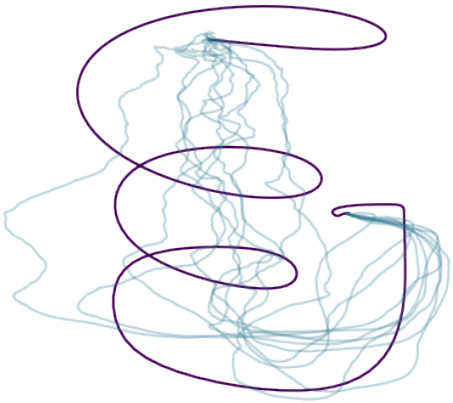
OPTIMIZATION WITHOUT BACKPROPAGATION

GABRIEL BELOUZE*

supervisor
VICTOR NICOLLET†

referent
VINCENT LEPETIT‡



Master II of Computer Science
Mathématiques, Vision & Apprentissage (MVA)

April-September 2022

* École Normale Supérieure
† Lokad Inc.
‡ École des Ponts ParisTech

ABSTRACT

Forward gradients have been recently introduced to bypass backpropagation in autodifferentiation, while retaining unbiased estimators of true gradients. We derive an optimality condition to obtain best approximating forward gradients, which leads us to mathematical insights that suggest optimization in high dimension is challenging with forward gradients. Our experiments both on test functions and high-dimensional real-world problems support this claim.

In view of all that we have said in the foregoing sections,
the many obstacles we appear to have surmounted,
what casts the pall over our victory celebration?
It is the curse of dimensionality, a malediction
that has plagued the scientist from the earliest days.

— Richard Bellman [7]

ACKNOWLEDGEMENTS

Many thanks to everybody who helped me conduct this work.

Thank you to Paul Peseux and Ziyad Benomar for their insightful comments and ideas in many informal discussions. Thank you also to Raphaël Rozenberg who, even from outside of Lokad, was able to give many helpful advices. Thanks to the engineering team of Lokad for helping me understand the ins and outs of the Envision compiler, and to all Lokad for welcoming me and providing a profitable working environment.

And of course, thank you especially to Victor Nicollet for the creation of this internship, his continuous support and his never ending expertise.

CONTENTS

ACRONYMS

AD Automatic Differentiation

ADSL Automatically Differentiable Sub-Language

DP Differentiable Programming

DSL Domain Specific Language

IL Intermediate Language

IR Intermediate Representation

JVP Jacobian Vector Product

SGD Stochastic Gradient Descent

VJP Vector Jacobian Product

1 INTRODUCTION

This document presents my work done at Lokad, as part of my end of study internship for the MVA master.

1.1 CONTEXT OF THE INTERNSHIP

1.1.1 *Lokad*

The Lokad company addresses Supply Chain Management challenges. It sells both a consulting expertise, through the roles of *Supply Chain Scientists*, as well as a complementary IT solution with its home-made Domain Specific Language (DSL) Envision. Typically, supply chain scientists answer questions about stock prediction, demand modelisation, pricing, etc.
The Envision programming language is the primary tool that accompanies supply chain scientists. The language is tailored towards expressing relational queries, à la SQL, but exposes a Python-like syntax. It offers two essential features : static relational inference, à la pandas, and Differentiable Programming (DP).

1.1.2 *Goals of the internship*

AUTOMATIC DIFFERENTIATION The DP construct in Envision is fairly recent: it stems from the work of Paul Peseux for his PhD thesis, which is still ongoing [20]. It exposes Reverse Mode Automatic Differentiation (AD) capabilities in Envision.
Automatic differentiation has become an ubiquitous tool for the machine learning practitioneer ; it enables one to use the Jacobian J_f of a primal function $f : \mathbb{R}^n \to \mathbb{R}^m$, whilst only writing the program that computes the primal. This becomes crucial for any gradient based optimization schemes (or even higher order methods). It usually comes in two modes : Forward and Reverse. Forward mode is suited for problems where $n \ll m$ and only adds a constant factor of memory and time overhead. Reverse mode is suited for problems where $m \ll n$ and adds, in its vanilla form, an unbounded memory overhead factor. In many applications, f is a loss, with $m = 1, n \gg 1$, and reverse mode, despite its cost, is prefered.

FORWARD GRADIENT However a recent paper "Gradients without Backpropagation" by Baydin et al. [5] proposes a method to use forward mode AD even in cases where $m \ll n$, through the use of

forward gradients, which are unbiased estimates of the true gradients, and showed promising results in high dimensional contexts such as Deep Learning.
This method would be, if successfully applied, especially relevant for Lokad. Data in industrial Supply Chain contexts is generally large (often of the TeraByte order), and memory management is a concern in Envision. The internship goal was thus to explore the viability of the method by Baydin et al. for the Lokad applicative context.
Models used at Lokad are typically lower dimensional than in deep learning, but have a different struture (e. g. they are not overparametrized). Evaluating the performance of forward gradient in this context would serve to further support its use in real-world applications, or on the contrary expose some of its limitations.

1.2 RESULTS

Our contribution is two fold.

The first facet of our work is theoretical. Among the family of acceptable forward gradients proposed by Baydin et al., we exhibit one that is optimal. We also find that the criterion for acceptability from [5] can be relaxed, and provide the associated relaxed optimal forward gradients.

The form that takes optimal forward gradient is particularly simple and amenable to further analysis. This leads us to show that forward gradients have theoretical shortcomings in high dimensions.

The second facet is experimental. We further the experiments from [5] with a much more comprehensive set of test functions, and find that in practice as well we observe degrading performance of forward gradients in higher dimensions.

We also implemented forward gradients in Envision, and tested them against the actual models used in production at Lokad, with their associated real world datasets. We show then that forward gradients both converge to worse loss than true gradients, and are less stable. We do find that the runtime is alleviated with forward mode, but although we show that forward mode may use arbitrily less memory on degenerated examples, there is no tangible difference in practice.

1.3 ORGANIZATION OF THIS REPORT

This documents is split into two main parts.

In chapters 2 and 3, we provide a primer on the theoretical background which the work done during the internship relies upon.

chapter 2 exposes a state of the art of autodifferentiation. Specific attention is given to the difference between forward and reverse modes, and a detail presentation of "Gradients without Backpropagation" is given, which is seminal to this internship.

chapter 3 summarizes classical gradient-based optimization algorithms in machine learning. Specifically, SGD and Adam (and its derivates) are reviewed, as they constitute the basis of our experiments.

In the second part, chapters 4, 5 and 6, we detail our work and our results.

chapter 4 presents the current implementation of autodifferentiation in Envision, and how we built upon it to expose forward gradient capabilities.

chapter 5 is more theoretical by nature. There, we derive an optimal variation to Baydin et al.'s forward gradients, and provide analytical insights as to why forward gradient descent could be challenging.

Finally, chapter 6 presents our experimental results, and shows that forward gradients fail to match the results obtained from reverse autodifferentiation.

Appendix A contains some complementary proofs.

2 AUTOMATIC DIFFERENTIATION

This chapter acts as a primer on Automatic Differentiation (AD). Automatic Differentiation is a family of algorithms that take a program that computes $f : \mathbb{R}^n \to \mathbb{R}^m$ and derive a program that computes the Jacobian $J_f : \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^n$. We review here the two *modes* of automatic differentiation, Forward and Reverse, the main implemention and optimization techniques, and finally we summarize specifically the article by Baydin et al. [5], which offers a new perspective on the modes of differentiation and is seminal for our work.

