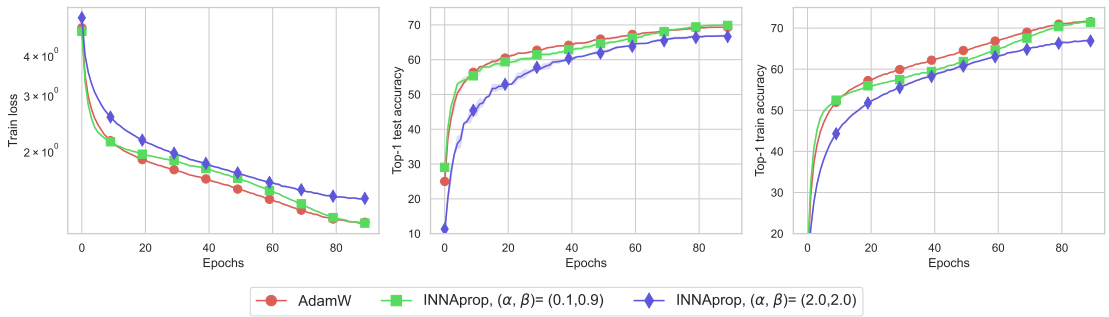


Figure 4.17: Training ResNet18 on ImageNet. Left: train loss, middle: test accuracy (%), right: train accuracy (%), with 3 random seeds.

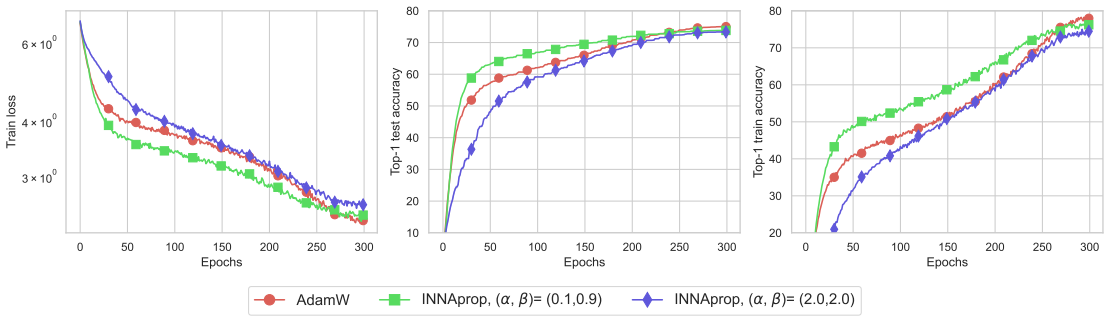


Figure 4.18: Fast training ViT/B-32 on ImageNet with weight decay $\lambda = 0.01$ for INNAprop $(\alpha, \beta) = (0.1, 0.9)$. Left: train loss, middle: test accuracy (%), right: train accuracy (%), with 3 random seeds.

4.9.4 Comparision with INNA

We evaluate INNA on GPT-2 Mini and compare it to INNAprop and AdamW. Following [12], we used the recommended hyperparameters $(\alpha, \beta) = (0.5, 0.1)$ and tested learning rates $\{1e-4, 1e-3, 1e-2, 1e-1\}$, selecting $\gamma_0 = 0.1$ as the best. Figure 4.19 shows that INNAprop and AdamW outperform INNA in both convergence speed and final validation loss.

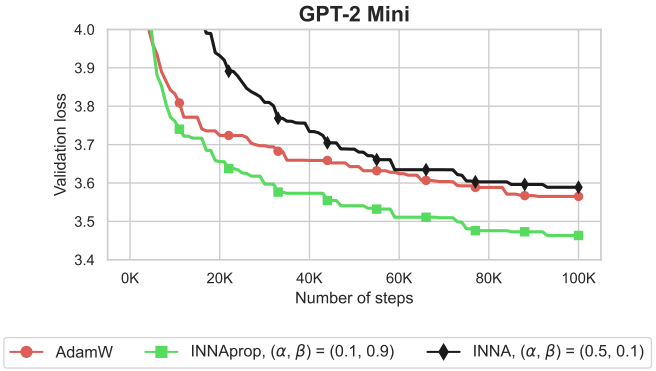


Figure 4.19: Validation loss comparison during GPT-2 mini training from scratch on the OpenWebText dataset.

4.10 Experimental Setup

4.10.1 CIFAR-10

We used custom training code based on the PyTorch tutorial code for this problem. Following standard data-augmentation practices, we applied random horizontal flips and random offset cropping down to 32x32, using reflection padding of 4 pixels. Input pixel data was normalized by centering around 0.5.

Hyper-parameter	Value
Architecture	VGG11 and ResNet18
Epochs	200
GPUs	1×V100
Batch size per GPU	256
Baseline LR	0.001
Seeds	8 runs

Hyper-parameter	Value
Baseline schedule	cosine
Weight decay λ	0.01
β_1, β_2 (for AdamW)	0.9, 0.999
σ (for INNAprop)	0.999

4.10.2 Food101

We used the pre-trained models available on PyTorch for VGG11 and ResNet18.[5].