For a more comprehensive review of the theory on 'autodiff', we recommend Baydin et al.[4], and Margossian[16] for details on implementation techniques, which we barely address.

2.1 AUTOMATIC DIFFERENTIATION IN A NUTSHELL

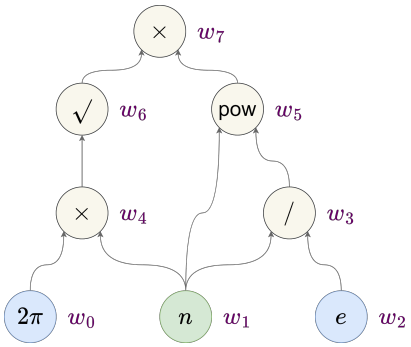
Take a program that computes some differentiable function $f : \mathbb{R}^n \to \mathbb{R}^m$, which may use complex control flow constructs, such as loops, conditionals or recursion. For a given evaluation point $\boldsymbol{x}$, we may record the computation flow to construct an *evaluation graph*, which is a directed acyclic graph where nodes express an atomic computation (Figure 1a). This in turn may be flatten in topological sort order to ultimately obtain the *evaluation trace* represented as a program in its simplest form, a Wengert list (Wengert [25]) (Figure 1b).

2.1.1 *Forward and reverse modes*

At the heart of automatic differentiation is the chain rule. For each primal variable in the evaluation trace, we compute a differential variable which carries order 1 *sensitivity* information. The specification for this information defines the *mode* of the automatic differentiation. In both modes, the chain rules expresses relationships between those variables, which allows us to compute them.

FORWARD MODE In the forward mode of automatic differentiation, we choose an *initial tangent* $\boldsymbol{v} \in \mathbb{R}^n$. v_i defines the sensitivity of the i-th parameter x_i. Then for each intermediary variable w in the Wengert list, which mathematically correspond to some function f_w of the input $\boldsymbol{x}$, we compute the *tangent* of w, which is

$$\dot{w} \triangleq J_{f_w} \cdot v$$



(a) Evaluation graph

```
w0 <- 2π
w1 <- n
w2 <- e
w3 <- w1 / w2
w4 <- w0 * w1
w5 <- w3 ^ w1
w6 <- sqrt(w4)
w7 <- w6 * w5
```

(b) Wengert list

Figure 1: Representations of a program execution

or less formally, $\dot{w} = \frac{\partial w}{\partial \boldsymbol{x}} \cdot \boldsymbol{v}$. Consider now w as an atomic function ϕ of $\boldsymbol{w}_{\text{in}} = (w_{i_1}, \ldots, w_{i_k})$, which is the vector of wengert variables that are the parent of w in the computation graph. $\boldsymbol{w}_{\text{in}}$ is itself a function f_{in} of the input, whence we get $f_w = \phi \circ f_{\text{in}}$. The chain rules then writes

$$J_{f_w}(\boldsymbol{x}) \cdot v = J_\phi(\boldsymbol{w}_{\text{in}}) \cdot J_{f_{\text{in}}}(\boldsymbol{x}) \cdot \boldsymbol{v} = J_\phi(\boldsymbol{w}_{\text{in}}) \cdot (\dot{w}_{i_1}, \ldots, \dot{w}_{i_k})$$

or again less formally, $\dot{w} = \sum_k \frac{\partial w}{\partial w_{i_k}} \cdot \dot{w}_{i_k}$. Thus, it is enough to know the jacobians of the atomic functions to inductively compute all tangents variable.
Ultimately, we compute simultaneously the primal $f(\boldsymbol{x})$ and the Jacobian Vector Product (jvp) $J_f(\boldsymbol{x}) \cdot \boldsymbol{v}$. This is a single pass of the forward mode – if we wish to compute the full jacobian instead, then n passes are necessary.

REVERSE MODE In the reverse mode of automatic differentiation, we choose an *initial cotangent* $\boldsymbol{u} \in \mathbb{R}^m$. For an intermediary Wengert variable w, we compute the *adjoint* of w, which represents the sensitivity of the output $\boldsymbol{y} \in \mathbb{R}^m$ in the cotangent direction with respect to w. That is,

$$\bar{w} \triangleq \frac{\partial \boldsymbol{y}}{\partial w} \cdot \boldsymbol{u}$$

Consider $\boldsymbol{w}_{\mathtt{out}} = (w_{j_1}, \dots, w_{j_L})$ the wengert variables that are children of w, with associated atomic functions $\phi_{j_1}, \dots, \phi_{j_L}$. Now we can again leverage the chain rule to write

$$\begin{aligned}\bar{w} &= \frac{\partial \boldsymbol{y}}{\partial w} \cdot \boldsymbol{u} \\ &= \sum_l \frac{\partial w_{i_l}}{\partial w} \cdot \frac{\partial \boldsymbol{y}}{\partial w_{i_l}} \cdot \boldsymbol{u} \\ &= \sum_l \frac{\partial \phi_{i_l}}{\partial w} \cdot \bar{w}_{i_l}\end{aligned}$$

Again, it is enough to know the jacobians of the atomic functions to inductively compute all adjoints.
Ultimately, we compute the primal $f(\boldsymbol{x})$ and the Vector Jacobian Product (vjp) $J_f(\boldsymbol{x})^\top \cdot \boldsymbol{u}$. If we wish to compute the full jacobian instead, then m passes are necessary.

There are two key differences between the two modes. First, if we wish to obtain the full jacobian, then the problem dimensions n and m will dictate which mode is better suited ; in particular in many machine learning contexts, f is a loss and $m = 1$, whence a single reverse mode pass is enough to obtain the full gradient. Second, the two modes compute sensitivity information in different orders : while forward mode compute the tangents in the same order as the primal, reverse mode starts from the end. This in fact is crucial for two reasons:

1. Forward mode may be implemented by interweaving primal and tangent computations, and in particular does not require more than twice the primal memory usage. Reverse mode must be implemented in two passes, the forward – or accumulation – pass, and the infamous backpropagation pass. Before backpropagation, *all* intermediary values must be stored. The memory overhead is proportionnal to the amount of computations, which in general is an unbounded factor of the primal memory bound (think for instance of `while` loops).

2. Forward mode does not require knowledge of the evaluation graph (the parents of a variable can be determined on the fly by looking at the atomic function call). Reverse mode must build the evaluation graph during the forward pass, which not only adds to the memory overhead, but also requires a more involved implementation.

2.1.2 *Checkpointing*

Checkpointing is a technique used to reduce peak memory usage in reverse mode, and more generally trades memory cost for runtime.

Instead of storing all intermediary variables, which may be infeasible, checkpointing strategies propose to store some variables in snapshots, and recompute some other on the fly. Thus, less memory is necessary, but primal computations may be duplicated. One key insight is that recomputations may not necessarily start from the very beginning, but instead can use some earlier intermediary snapshots..

In general, the trade-off between memory and computations translates into the choice of where to take snapshots and which variables to store in them. As argued by Dauvergne and Hascoët[9], even if the places for the snapshots are fixed, there is no single optimal choice of variables to store, but rather the optimal choice is problem dependent.

2.2 FORWARD GRADIENT

As stated earlier, forward mode autodifferentiation has better computational properties but is generally infeasible for many machine learning problems. We present here the idea by Baydin et al.[5] which opens the door to forward mode for $f : \mathbb{R}^n \to \mathbb{R}$ even with $n \gg 1$.

Given an initial tangent, a single forward pass produces the jvp $\langle \nabla f \cdot \boldsymbol{v} \rangle$, which tells us how much the gradient agrees with the tangent direction $\boldsymbol{v}$. Hence as a proxy for the true gradient, we can use the tangent scaled by this jvp. That is, Baydin et al. define the *forward gradient* to be $g(\theta) = \langle \nabla f \cdot \boldsymbol{v} \rangle \boldsymbol{v}$. The key idea is that this can be an unbiased estimate of the gradient granted $\boldsymbol{v}$ is sampled according to a carefully designed distribution. Let us first state the properties that this law must satisfy.

> DEFINITION 1: TANGENT LAW PROPERTIES
>
> We say that the probability law $\boldsymbol{p}$ on $\mathbb{R}^n$ satisfies the *tangent law properties* when the marginals $(v_1, \dots, v_n)$ of $\boldsymbol{v} \sim \boldsymbol{p}$ satisfy
>
> $$v_i \perp\!\!\!\perp v_j \quad \forall i \neq j \tag{1a}$$
> $$\mathbb{E}(v_i) = 0 \quad \forall i \tag{1b}$$
> $$\mathbb{V}(v_i) = 1 \quad \forall i \tag{1c}$$

Now, the following theorem from [5] states that indeed the tangent law properties given above are enough to make the forward gradient a good estimator for the gradient.

> THEOREM 1
>
> Let $f : \mathbb{R}^n \to \mathbb{R}$, $\boldsymbol{p}$ that satisfies the tangent law properties, and g the forward gradient associated to f and $\boldsymbol{p}$. Then $g(\boldsymbol{\theta})$ is an unbiased estimator of $\nabla f(\boldsymbol{\theta})$.

Many convergence results, e. g. for stochastic gradient descent, only assume given an unbiased estimate of the gradient, and not the true

gradient. There are thus theoretical ground supporting the use of forward gradients. Furthermore, the authors conducted experiments with simple neural networks architectures, using $\mathcal{N}(\mathbf{0}_n, \boldsymbol{I_n})$ as tangent distribution, and report encouraging results : the network weights are optimized as fast as with true gradients in terms of epochs, and faster in terms of CPU time (this makes sense as forward mode is generally faster than reverse mode).

3

OPTIMIZATION IN MACHINE LEARNING

Many machine learning problems ask to minimize an empirical risk objective, of the form

$$\mathcal{L}(\theta) = \frac{1}{M} \sum_{k=1}^{M} l(\theta; x_k)$$

with respect to the vector parameter $\theta \in \mathbb{R}^n$. The dataset $\{x_1, \dots, x_M\}$ being typically very large, gradient based optimization method use the batch gradient instead

$$g_B(\theta) = \frac{1}{|\mathcal{B}|} \sum_{k \in \mathcal{B}} \nabla l(\theta; x_k)$$

where $\mathcal{B}$ is sampled uniformly in the subsets of $\{1, \dots, M\}$ of size B. In the limit, B is equal to 1, and we obtain the simplest gradient scheme, Stochastic Gradient Descent (SGD) [22], which performs the iterative updates

$$\theta_{t+1} = \theta_t - \alpha \cdot g_1(\theta_t)$$

where the hyperparameter α is called the learning rate. When $\mathcal{L}$ is sufficiently regular, $g(\theta)$ being an unbiased estimator of the true gradient is enough to guarantee convergence, with rate $O(1/t)$ (see for instance Bach[1]).

Other accelerated schemes are derived from vanilla SGD, notably its momentum variants (Nesterov[18]), and may reach up to $O(1/t^2)$ convergence.

More recent methods, popular in the deep learning community, propose to update each coordinate of θ independently with still convergence guarantees (see Défossez et al.[10]) ; among those the ultra-widely-used Adam [13]. Adam maintains an element-wise moving average of gradients and of their square, called first and second moment.

$$\tilde{m}_{t+1,i} = \beta_1 \tilde{m}_{t,i} + (1-\beta_1) g(\theta_t)_i, \qquad m_{t,i} = \frac{\tilde{m}_{t,i}}{1-\beta_1^{t+1}}$$

$$\tilde{v}_{t+1,i} = \beta_2 \tilde{v}_{t,i} + (1-\beta_2) g^2(\theta_t)_i, \qquad v_{t,i} = \frac{\tilde{v}_{t,i}}{1-\beta_2^{t+1}}$$

The first moment acts as the usual gradient with heavy-ball momentum, and the second moment is used for element-wise scaling, yielding the following update

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{m_t}{\sqrt{v_t} + \varepsilon} \qquad \text{(operations are done element-wise)}$$

As noticed by Balles and Hennig[2], Adam can also be understood as a sign descent weighted inversely proportionnally to the relative variance of the gradient. That is

$$\begin{aligned} \frac{m_t}{\sqrt{v_t}} &= \frac{\operatorname{sign}(m_t)}{\sqrt{v_t / m_t^2}} \\ &= \operatorname{sign}(m_t) \sqrt{\frac{1}{1+\eta_t^2}} \end{aligned}$$

where $\eta_{t,i}^2 \overset{\Delta}{=} \frac{v_{t,i} - m_{t,i}^2}{m_{t,i}^2}$ is an approximation of the relative variance $\frac{\sigma_{t,i}^2}{\nabla \mathcal{L}_{t,i}^2}$, as long as m_t and v_t approximate well $\nabla \mathcal{L}$ and $\nabla \mathcal{L}^2$.

We will see later in chapter 5 that this decoupling of Adam into those 2 aspects provides insights as to what using Adam with forward gradients amounts to.

Finally, we mention here Adabelief [27] as an alternative to Adam which is believed to be more stable to noisy gradients, and have better generalization properties than Adam. Adabelief is obtained by replacing the second moment of Adam with the moving average of empirical variance, i. e.

$$\tilde{v}_{t+1} = \beta_2 \tilde{v}_t + (1 - \beta_2)(g(\theta_t) - m_t)^2$$

The rational for this update is that m_t can be interpreted as a prevision for the gradient, and v_t as our confidence in the current gradient sample with respect to what the prevision was. As such, we take big steps when our confidence is high (v_t is low), and conversely small steps when it is low.

4 AUTOMATIC DIFFERENTIATION IN ENVISION

The goal of this chapter is to bridge the gap between the theoretical AD exposition from chapter 2 and Envision. First, we detail how automatic differentiation is implemented in Envision (section 4.1). Then we discuss the specifity of differentiating over relational, categorical data (section 4.2) – a topic that stems from Paul Peseux's work [21]. Finally, we review forward mode in Envision in section 4.3: how we implemented it and what are its theoretical benefits.

4.1 INTERMEDIATE REPRESENTATION

Envision is a compiled language, and the `autodiff` construct is a first class citizen of the language. This allows for the autodifferentiation to be implemented directly over a suitable Intermediate Representation (IR) at compile time, where frameworks that add autodifferentiation capabilities to a language usually rely on operator overloading (e. g. torch.autograd for Python [19]), and work at runtime.

We review the two intermediate representations that matter from our work point of view: ADSL, where the differentiation happen, and Rex, the lowest representation before Intermediate Language (IL), the assembly language for the .NET suit which is the target of the Envision compiler. Rex matters to us as this the representation where we implement static memory analysis.

4.1.1 *ADSL*

Automatically Differentiable Sub-Language (ADSL) is a low level representation designed to be AD-closed, i. e. the adjoint of an ADSL program is an ADSL program.