Hyper-parameter	Value
Architecture	VGG11 and ResNet18
Epochs	200
GPUs	1×V100
Batch size per GPU	256
Baseline LR	0.001
Seeds	3 runs

Hyper-parameter	Value
Baseline schedule	cosine
Weight decay λ	0.01
β_1, β_2 (for AdamW)	0.9, 0.999
σ (for INNAprop)	0.999

4.10.3 ImageNet

We used the same code-base as for our CIFAR-10 experiments, and applied the same preprocessing procedure. The data-augmentations consisted of PyTorch's RandomResizedCrop, cropping to 224x224 followed by random horizontal flips. Test images used a fixed resize to 256x256 followed by a center crop to 224x224.

ResNet18

Hyper-parameter	Value
Architecture	ResNet18
Epochs	90
GPUs	4×V100
Batch size per GPU	64
Baseline LR	0.001
Seeds	3 runs

Hyper-parameter	Value
Baseline schedule	cosine
Weight decay λ	0.01
β_1, β_2 (for AdamW)	0.9, 0.999
σ (for INNAprop)	0.999

[5] `https://pytorch.org/vision/stable/models.html`

ResNet50

Hyper-parameter	Value
Architecture	ResNet18
Epochs	90
GPUs	4×V100
Batch size per GPU	64
Baseline LR	0.001
Mixed precision	True
Seeds	3 runs

Hyper-parameter	Value
Baseline schedule	cosine
Weight decay λ	0.1
β_1, β_2 (for AdamW)	0.9, 0.999
σ (for INNAprop)	0.999

ViT/B-32

Hyper-parameter	Value
Architecture	ViT/B-32
Epochs	300
GPUs	8×A100
Batch size per GPU	128
Baseline LR	0.001
Seeds	5000

Hyper-parameter	Value
Baseline schedule	cosine
Warmup	linear for 30 epochs
Weight decay λ	0.1
β_1, β_2 (for AdamW)	0.9, 0.999
σ (for INNAprop)	0.999

4.10.4 GPT2 from scratch

We followed the NanoGPT codebase [6] and we refer to [170] as closely as possible, matching the default batch-size and schedule.

Hyper-parameter	Value
Architecture	GPT-2
Batch size per gpu	12
Max Iters	100000
GPUs	4×A100
Dropout	0.0
Baseline LR	refer to [170]
Warmup Steps	500

Hyper-parameter	Value
Seeds	5000
Weight decay λ	0.1
β_1, β_2 (for AdamW)	0.9, 0.95
σ (for INNAprop)	0.99
Gradient Clipping	1.0
Float16	True

4.10.5 GPT-2 with LoRA

We followed the LoRA codebase [7] and we refer to [171] as closely as possible, matching the default batch-size, training length, and schedule. We train all of our GPT-2 models using AdamW [32] and INNAprop on E2E dataset with a linear learning rate schedule for 5 epochs. We report the mean result over 3 random seeds; the result for each run is taken from the best epoch.

[6] `https://github.com/karpathy/nanoGPT`
[7] `https://github.com/microsoft/LoRA`

Hyper-parameter	**Value**
Architecture	GPT-2
Batch size per gpu	8
Epochs	5
GPUs	1×A100
Dropout	0.1
Baseline LR	0.0002
Warmup steps	500

Hyper-parameter	**Value**
Seeds	3 runs
Weight decay λ	0.01
β_1, β_2 (for AdamW)	0.9, 0.98
σ (for INNAprop)	0.98
Learning Rate Schedule	Linear
LoRA α	32

Chapter 5

Conclusion and perspectives

This thesis focused on the theoretical and practical aspects of automatic differentiation and optimization algorithms for nonsmooth neural networks. It addressed critical issues related to certifiability and robustness while proposing tools and methods applicable to various machine learning contexts.

In Chapter 2, we extended the complexity analysis of automatic differentiation to nonsmooth programs. By introducing a model based on conservative gradients, we established principles such as the "cheap conservative gradient principle" and demonstrated that these gradients exhibit computational properties akin to classical derivatives. These results provide a rigorous framework for understanding the computational costs of nonsmooth programs.

In Chapter 3, we investigated the numerical reliability of automatic differentiation in nonsmooth architectures by analyzing numerical bifurcation zones in the context of MaxPool networks. Our experiments highlighted the significant role of numerical precision (16, 32, or 64 bits) in model stability and proposed strategies to manage errors introduced by floating-point arithmetic.

In Chapter 4, we introduced INNAprop, a novel optimization algorithm that combines adaptive scaling (via RMSprop) with second-order information derived from dynamic systems. This algorithm was validated on large-scale models, demonstrating its effectiveness across diverse contexts while maintaining computational complexity comparable to classical first-order algorithms.