4.1.1.1 *Primitives*

Without delving too far into the precise specifications of ADSL, we highlight the main constructs of the language. An ADSL program is a sequence of variable assignments, possibly destructuring tuples, and a final value, very much akin to a Wengert list [3]. Its primitives include usual arithmetic operations, calls to external process (which are ADSL programs obtained from Envision functions), and importantly control flow operators: conditionals and loops.

For a more in-depth description of how to differentiate an ADSL-like language, we refer the reader to *Don't Unroll Adjoint*[11] which

describes a very similar method to what Envision implements. Here we only add some precisions to the way loops are handled.

4.1.1.2 *Implicit Checkpointing*

Loops have two functions in ADSL. They may produce a table with the same shape as the one being looped on ; this happens if the body of the loop returns a value. They may pass values between iterations, through the use of *state variables* which persist across iterations – this may be used for instance to compute the sum of values in the table.

When no state variable is present, differentiating a loop is simple and amounts to differentiating the body independently at each iteration (this produce an adjoint table for every table that the primal loop produces). When a state variable is present, a tape is created and the successive values which the state holds during iterations are dumped onto it.

To understand why, consider the simple case where a loop accumulates a single state s over a table T which values are ignored. That is, s holds successively the values s_0, $f(s_0)$, $f(f(s_0))$, ..., for some non-linear f. Unrolling the loop, the program amounts to Program 1a (here we assume that T has size 5). The adjoint of this program writes as Program 1b. Notice how we require to have access to each intermediary values of s. As such, the reverse pass is prefaced by a taping pass where these intermediary values are computed and stored.

However, the non state variables in the loop are not stored. This means that we perform primal computations both in the taping pass and in the accumulation phase of the reverse pass. This is an instance of checkpointing (see subsection 2.1.2), characterized by the compute/store trade-off : we essentially double the computational load, but save on some memory. We will see in subsection 4.3.2 that we can leverage this mechanism to construct degenerated Envision examples that add more than a constant factor of overhead.

4.1.1.3 *Optimization*

The Envision compiler relies on one main mechanism to optimize the ADSL code produced by autodifferentiation, which is dead code elimination. The initial program computes the adjoint of all variables, however all adjoints that are not involved in the computation of the adjoints of the parameters are subsequently removed, e. g. the adjoints of constants.

4.1.2 *Rex*

ADSL is further compiled down to an even lower level language called Rex. The specifications of Rex are irrelevant for our work. The

(a) The primal loop

```
s1 <- f(s0)
s2 <- f(s1)
s3 <- f(s2)
s4 <- f(s3)
s5 <- f(s4)

s5
```

(b) The adjoint loop

```
s1 <- f(s0)
s2 <- f(s1)
s3 <- f(s2)
s4 <- f(s3)
s5 <- f(s4)

s'5 <- 1
s'4 <- s'5 * f(s'4)
s'3 <- s'4 * f(s'3)
s'2 <- s'3 * f(s'2)
s'1 <- s'2 * f(s'1)
s'0 <- s'1 * f(s'0)

s5, s'0
```

Program 1: Unrolling the loop in ADSL.

only speficity that matters to us is that the memory required by a Rex script is predictibly bounded. For instance, there are no WHILE loop primitive. This enabled us to write a memory analyzer which statically produces an upper bound on the required memory.

This upper bound is tight enough that we can use it in lieue of other solutions which measure the actual memory used during execution. The memory bounds that we will describe later in the experiments chapter are produced by this analyzer.

4.2 PARTIAL VIEWS OF PARAMETERS

Data used at Lokad, and in supply chain problems in general, is often categorical. Accordingly, the models that are learnt often involve categorical parameters.

For instance, we may want to learn the average price at which sells, respectively, t-shirts, dresses and hats. Let us ignore the fact that those quantities can be computed easily with a closed form formula, and instead suppose we want to learn them as a parameter $\alpha = (\alpha_{\text{shirts}}, \alpha_{\text{dress}}, \alpha_{\text{hat}})$ of our model, using data such as in Table 1.

DATE	ITEM	PRICE
01/01/2022	shirt	20$
01/01/2022	shirt	22$
02/01/2022	hat	15$

Table 1: Sample sell history

Machine learning practitioneers would tend to automatically encode the ITEM column with one hot vectors ϕ_{item}. Then our stochastic loss at a single observation i may be computed as $(\phi_{\text{item}}(i)^T \alpha - \text{price}(i))^2$, if for instance the quadratic loss is chosen.

However, Envision is more clever, following the idea from Peseux et al.[21]. Instead of one-hot encodings, Envision statically infers the relation between α and the ITEM column. We write our loss as $(\alpha - \text{price})^2$, and the compiler automatically selects the correct coordinate of α.

Although mathematically equivalent, the nuance is more than anecdotical, the loss is now 1-dimensional instead of 3-dimensional. We will see in chapter 6 how this is critical for forward gradients, which we will show to suffer from high dimensionality. Moreover, if we use optimizers which accumulate gradients, such as Adam, the one-hot encoding approach would pollute the moments with spurious zeros.

4.3 FORWARD MODE

4.3.1 *Implementing forward mode AD*

The implementation of forward mode is essentially trivial. Each ADSL variable is associated a dual variable which holds its tangent, akin to dual numbers [8]. The only subtlety lies in interacting well with dead code elimination (see subsubsection 4.1.1.3).

When loading a non-parameter value to a variable, such as a constant, one may be tempted to initialize its dual variable to 0. This would be mathematically correct, but the subsequent dead code elimination passes would not be able to detect that this tangent variable may be erased, nor that some of the following computations involving this variable may be erased. Instead, we keep a data structure of variables which we have created a tangent variable for, and we derive the tangent of each ADSL primitive according to the available tangent variables. In other words, we remember which tangents are null, and simplify computations accordingly.

To illustrate this point, consider the different derivations in Program 2. Program 2b does not appear to have unused variables, while mathematically it could be simplified to Program 2c.

4.3.2 *Proof of Concept for Memory Optimizations*

The purpose of this section is to illustrate the benefits from using forward mode AD in Envision on some simple programs. We exhibit a sequence of programs where forward mode uses (asymptotically) arbitrarily less memory than reverse mode AD, and runs in arbitrarily less time. This serves both as an illustration of the shortcomings of

(a) The primal program

```
$0 <- 3.000
$1 <- 4.000
$2 <- load x

$3 <- $1 * $2
$4 <- $0 * $3

loss $4
```

(b) Naive tangent program

```
$0  <- 3.000
$0′ <- 0.000
$1  <- 4.000
$1′ <- 0.000
$2  <- load x
$2′ <- load x′

$3  <- $1 * $2
$5  <- $1′ * $2
$6  <- $1 * $2′
$3′ <- $5 + $6

$4  <- $0 * $3
$7  <- $0′ * $3
$8  <- $0 * $3′
$4′ <- $7 + $8

loss  $4
loss′ $4′
```

(c) Optimized tangent program

```
$0  <- 3.000
$1  <- 4.000
$2  <- load x
$2′ <- load x′

$3  <- $1 * $2
$3′ <- $1 * 2′

$4  <- $0 * $3
$4′ <- $0 * $3′

loss  $4
loss′ $4′
```

Program 2: Naive and optimized derivation of a simple ADSL program that computes $f(x) = 3 * (4 * x)$.

```
def T(n):
    return range(10)

def F_0(x):
    return x**2

def F(n):
    if n == 0:
        return F_0

    F_{n-1} = F(n - 1)

    def F_n(x):
        for _ in T(n):
            x = F_{n-1}(x)
        return x

    return F_n
```

```
table T_1 = extend.range(10)
table T_2 = extend.range(10)
table T_3 = extend.range(10)

def autodiff pure F_0(x:
↪   number) with
    return x^2

def autodiff pure F_3(x:
↪   number) with
    each T_3 scan auto
        keep x
        each T_2 scan auto
            keep x
            each T_1 scan
            ↪   auto
                keep x
                x = F_0(x)
    return x
```

Program 3: The F_n program in Python and in Envision.

reverse mode AD seen in section 2.1, and as a proof of concept that Envision is expressive enough to create such degenerated cases.