This work is motivated by the need for AI certifiability, focusing on providing theoretical and numerical guarantees for widely adopted practices in the machine learning community. A natural extension is to develop and refine algorithms tailored to nonsmooth and nonconvex settings. The results presented here open the way for several promising directions for future research:

- Explore nonsmooth optimization in new domains like reinforcement learning, optimal transport, and scientific computing to uncover fresh theoretical and practical insights.
- Expand robustness guarantees to address adversarial scenarios, hardware limitations (e.g., quantization), and extreme noise conditions, enhancing real-world applicability.
- Develop effective scheduling strategies for the hyperparameters α and β in INNAprop.

In conclusion, this thesis contributed to establishing a theoretical foundation for learning algorithms based on nonsmooth automatic differentiation while providing practical solutions to computational and numerical challenges in deep learning.

Bibliography

[1] L. Bottou and O. Bousquet, "The tradeoffs of large scale learning", *Advances in neural information processing systems*, vol. 20, 2007.

[2] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey", *Journal of Marchine Learning Research*, vol. 18, pp. 1–43, 2018.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[4] B. Speelpenning, *Compiling fast partial derivatives of functions given by algorithms*. University of Illinois at Urbana-Champaign, 1980.

[5] M. Abadi, P. Barham, J. Chen, *et al.*, "Tensorflow: A system for large-scale machine learning", in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: `https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf`.

[6] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[7] J. Bradbury, R. Frostig, P. Hawkins, *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.2.5, 2018. [Online]. Available: `http://github.com/google/jax`.

[8] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[9] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and J. Z. Kolter, "Differentiable convex optimization layers", in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[10] J. Bolte and E. Pauwels, "Conservative set valued fields, automatic differentiation, stochastic gradient methods and deep learning", *Mathematical Programming*, pp. 1–33, 2020.

[11] J. Bolte and E. Pauwels, "A mathematical model for automatic differentiation in machine learning", in *Conference on Neural Information Processing Systems*, 2020.

[12] C. Castera, J. Bolte, C. Févotte, and E. Pauwels, "An inertial newton algorithm for deep learning", *The Journal of Machine Learning Research*, vol. 22, no. 1, pp. 5977–6007, 2021.

[13] T. Tieleman, G. Hinton, *et al.*, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude", *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[14] D. O. Hebb, *The organization of behavior: A neuropsychological theory*. Psychology press, 2005.

[15] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.", *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[16] Y. LeCun, B. Boser, J. S. Denker, *et al.*, "Backpropagation applied to handwritten zip code recognition", *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[17] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks", *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[18] Y. Bengio, "Gradient-based optimization of hyperparameters", *Neural computation*, vol. 12, no. 8, pp. 1889–1900, 2000.

[19] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks", *Advances in neural information processing systems*, vol. 19, 2006.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", pp. 1097–1105, 2012.

[21] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space", *arXiv preprint arXiv:1301.3781*, 2013.

[22] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database", pp. 248–255, 2009.

[23] A. Krizhevsky and G. Hinton, "The cifar-10 dataset", 2010.

[24] L. Bossard, M. Guillaumin, and L. Van Gool, "Food-101–mining discriminative components with random forests", in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part VI 13*, Springer, 2014, pp. 446–461.

[25] G. Van Rossum and F. L. Drake Jr, *Python tutorial*, 1995.

[26] H. Robbins and S. Monro, "A stochastic approximation method", *The annals of mathematical statistics*, pp. 400–407, 1951.

[27] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning", *Siam Review*, vol. 60, no. 2, pp. 223–311, 2018.

[28] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for modern deep learning research", in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, 2020, pp. 13 693–13 696.

[29] D. Patterson, J. Gonzalez, Q. Le, *et al.*, "Carbon emissions and large neural network training", *arXiv preprint arXiv:2104.10350*, 2021.

[30] G. Team, R. Anil, S. Borgeaud, *et al.*, "Gemini: A family of highly capable multimodal models", *arXiv preprint arXiv:2312.11805*, 2023.

[31] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization.", *Journal of machine learning research*, vol. 12, no. 7, 2011.

[32] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization", *arXiv preprint arXiv:1711.05101*, 2017.

[33] X. Chen, C. Liang, D. Huang, *et al.*, "Symbolic discovery of optimization algorithms", *arXiv preprint arXiv:2302.06675*, 2023.

[34] E. Moulines and F. Bach, "Non-asymptotic analysis of stochastic approximation algorithms for machine learning", *Advances in neural information processing systems*, vol. 24, 2011.

[35] S. Ghadimi and G. Lan, "Stochastic first-and zeroth-order methods for nonconvex stochastic programming", *SIAM journal on optimization*, vol. 23, no. 4, pp. 2341–2368, 2013.