From 4.1.1.2 we know that Envision may trade memory for performance. We show that this does not prevent bloating either performance nor memory. Consider the program F_0 which takes an input variable x and returns $f(x)$, where f is non-linear, say $f(x) = x^2$. We will define a family of programs F_n with F_0 as their building block.

> DEFINITION 2: F_n PROGRAM
>
> Let T_n be some table of size S_n. We define F_n recursively by calling F_{n-1} for each item of T_n, see Program 3.

Such program is accepted as differentiable in Envision (note the `autodiff pure` keywords), and compiles in ADSL (and later in Rex) to nested loops. How is it differentiated ?

THE DIFFERENTIATED PROGRAM F'_n Since F_0 (and inductively, all F_n) is non-linear (in the analytic sense), Reverse Mode AD needs to know `a` to compute the adjoint of `b = F_{n-1}(a)`, which writes `a' = b' * F'_{n-1}(a)`. At this point, checkpointing occurs: instead of starting over to compute a, a table T'_n is created during the forward pass, where are stored the intermediate variables $x, F_{n-1}(x), \ldots, F_{n-1}^{\circ S_n - 1}(x)$. From a high level point of view, this yields Program 4.

```
def F'_0(x):
    return 2 * x

def F'(n):
    if n == 0:
        return F'_0

    F_{n-1} = F(n - 1)
    F'_{n-1} = F'(n - 1)

    def F'_n(x):
        # Forward pass +
        ↪  taping
        T'_n = []
        for _ in T(n):
            T'_n.append(x)
            x = F_{n-1}(x)

        # Reverse pass
        x' = 1
        for x in reverse(T'_n):
            x' = x' * F'_{n-1}(x)

        return x'

    return F'_n
```

Program 4: The differentiated F'_n program, written in pseudo-Python.

Notice that F'_n yields recursive calls to both F_{n-1} and F'_{n-1}. This will explain the bloating of the amount of computations : forward computations are made several times. The memory overhead simply comes from the checkpointing mechanism which creates T'_n. We can represent the full program (without the recursive abstraction) as a temporal computation graph, à la Siskind and Pearlmutter [23], see Figure 2. Notice the fractal structure.

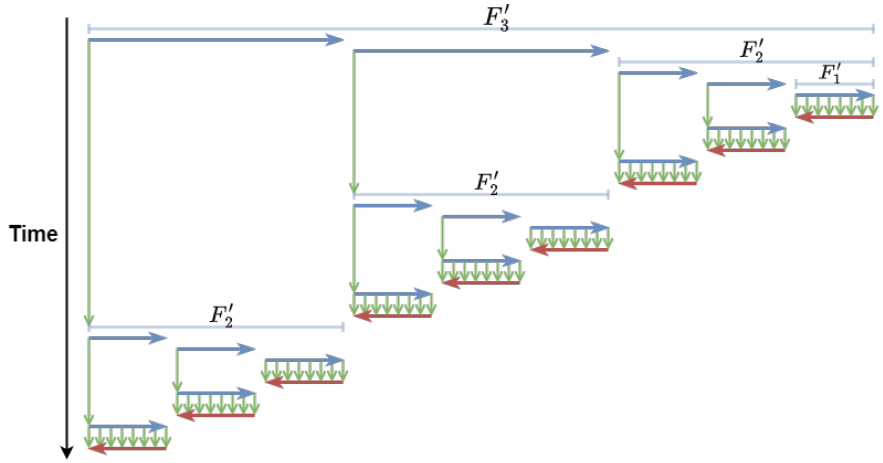


Figure 2: Derivative of the F_n program at $n = 3$. Green arrows show the flow of data, blue (resp. red) arrows show the flow of forward computations (resp. backward computations).

We now prove the following property, which quantifies the memory and computation overhead from reverse mode differentiation. The amount of computation is expressed in number of calls to F_0. The memory overhead bound assumes that the compiler is able to perfectly reuse the allocated space (e. g. T'_n is allocated only once), which is the best case scenario.

> PROPERTY 1: OVERHEAD OF F'_n
>
> The F'_n program has the following costs
>
> A. $O(n \prod_i S_i)$ calls to F_0 and F'_0
>
> B. $O(\sum_i S_i)$ memory overhead

Proof. Let C_n be the number of calls to F_0 and F'_0 of F_n, D_n that of F'_n. The following relation holds for $n \geqslant 1$

$$\begin{cases} C_n = S_n C_{n-1} \\ D_n = S_n C_{n-1} + S_n D_{n-1} \end{cases}$$

Let $X_n = \begin{pmatrix} C_n \\ D_n \end{pmatrix}$. We readily have

$$X_n = (\prod_i S_i) \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 1 \\ 1 \end{pmatrix} = (\prod_i S_i) \begin{pmatrix} 1 & 0 \\ n & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

This is enough to conclude. □

The costs from Property 1 should be compared with those of the primal program, which are

A. $C = O(\prod_i S_i)$ calls to F_0

B. $D = O(1)$ memory overhead

Hence in the worst case scenario where all tables have size 2, reverse mode adds a logarithmic factor $\log(C)$ to both the number of computations and the memory allocations. On the contrary, forward mode AD, used for computing forward gradients, adds only a constant factor to both.

5 OPTIMIZATION WITH FORWARD GRADIENTS

In this chapter, we study the mathematical implications underpinning the use of forward gradients, rather than true gradients, in standard optimization. section 5.1 focuses on the choice of distribution for the initial tangent ; we exhibit a *best distribution* according to the constraints from Baydin et al., then show that those constraints can be relaxed, which notably yields a more general family of best distributions. Finally, the important section 5.2 features evidence that optimization with forward gradient should get challenging with higher dimensional functions.

5.1 CHOICE OF TANGENT LAW

Forward gradients are parametrized by a choice of direction of projection for the real gradient. To compute the forward gradient g at $\boldsymbol{\theta}$, a random direction $\boldsymbol{v}$ is sampled and used as the *tangent*. Forward mode AD then yields the jacobian vector product $\nabla f(\boldsymbol{\theta}) \cdot \boldsymbol{v}$, and finally the forward gradient is computed as $g(\boldsymbol{\theta}) \stackrel{\Delta}{=} (\nabla f(\boldsymbol{\theta}) \cdot \boldsymbol{v})\boldsymbol{v}$. We naturally refer to the distribution for $\boldsymbol{v}$ as the *tangent law*, and write $\boldsymbol{p_v}$. From the tangent law properties, it is natural –though not necessary– that the v_i should be i. i. d., in which case we note the common distribution $\boldsymbol{p}_v$.

In the original forward gradient paper, Baydin et al. use $\boldsymbol{p_v} = \mathcal{N}(\mathbf{0}_n, \boldsymbol{I}_n)$, and note that any distribution that satisfies the tangent law properties yields valid forward gradients, in the sense that they are unbiased estimators of the gradient. This opens the door to other choices of tangent laws – notably one that minimizes variance.