[36] X. Li and F. Orabona, "On the convergence of stochastic gradient descent with adaptive stepsizes", in *The 22nd international conference on artificial intelligence and statistics*, PMLR, 2019, pp. 983–992.

[37] D. Davis, D. Drusvyatskiy, S. Kakade, and J. D. Lee, "Stochastic subgradient method converges on tame functions", *Foundations of computational mathematics*, vol. 20, no. 1, pp. 119–154, 2020.

[38] R. Sun and Z.-Q. Luo, "Guaranteed matrix completion via non-convex factorization", *IEEE Transactions on Information Theory*, vol. 62, no. 11, pp. 6535–6579, 2016.

[39] C. Jin, R. Ge, P. Netrapalli, S. M. Kakade, and M. I. Jordan, "How to escape saddle points efficiently", in *International Conference on Machine Learning (ICML)*, 2017, pp. 1724–1732.

[40] Y. Nesterov, *Introductory lectures on convex optimization: A basic course*. Springer Science & Business Media, 2004.

[41] D. Bertoin, J. Bolte, S. Gerchinovitz, and E. Pauwels, "Numerical influence of relu'(0) on backpropagation", *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[42] R. Boustany, "On the numerical reliability of nonsmooth autodiff: A maxpool case study", *Transactions on Machine Learning Research*, 2024.

[43] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge university press, 2004.

[44] R. Rockafellar, "Convex analysis", *Princeton Math. Series*, vol. 28, 1970.

[45] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", pp. 770–778, 2016.

[46] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need", *Advances in neural information processing systems*, vol. 30, 2017.

[47] G. Cybenko, "Approximation by superpositions of a sigmoidal function", *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[48] J. B. Simon, D. Karkada, N. Ghosh, and M. Belkin, "More is better in modern machine learning: When infinite overparameterization is optimal and overfitting is obligatory", *arXiv preprint arXiv:2311.14646*, 2023.

[49] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[50] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks", pp. 818–833, 2014.

[51] A. Dosovitskiy, L. Beyer, A. Kolesnikov, *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale", *arXiv preprint arXiv:2010.11929*, 2020.

[52] A. Q. Jiang, A. Sablayrolles, A. Mensch, *et al.*, "Mistral 7b", *arXiv preprint arXiv:2310.06825*, 2023.

[53] B. Widrow and M. A. Lehr, "30 years of adaptive neural networks: Perceptron, madaline, and backpropagation", *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1415–1442, 1990.

[54] W. Baur and V. Strassen, "The complexity of partial derivatives.", *Theoretical Computer Science, 22:317–330*, 1983.

[55] J. Bolte, R. Boustany, E. Pauwels, and B. Pesquet-Popescu, "On the complexity of nonsmooth automatic differentiation", in *The Eleventh International Conference on Learning Representations*, 2022.

[56] "Ieee standard for floating-point arithmetic", *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. DOI: 10.1109/IEEESTD.2019.8766229.

[57] P. Micikevicius, S. Narang, J. Alben, *et al.*, "Mixed precision training", *arXiv preprint arXiv:1710.03740*, 2017.

[58] M. Ott, S. Edunov, D. Grangier, and M. Auli, "Scaling neural machine translation", *arXiv preprint arXiv:1806.00187*, 2018.

[59] L. Armijo, "Minimization of functions having lipschitz continuous first partial derivatives", *Pacific Journal of mathematics*, vol. 16, no. 1, pp. 1–3, 1966.

[60] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts", *arXiv preprint arXiv:1608.03983*, 2016.

[61] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", *Advances in neural information processing systems*, vol. 27, 2014.

[62] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods", *Ussr computational mathematics and mathematical physics*, vol. 4, no. 5, pp. 1–17, 1964.

[63] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning", in *International conference on machine learning*, PMLR, 2013, pp. 1139–1147.

[64] Y. E. Nesterov, "A method of solving a convex programming problem with convergence rate $O(1/k^2)$", in *Doklady Akademii Nauk*, Russian Academy of Sciences, vol. 269, 1983, pp. 543–547.

[65] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014.

[66] J. Zhuang, T. Tang, Y. Ding, *et al.*, "Adabelief optimizer: Adapting stepsizes by the belief in observed gradients", *Advances in neural information processing systems*, vol. 33, pp. 18 795–18 806, 2020.

[67] M. D. Zeiler, "Adadelta: An adaptive learning rate method", *arXiv preprint arXiv:1212.5701*, 2012.

[68] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond", *arXiv preprint arXiv:1904.09237*, 2019.

[69] M. Zhang, J. Lucas, J. Ba, and G. E. Hinton, "Lookahead optimizer: K steps forward, 1 step back", *Advances in neural information processing systems*, vol. 32, 2019.

[70] J. Martens *et al.*, "Deep learning via hessian-free optimization.", in *Icml*, vol. 27, 2010, pp. 735–742.

[71] R. H. Byrd, G. M. Chin, W. Neveitt, and J. Nocedal, "On the use of stochastic hessian information in optimization methods for machine learning", *SIAM Journal on Optimization*, vol. 21, no. 3, pp. 977–995, 2011.

[72] C. Boyer and A. Godichon-Baggioni, "On the asymptotic rate of convergence of stochastic newton algorithms and their weighted averaged versions", *Computational Optimization and Applications*, vol. 84, no. 3, pp. 921–972, 2023.

[73] J. Martens and R. Grosse, "Optimizing neural networks with kronecker-factored approximate curvature", in *International conference on machine learning*, PMLR, 2015, pp. 2408–2417.

[74] H. Liu, Z. Li, D. Hall, P. Liang, and T. Ma, "Sophia: A scalable stochastic second-order optimizer for language model pre-training", *arXiv preprint arXiv:2305.14342*, 2023.

[75] J. Bolte, R. Boustany, E. Pauwels, and A. Purica, *A second-order-like optimizer with adaptive gradient scaling for deep learning*, 2024. arXiv: `2410.05871 [cs.LG]`. [Online]. Available: `https://arxiv.org/abs/2410.05871`.

[76] L. M. Beda, L. N. Korolev, N. V. Sukkikh, and T. S. Frolova, "Programs for automatic differentiation for the machine BESM", Institute for Precise Mechanics and Computation Techniques, Academy of Science, Moscow, USSR, Tech. Rep., 1959.

[77] R. E. Wengert, "A simple automatic derivative evaluation program", *Communications of the ACM*, vol. 7, no. 8, pp. 463–464, 1964.

[78] A. Griewank *et al.*, "On automatic differentiation", *Mathematical Programming: recent developments and applications*, vol. 6, no. 6, pp. 83–107, 1989.

[79] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-propagating Errors", *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. DOI: 10.1038/323533a0. [Online]. Available: http://www.nature.com/articles/323533a0.

[80] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[81] A. Griewank and C. Faure, "Piggyback differentiation and optimization", in *Large-scale PDE-constrained optimization*, Springer, 2003, pp. 148–164.

[82] A. Agrawal, S. Barratt, S. Boyd, E. Busseti, and W. M. Moursi, "Differentiating through a cone program", *J. Appl. Numer. Optim*, vol. 1, no. 2, pp. 107–115, 2019.

[83] S. Bai, J. Z. Kolter, and V. Koltun, "Deep equilibrium models", *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[84] J. Bolte, T. Le, E. Pauwels, and T. Silveti-Falls, "Nonsmooth implicit differentiation for machine-learning and optimization", *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[85] M. Blondel, Q. Berthet, M. Cuturi, *et al.*, "Efficient and modular implicit differentiation", *arXiv preprint arXiv:2105.15183*, 2021.

[86] P. E. Farrell, D. A. Ham, S. W. Funke, and M. E. Rognes, "Automated derivation of the adjoint of high-level transient finite element programs", *SIAM Journal on Scientific Computing*, vol. 35, no. 4, pp. C369–C393, 2013.

[87] B. A. Pearlmutter, "Gradient calculations for dynamic recurrent neural networks: A survey", *IEEE Transactions on Neural networks*, vol. 6, no. 5, pp. 1212–1228, 1995.

[88] R.-E. Plessix, "A review of the adjoint-state method for computing the gradient of a functional with geophysical applications", *Geophysical Journal International*, vol. 167, no. 2, pp. 495–503, 2006.

[89] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations", *Advances in neural information processing systems*, vol. 31, 2018.

[90] S. Mehmood and P. Ochs, "Automatic differentiation of some first-order methods in parametric optimization", in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2020, pp. 1584–1594.

[91] Q. Bertrand, Q. Klopfenstein, M. Blondel, S. Vaiter, A. Gramfort, and J. Salmon, "Implicit differentiation of lasso-type models for hyperparameter optimization", in *International Conference on Machine Learning*, PMLR, 2020, pp. 810–821.

[92] J. Lorraine, P. Vicol, and D. Duvenaud, "Optimizing millions of hyperparameters by implicit differentiation", in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2020, pp. 1540–1552.

[93] S. Gratton, D. Titley-Peloquin, P. Toint, and J. T. Ilunga, "Differentiating the method of conjugate gradients", *SIAM Journal on Matrix Analysis and Applications*, vol. 35, no. 1, pp. 110–126, 2014. DOI: 10.1137/120889848. eprint: https://doi.org/10.1137/120889848. [Online]. Available: https://doi.org/10.1137/120889848.

[94] P. Wolfe, "Checking the calculation of gradients", *ACM Transactions on Mathematical Software (TOMS)*, vol. 8, no. 4, pp. 337–343, 1982.