> DEFINITION 3: MINIMAL TANGENT LAW
>
> We call *minimal tangent law*, and we write $\boldsymbol{p}_v^{\min}$, the centered Rademacher law
>
> $$v \sim \mathcal{R}ad(0.5) \Leftrightarrow \begin{cases} v &= 1 \quad \text{w. p. } 0.5 \\ v &= -1 \quad \text{w. p. } 0.5 \end{cases}$$

> LEMMA 1: MINIMALLY DEVIATING FORWARD GRADIENTS
>
> Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function of n real variables $(\theta_1, \dots, \theta_n)$. Among the tangent laws that satisfy the tangent law properties, the choice $\boldsymbol{p}_v = \boldsymbol{p}_v^{\min}$ is *the* minimizer for the forward gradient g of the mean squared deviation $\mathbb{E}\left[\|g(\boldsymbol{\theta}) - \nabla f(\boldsymbol{\theta})\|^2\right]$.

Proof. After expansion of the mean squared deviation, this amounts to minimizing each $\mathbb{E}\left[(g_i(\boldsymbol{\theta}) - \nabla f_i)^2\right]$. From the bias-variance property, and because the forward gradient is unbiased, this in turns asks to minimize $\mathbb{V}[g_i(\boldsymbol{\theta})]$. The mean $\mathbb{E}[g_i(\boldsymbol{\theta})]$ being fixed (equal to ∇f_i), minimizing the variance amounts to minimizing each

$$\begin{aligned}
\mathbb{E}\left[g_i(\boldsymbol{\theta})^2\right] &= \mathbb{E}\left[(\nabla f \cdot v)^2 v_i^2\right] \\
&= \left(\frac{\partial f}{\partial \theta_i}\right)^2 \mathbb{E}[v_i^4] + \sum_{j \neq i} \left(\frac{\partial f}{\partial \theta_j}\right)^2 \mathbb{E}\left[v_i^2 v_j^2\right] \\
&\quad + 2 \sum_{k<l} \left(\frac{\partial f}{\partial \theta_k}\right)\left(\frac{\partial f}{\partial \theta_l}\right) \mathbb{E}\left[v_i^2 v_k v_l\right] \\
&= \left(\frac{\partial f}{\partial \theta_i}\right)^2 (1 + \mathbb{V}[v_i^2]) + \sum_{j \neq i} \left(\frac{\partial f}{\partial \theta_j}\right)^2 \mathbb{E}\left[v_i^2 v_j^2\right] \\
&= \left(\frac{\partial f}{\partial \theta_i}\right)^2 \mathbb{V}[v_i^2] + \|\nabla f\|^2
\end{aligned}$$

where we used the following properties

1. $\mathbb{E}\left[v_i^2 v_k v_l\right] = 0$ when $k \neq l$. Indeed the v_j are independent and centered (from Equation 1), and at least one of v_k, v_l appears alone in $v_i^2 v_k v_l$.
2. $\mathbb{E}[v_i^2 v_j^2] = 1$ when $i \neq j$ (this follows again from Equation 1).
3. $\mathbb{E}[v_i^4] = 1 + \mathbb{V}[v_i^2]$.

Finally, $\mathbb{V}[v_i^2]$ is minimum at 0, which is feasible when v_i takes value in $\{-a, a\}$ for some $a \in \mathbb{R}$. We obtain the *minimal deviation conditions*

$$\begin{cases}
v_i \perp\!\!\!\perp v_j & \forall i \neq j & \text{(2a)} \\
\mathbb{E}(v_i) = 0 & \forall i & \text{(2b)} \\
\mathbb{V}(v_i) = 1 & \forall i & \text{(2c)} \\
\mathbb{V}(v_i^2) = 0 & \forall i & \text{(2d)}
\end{cases}$$

They are met only when the v_i are independent Rademacher variables with parameter 0.5, i. e. $p_v = p_v^{\min}$. □

Note that in the course of the proof we also found an explicit value for the mean squared deviation. The following property makes it explicit (see the short proof in the appendix).

PROPERTY 2

> The mean squared deviation of the minimally deviating forward gradients is
>
> $$\mathbb{E}\left[\|\nabla f(\boldsymbol{\theta}) - g(\boldsymbol{\theta})\|^2\right] = (n-1)\,\|\nabla f(\boldsymbol{\theta})\|^2$$

At this point, one may worry that the minimal tangent law that we found is anisotropic, while it is not obvious where the loss of isotropy happened. It is in fact the tangent law properties that implicitly assumes the anisotropic choice of canonical basis. Indeed, if the marginals of $\boldsymbol{v}$ according to some orthogonal basis $\boldsymbol{b}$ respect Equation 1, the marginals according to some other orthogonal basis $\boldsymbol{b}'$ may not.

The tangent law properties can however be readily relaxed to an isotropic formulation. This the purpose of the following definition, and its associated theorem which extends Theorem 1.

> DEFINITION 4: EXTENDED TANGENT LAW PROPERTIES
>
> We say that the random vector $\boldsymbol{w}$ on $\mathbb{R}^n$ satisfies the *extended tangent law properties* when there exists a random variable $\boldsymbol{q} \in \mathbb{O}_n$ over the orthogonal group of $\mathbb{R}^n$, and $\boldsymbol{v}$ that satisfies Equation 1, such that $\boldsymbol{w} = \boldsymbol{q} \cdot \boldsymbol{v}$.

Does the extended tangent law property add any useful distribution to the set of available tangent laws ? It arguably does. For instance, the uniform distibution over the L_2-sphere of radius $\sqrt{n}$, which is not admissible in the formulation of Baydin et al., satisfies the extended tangent law property. Indeed, it can be seen as the law of $\boldsymbol{q} \cdot \boldsymbol{v}$ where $\boldsymbol{q}$ is uniform over $\mathbb{O}_n$ (i. e. following the translation invariant measure, or Haar measure, over the orthogonal group), and $\boldsymbol{v}$ follows the minimal tangent law.

> THEOREM 2: EXTENDED FORWARD GRADIENT THEOREM
>
> Let $f : \mathbb{R}^n \to \mathbb{R}$, $\boldsymbol{p}$ that satisfies the extended tangent law properties, and g the forward gradient associated to f and $\boldsymbol{p}$. Then $g(\boldsymbol{\theta})$ is an unbiased estimator of $\nabla f(\boldsymbol{\theta})$.