[95] P. I. Barton, K. A. Khan, P. Stechlinski, and H. A. Watson, "Computationally relevant generalized derivatives: Theory, evaluation and applications", *Optimization Methods and Software*, vol. 33, no. 4-6, pp. 1030–1072, 2018. DOI: 10.1080/10556788.2017.1374385. eprint: https://doi.org/10.1080/10556788.2017.1374385. [Online]. Available: https://doi.org/10.1080/10556788.2017.1374385.

[96] A. Lewis and T. Tian, "The structure of conservative gradient fields", *arXiv preprint arXiv:2101.00699*, 2021.

[97] D. Davis and D. Drusvyatskiy, "Conservative and semismooth derivatives are equivalent for semialgebraic maps", *arXiv preprint arXiv:2102.08484*, 2021.

[98] A. Griewank, "On stable piecewise linearization and generalized algorithmic differentiation", *Optimization Methods and Software*, vol. 28, Jul. 2013. DOI: 10.1080/10556788.2013.796683.

[99] A. Griewank and A. Rojas, "Treating artificial neural net training as a nonsmooth global optimization problem.", *In International Conference on Machine Learning, Optimization, and Data Science (pp. 759-770). Springer, Cham.*, 2019.

[100] A. Griewank and A. Walther, "Beyond the oracle: Opportunities of piecewise differentiation.", *In Numerical Nonsmooth Optimization (pp. 331-361). Springer, Cham.*, 2020.

[101] S. Scholtes, *Introduction to piecewise differentiable equations*. Springer Science & Business Media, 2012.

[102] S. M. Kakade and J. D. Lee, "Provably correct automatic sub-differentiation for qualified programs", in *Advances in Neural Information Processing Systems*, vol. 31, Curran Associates, Inc., 2018.

[103] K. A. Khan and P. I. Barton, "Evaluating an element of the clarke generalized jacobian of a piecewise differentiable function", in *Recent Advances in Algorithmic Differentiation*, Springer, 2012, pp. 115–125.

[104] K. A. Khan and P. I. Barton, "Evaluating an element of the clarke generalized jacobian of a composite piecewise differentiable function", *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 4, pp. 1–28, 2013.

[105] K. A. Khan and P. I. Barton, "A vector forward mode of automatic differentiation for generalized derivative evaluation", *Optimization Methods and Software*, vol. 30, no. 6, pp. 1185–1212, 2015.

[106] Y. Nesterov, "Lexicographic differentiation of nonsmooth functions", *Mathematical programming*, vol. 104, no. 2, pp. 669–700, 2005.

[107] F. H. Clarke, *Optimization and nonsmooth analysis*. SIAM, 1983.

[108] R. T. Rockafellar and R. J. B. Wets, *Variational Analysis*. 1998.

[109] R. T. Rockafellar and R. J.-B. Wets, *Variational analysis*. Springer Science & Business Media, 2009, vol. 317.

[110] J. J. Moreau, "Fonctionnelles sous-différentiables", *Comptes rendus hebdomadaires des séances de l'Académie des sciences*, vol. 257, pp. 4117–4119, 1963.

[111] V. Strassen *et al.*, "Gaussian elimination is not optimal", *Numerische mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[112] S. Robinson, "Toward an optimal algorithm for matrix multiplication", *SIAM news*, vol. 38, no. 9, pp. 1–3, 2005.

[113] V. V. Williams, "Multiplying matrices faster than coppersmith-winograd", in *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, 2012, pp. 887–898.

[114] F. Le Gall, "Powers of tensors and fast matrix multiplication", in *Proceedings of the 39th international symposium on symbolic and algebraic computation*, 2014, pp. 296–303.

[115] J. Alman and V. V. Williams, "A refined laser method and faster matrix multiplication", in *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SIAM, 2021, pp. 522–539.

[116] A. Shapiro, "On concepts of directional differentiability", *Journal of optimization theory and applications*, vol. 66, no. 3, pp. 477–487, 1990.

[117] K. A. Khan, "Branch-locking ad techniques for nonsmooth composite functions and nonsmooth implicit functions", *Optimization Methods and Software*, vol. 33, no. 4-6, pp. 1127–1155, 2018.

[118] M. Coste, *An introduction to o-minimal geometry*. Istituti editoriali e poligrafici internazionali Pisa, 2000.

[119] M. Coste, *An introduction to semialgebraic geometry*, 2000.

[120] J. Bochnak, M. Coste, and M.-F. Roy, *Real algebraic geometry*. Springer Science & Business Media, 2013, vol. 36.

[121] R. Arora, A. Basu, P. Mianjy, and A. Mukherjee, "Understanding deep neural networks with rectified linear units", in *International Conference on Learning Representations, Conference Track Proceedings*, 2018.

[122] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. Sohl-Dickstein, "On the expressive power of deep neural networks", in *international conference on machine learning*, PMLR, 2017, pp. 2847–2854.