Proof. Let $\boldsymbol{w} = \boldsymbol{q} \cdot \boldsymbol{v}$ be the random tangent associated to $\boldsymbol{p}$ (see Definition 4). We rely on the decomposition

$$\mathbb{E}[g(\boldsymbol{\theta})] = \int_{q \in \mathbb{O}_n} \mathbb{E}[g(\boldsymbol{\theta}) \mid \boldsymbol{q} = q]\,dq$$

Hence it suffices to show that $g(\boldsymbol{\theta})$ is an unbiased estimator of the gradient given $\boldsymbol{q}$. We can simply recycle the proof of Theorem 1 from "Gradients without Backpropagation", with a change of basis defined by q. We have

$$\begin{aligned}\langle \nabla f; \boldsymbol{w}\rangle &= \langle q^{-1} \cdot \nabla f; \boldsymbol{v}\rangle \\ \langle \nabla f; \boldsymbol{w}\rangle \boldsymbol{w} &= q \cdot \langle q^{-1} \cdot \nabla f; \boldsymbol{v}\rangle \boldsymbol{v}\end{aligned}$$

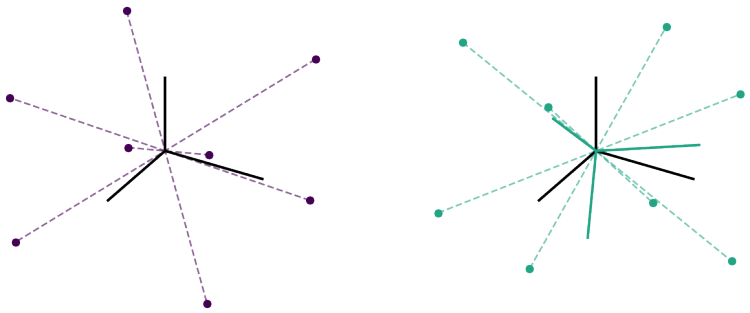
whence from Theorem 1 we get that $q^{-1}g(\boldsymbol{\theta})$ is an unbiased estimator of $q^{-1}\nabla f(\boldsymbol{\theta})$, i. e. that $g(\boldsymbol{\theta})$ is an unbiased estimator of $\nabla f(\boldsymbol{\theta})$. This concludes the proof that $\mathbb{E}[g(\boldsymbol{\theta}) \mid \boldsymbol{q} = q] = \nabla f(\boldsymbol{\theta})$, and hence proves the theorem. □

Of course, the extended formulation yields an extended family of minimally deviating forward gradients.

> DEFINITION 5: EXTENDED MINIMAL TANGENT LAWS
>
> Let $\mathcal{O}_n$ be the set of orthogonal automorphism of $\mathbb{R}_n$. The family of *extended minimal tangent laws* is described by the random variables $T = \boldsymbol{q} \cdot \boldsymbol{v}$ where $\boldsymbol{v} \sim \boldsymbol{p}_v^{\min}$ and $\boldsymbol{q}$ is any random variable over $\mathcal{O}_n$.

The simplest members of this family are those associated to almost surely constant $\boldsymbol{q}$, which are the independent centered Rademacher marginals associated to each orthogonal basis – see Figure 3. The uniform distibution over the L_2-sphere of radius $\sqrt{n}$ also belongs to the extended minimal tangent laws, and is *the* minimal tangent law that is isotropic.

We can furthermore check that all laws in this family are equivalently good (regardless of ∇f), as the mean squared deviation given by Property 2 is invariant to orthogonal transformations of $\boldsymbol{v}$, and equals $(n-1)\|\nabla f\|^2$.



(a) Canonical minimal tangent law.

(b) Another minimal tangent law. Canonical basis is in black, rotated basis is in turquoise.

Figure 3: Minimally deviating tangent laws in dimension $n = 3$. Each vertex has equal probability $1/8$.

5.2 MISTAKES IN FORWARD GRADIENT DESCENT

Proofs of convergence for optimizers generally only assume an unbiased oracle of the gradient of the objective (see for instance Moulines

and Bach[17] or Défossez et al.[10]) – usually to account for stochastic data samples. In particular, this means that we keep the same theoretical guarantees while using forward gradients. However, this does not account for the intuition that noisier oracles yield harder optimization. The goal of this section is to provide a *quantitative description* of the added stochasticity that comes from using forward gradients rather than true gradients.

In this section, we only look at minimally deviating forward gradients. Not only is the minimal tangent law more convenient to analyze because of its simple form, but we expect them to yield better optimizations than any other tangent laws.

5.2.1 *The Curse of Dimensionality*

The purpose of this section is to prove Theorem 3, which shows that the sign of the forward gradient correlates with the sign of the true gradient only on $o(n)$ dimensions more than what pure chance accounts for.

First, however, we provide two reasons we wish the sign of forward gradients to approximate well the sign of the true gradient. They both stem from the remark that the forward gradient associated to a minimally deviating tangent law has a distinct structure, namely all its coefficients share the same magnitude. What consequence does this structure have with different optimizers ?

Consider first the case of Clipped SGD. Clipped SGD is a regularized version of SGD, which enforces an upperbound on the magnitude of the coordinates: in lieu of the gradient $\nabla f(\theta)$, clipped SGD uses its projection onto the ball $\{\theta \mid \|\theta\|_\infty \leqslant 1\|\}$. That is, for coefficients that have a magnitude greater than 1, their sign is used instead. When using Clipped SGD with forward gradients, either all coefficients are transformed into their sign, or none are. Thus, when the gradient is large enough, all coefficients have too large a magnitude, Clipped SGD with forward gradients amouts to sign descent.

A similar mechanism can be seen with Adam. Recall from chapter 3 the decoupled interpretation of Adam, as coordinate-wise weighted sign descent. With forward gradients, the weights are in fact equal across coordinates, whence, up to this scaling factor, the optimization process amounts to sign descent.

> THEOREM 3

> The expected number of dimensions where the minimally deviating forward gradients has the same sign as the true gradient is upper bounded by
>
> $$\frac{n}{2} + \sqrt{\frac{n}{2\pi}} + O(\frac{1}{\sqrt{n}})$$

Proof. Let us notice first that the forward gradient $g = (\nabla f \cdot v)v$ always has a positive correlation with the true gradient:

$$\nabla f \cdot g = (\nabla f \cdot v)^2 \geqslant 0$$

In other words, the forward gradient changes the sign of v if necessary, so that it correlates with ∇f (it also adds a scaling factor which is irrelevant for the theorem).

Now, first, let us assume that $\nabla f \in \{-1, 1\}^n$. Let us note P the expected number of dimensions with the same sign as the true gradient, and N that of dimensions with the opposite sign. We have

$$\begin{cases} P + N = n & \text{(3a)} \\ P - N = \mathbb{E}\left[\text{sign}(\sum_i \nabla f_i v_i) \sum_i \text{sign}(\nabla f_i) v_i\right] = \mathbb{E}\left[|\sum_i \nabla f_i v_i|\right] & \text{(3b)} \end{cases}$$

By symmetry, $\mathbb{E}[|\sum_i \nabla f_i v_i|]$ is $\mathbb{E}[|\sum_i v_i|]$. This is the well known problem of estimating how far away from 0 does a n-step random walk goes. The derivation is detailed in Appendix A, see section A.2, and gives $\mathbb{E}[|\sum_i v_i|] = \sqrt{\frac{2n}{\pi}} + O(\frac{1}{\sqrt{n}})$. Eventually, we get

$$P = \frac{n}{2} + \frac{1}{2}\mathbb{E}[|\sum_i v_i|] = \frac{n}{2} + \sqrt{\frac{n}{2\pi}} + O(\frac{1}{\sqrt{n}})$$

Remains to show that this holds as an upper bound for general ∇f. This is readily obtained, as Equation 3 now writes

$$\begin{cases} P + N = n \\ P - N = \mathbb{E}\left[\text{sign}(\sum_i \nabla f_i v_i) \sum_i \text{sign}(\nabla f_i) v_i\right] \end{cases}$$

where we have an upper bound $\mathbb{E}\left[\text{sign}(\sum_i \nabla f_i v_i) \sum_i \text{sign}(\nabla f_i) v_i\right] \leqslant \mathbb{E}\left[|\sum_i \text{sign}(\nabla f_i) v_i|\right]$ where we previously had an equality. □

Note that a pure random walk in $\mathbb{R}^n$ would get right on average $\frac{n}{2}$ dimensions.