[123] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[124] D. Davis, D. Drusvyatskiy, S. Kakade, and J. D. Lee, "Stochastic subgradient method converges on tame functions.", *Foundations of Computational Mathematics.*, 2018.

[125] K. Yamaguchi, K. Sakamoto, T. Akabane, and Y. Fujimoto, "A neural network for speaker-independent isolated word recognition.", in *ICSLP*, 1990.

[126] W. Lee, S. Park, and A. Aiken, *On the correctness of automatic differentiation for neural networks with machine-representable parameters*, 2023. arXiv: 2301.13370 [cs.LG].

[127] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors", *Nature*, vol. 323, pp. 533–536, 1986.

[128] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[129] J. Bolte, T. Le, E. Pauwels, and A. Silveti-Falls, "Nonsmooth implicit differentiation for machine learning and optimization", *CoRR*, vol. abs/2106.04350, 2021. arXiv: 2106.04350. [Online]. Available: https://arxiv.org/abs/2106.04350.

[130] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", pp. 1026–1034, 2015.

[131] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", *arXiv preprint arXiv:1409.1556*, 2014.

[132] D. Bertoin, J. Bolte, S. Gerchinovitz, and E. Pauwels, *Erratum: Numerical influence of relu'(0) on backpropagation*, 2023. arXiv: 2106.12915 [cs.LG].

[133] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning", vol. 2011, p. 5, 2011.

[134] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks", *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[135] X. Jia, S. Song, W. He, *et al.*, "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes", *arXiv preprint arXiv:1807.11205*, 2018.

[136] T. Susnjak, T. R. McIntosh, A. L. Barczak, *et al.*, "Over the edge of chaos? excess complexity as a roadblock to artificial general intelligence", *arXiv preprint arXiv:2407.03652*, 2024.

[137] G. Varoquaux, A. S. Luccioni, and M. Whittaker, "Hype, sustainability, and the price of the bigger-is-better paradigm in ai", *arXiv preprint arXiv:2409.14160*, 2024.

[138] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green ai", *Communications of the ACM*, vol. 63, no. 12, pp. 54–63, 2020.

[139] L. F. W. Anthony, B. Kanding, and R. Selvan, "Carbontracker: Tracking and predicting the carbon footprint of training deep learning models", *arXiv preprint arXiv:2007.03051*, 2020.

[140] J. Achiam, S. Adler, S. Agarwal, *et al.*, "Gpt-4 technical report", *arXiv preprint arXiv:2303.08774*, 2023.

[141] A. Chowdhery, S. Narang, J. Devlin, *et al.*, "Palm: Scaling language modeling with pathways", *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[142] L. Prechelt, "Early stopping-but when?", in *Neural Networks: Tricks of the trade*, Springer, 2002, pp. 55–69.

[143] Y. Bai, E. Yang, B. Han, *et al.*, "Understanding and improving early stopping for learning with noisy labels", *Advances in Neural Information Processing Systems*, vol. 34, pp. 24 392–24 403, 2021.

[144] L. Ljung, "Analysis of recursive stochastic algorithms", *IEEE transactions on automatic control*, vol. 22, no. 4, pp. 551–575, 1977.

[145] J. Harold, G. Kushner, and G. Yin, "Stochastic approximation and recursive algorithm and applications", *Application of Mathematics*, vol. 35, no. 10, 1997.

[146] M. Benaim, "Dynamics of stochastic approximation algorithms", in *Seminaire de probabilites XXXIII*, Springer, 2006, pp. 1–68.

[147] V. S. Borkar and V. S. Borkar, *Stochastic approximation: a dynamical systems viewpoint*. Springer, 2008, vol. 9.

[148] H. Attouch, J. Peypouquet, and P. Redont, "Fast convex optimization via inertial dynamics with hessian driven damping", *Journal of Differential Equations*, vol. 261, no. 10, pp. 5734–5783, 2016.

[149] J.-F. Aujol, C. Dossal, and A. Rondepierre, "Optimal convergence rates for nesterov acceleration", *SIAM Journal on Optimization*, vol. 29, no. 4, pp. 3131–3153, 2019.

[150] A. Barakat and P. Bianchi, "Convergence and dynamical behavior of the adam algorithm for nonconvex stochastic optimization", *SIAM Journal on Optimization*, vol. 31, no. 1, pp. 244–274, 2021.

[151] L. Chen, B. Liu, K. Liang, and Q. Liu, "Lion secretly solves constrained optimization: As lyapunov predicts", *arXiv preprint arXiv:2310.05898*, 2023.

[152] F. Alvarez, H. Attouch, J. Bolte, and P. Redont, "A second-order gradient-like dissipative dynamical system with hessian-driven damping.: Application to optimization and mechanics", *Journal de mathématiques pures et appliquées*, vol. 81, no. 8, pp. 747–779, 2002.

[153] L. Chen and H. Luo, "First order optimization methods based on hessian-driven nesterov accelerated gradient flow", *arXiv preprint arXiv:1912.09276*, 2019.

[154] H. Attouch, Z. Chbani, J. Fadili, and H. Riahi, "First-order optimization algorithms via inertial systems with hessian driven damping", *Mathematical Programming*, pp. 1–43, 2022.

[155] N. Qian, "On the momentum term in gradient descent learning algorithms", *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[156] W. Su, S. Boyd, and E. J. Candes, "A differential equation for modeling nesterov's accelerated gradient method: Theory and insights", *Journal of Machine Learning Research*, vol. 17, no. 153, pp. 1–43, 2016.

[157] H. Attouch, Z. Chbani, and H. Riahi, "Rate of convergence of the nesterov accelerated gradient method in the subcritical case $\alpha \leq 3$", *ESAIM: Control, Optimisation and Calculus of Variations*, vol. 25, p. 2, 2019.

[158] T. Dozat, "Incorporating nesterov momentum into adam", 2016.

[159] N. Shazeer and M. Stern, "Adafactor: Adaptive learning rates with sublinear memory cost", in *International Conference on Machine Learning*, PMLR, 2018, pp. 4596–4604.

[160] Y. You, J. Li, S. Reddi, *et al.*, "Large batch optimization for deep learning: Training bert in 76 minutes", *arXiv preprint arXiv:1904.00962*, 2019.

[161] M. Pagliardini, P. Ablin, and D. Grangier, "The ademamix optimizer: Better, faster, older", *arXiv preprint arXiv:2409.03137*, 2024.

[162] A. Defazio, H. Mehta, K. Mishchenko, A. Khaled, A. Cutkosky, *et al.*, "The road less scheduled", *arXiv preprint arXiv:2405.15682*, 2024.

[163] V. Gupta, T. Koren, and Y. Singer, "Shampoo: Preconditioned stochastic tensor optimization", in *International Conference on Machine Learning*, PMLR, 2018, pp. 1842–1850.

[164] M. Jahani, S. Rusakov, Z. Shi, P. Richtárik, M. W. Mahoney, and M. Takáč, "Doubly adaptive scaled algorithm for machine learning using second-order information", *arXiv preprint arXiv:2109.05198*, 2021.

[165] X. Qian, R. Islamov, M. Safaryan, and P. Richtárik, "Basis matters: Better communication-efficient second order methods for federated learning", *arXiv preprint arXiv:2111.01847*, 2021.

[166] A. Graves, "Generating sequences with recurrent neural networks", *arXiv preprint arXiv:1308.0850*, 2013.

[167] J. Chen, D. Zhou, Y. Tang, Z. Yang, Y. Cao, and Q. Gu, "Closing the generalization gap of adaptive gradient methods in training deep neural networks", *arXiv preprint arXiv:1806.06763*, 2018.

[168] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training data-efficient image transformers & distillation through attention", in *International conference on machine learning*, PMLR, 2021, pp. 10 347–10 357.

[169] K. Mishchenko and A. Defazio, "Prodigy: An expeditiously adaptive parameter-free learner", *arXiv preprint arXiv:2306.06101*, 2023.

[170] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners", *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[171] E. J. Hu, Y. Shen, P. Wallis, *et al.*, "Lora: Low-rank adaptation of large language models", *arXiv preprint arXiv:2106.09685*, 2021.

[172] P. T. Sivaprasad, F. Mai, T. Vogels, M. Jaggi, and F. Fleuret, "Optimizer benchmarking needs to account for hyperparameter tuning", in *International conference on machine learning*, PMLR, 2020, pp. 9036–9045.

[173] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, *Optuna: A next-generation hyperparameter optimization framework*, 2019. arXiv: 1907.10902 [cs.LG].

[174] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, "The marginal value of adaptive gradient methods in machine learning", *Advances in neural information processing systems*, vol. 30, 2017.

[175] J. Zhang, S. P. Karimireddy, A. Veit, *et al.*, "Why are adaptive methods good for attention models?", *Advances in Neural Information Processing Systems*, vol. 33, pp. 15 383–15 393, 2020.

[176] A. Defazio and K. Mishchenko, "Learning-rate-free learning by d-adaptation", in *International Conference on Machine Learning*, PMLR, 2023, pp. 7449–7479.

[177] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, "Language models are unsupervised multitask learners", *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[178] S. Ruder, "An overview of gradient descent optimization algorithms", *arXiv preprint arXiv:1609.04747*, 2016.

[179] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, "Improving language understanding by generative pre-training", 2018.

[180] Z. Yao, A. Gholami, S. Shen, M. Mustafa, K. Keutzer, and M. Mahoney, "Adahessian: An adaptive second order optimizer for machine learning", in *proceedings of the AAAI conference on artificial intelligence*, vol. 35, 2021, pp. 10 665–10 673.