5.2.2 *Forward Gradient for Linear Objectives*

The essence of first-order gradient descent methonds is to use at θ the best linear approximation of the objective and move accordingly.

Here we provide analysis of the behaviour of forward gradients when the linear approximation is exact, i. e. for a linear objective. Its ability – or lack there of – to degenerate the objective to $-\infty$ will give insights as to how it will fair against reverse mode gradients on more complex objectives.

In the following, we let $f(\boldsymbol{\theta}) \triangleq \boldsymbol{\mu}^T \cdot \boldsymbol{\theta}$ for some $\boldsymbol{\mu} \in \mathbb{R}^n$.

5.2.2.1 *SGD*

The update rule for vanilla SGD writes here

$$\begin{cases} \delta\boldsymbol{\theta}_t = (\boldsymbol{v}_t^T \cdot \boldsymbol{\mu})\boldsymbol{v}_t \\ \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \delta\boldsymbol{\theta}_t \end{cases}$$

where we assumed a learning rate of 1 for simplicity, and where the $(\boldsymbol{v}_s)_{s=1,2,\dots}$ are independent random variables with law $\boldsymbol{p}_{\boldsymbol{v}}^{\min}$.

We can compute the mean gain to the objective of a single SGD step

$$\begin{aligned} \mathbb{E}\left[f(\boldsymbol{\theta}_{t+1}) - f(\boldsymbol{\theta}_t)\right] &= -\mathbb{E}[(\boldsymbol{v}_t^T \cdot \boldsymbol{\mu})^2] \\ &= -\boldsymbol{\mu}^T \cdot \Sigma \cdot \boldsymbol{\mu} \\ &= -\|\boldsymbol{\mu}\|^2 \end{aligned}$$

where the tangent covariance $\Sigma \triangleq \mathbb{E}[\boldsymbol{v} \cdot \boldsymbol{v}^T]$ is $\boldsymbol{I}_n$ (following the tangent law properties).

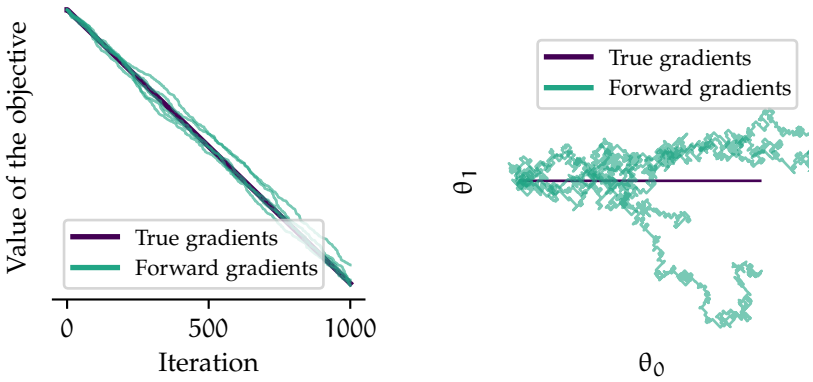
This is the same expected gain as with regular reverse-mode SGD, and indeed Figure 4a shows similar asymptotic evolution of the objective. Yet in high dimensions, we showed in subsection 5.2.1 that on average the forward gradient gets only $o(n)$ more than half directions correct. How does it manage to keep up with the always-correct true gradient ? Simply, forward SGD takes bigger steps.

Indeed consider the following simple derivation when $\boldsymbol{v} \sim \boldsymbol{p}_{\boldsymbol{v}}^{\min}$

$$\begin{aligned} \mathbb{E}\left[\left\|(\nabla f^T \cdot \boldsymbol{v})\boldsymbol{v}\right\|^2\right] &= n\mathbb{E}\left[(\nabla f^T \cdot \boldsymbol{v})^2\right] \\ &= n\|\nabla f\|^2 \end{aligned}$$

Whereas SGD in reverse mode takes step of squared norm size $\|\nabla f\|^2$, forward adds on average a factor equal to the dimension. While this has no effect for the simple linear objective, this may slow down or prevent convergence on more chaotic problems. As seen in Figure 4b, the parameter $\boldsymbol{\theta}$ may wander far from the gradient direction.

Note that in fact the derivation above remains valid if the gradient has only constant direction with possibly varying norm. That is, in a region where the isopleths are plane and parallel, we expect forward gradient descent and true gradient descent to perform equally well.



(a) Evolution of the objective.

(b) Evolution of the first two coordinates of $\boldsymbol{\theta}$

Figure 4: SGD with linear objective, with forward gradients and true gradients. 5 iterations of forward gradient descent are represented.

5.2.2.2 *Adam*

Adam with constant gradient $\nabla f = \boldsymbol{\mu}$ amounts to sign descent (up to the Adam's ε), i. e. $\delta\boldsymbol{\theta}_i = \operatorname{sign}(\boldsymbol{\mu}_i)$. The speed of divergence in this case is characterized by

$$\delta\boldsymbol{\theta}^T \cdot \boldsymbol{\mu} = \|\boldsymbol{\mu}\|_1$$

How does Adam with forward gradients compare ? The update rule for Adam writes as the following (we use p_t for the second moment of Adam to not conflict with the notation for the initial forward tangent $\boldsymbol{v}$).

$$\begin{cases} m_t = (1-\beta_1)\sum_{s=1}^{t} \beta_1^{t-s}(\boldsymbol{v}_t^T \cdot \boldsymbol{\mu})\boldsymbol{v}_t \\ p_t = (1-\beta_2)\sum_{s=1}^{t} \beta_2^{t-s}(\boldsymbol{v}_t^T \cdot \boldsymbol{\mu})^2 \mathbf{1} \\ \delta\boldsymbol{\theta}_{t,i} = \dfrac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}\dfrac{m_{t,i}}{\sqrt{p_{t,i}}+\varepsilon} \\ \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \delta\boldsymbol{\theta}_t \end{cases}$$

where we assumed a learning rate of 1 for simplicity, and where the $\boldsymbol{v}_i$ are independent random variables with law $p_v^{\min}$. Typically, β_1 and β_2 are close to 1. To derive analytical results, we will consider the degenerated case where they are both 1, whence

$$\delta\boldsymbol{\theta}_{t,i} = \frac{(\boldsymbol{v}_t^T \cdot \boldsymbol{\mu})\boldsymbol{v}_{t,i}}{\sqrt{(\boldsymbol{v}_t^T \cdot \boldsymbol{\mu})^2}} = \operatorname{sign}(\boldsymbol{v}_t^T \cdot \boldsymbol{\mu})\boldsymbol{v}_{t,i}$$

(we also remove the ε as we did in reverse mode).

Now we can compute the associated speed of divergence:

$$\begin{aligned}\mathbb{E}[(\delta\boldsymbol{\theta}_t \cdot \boldsymbol{\mu})] &= \mathbb{E}[\operatorname{sign}(\boldsymbol{v}_t^\top \cdot \boldsymbol{\mu})(\boldsymbol{v}_t^\top \cdot \boldsymbol{\mu})] \\ &= \mathbb{E}[|\boldsymbol{v}_t^\top \cdot \boldsymbol{\mu}|]\end{aligned}$$

In all generality, we can only bound this coefficient by the one obtained with true gradients: $\mathbb{E}[(\delta\boldsymbol{\theta}_t \cdot \boldsymbol{\mu})] \leqslant \|\boldsymbol{\mu}\|_1$. However, for most $\boldsymbol{\mu}$, this bound is very coarse, and there is an additional factor of degradation proportional to the dimension.

In particular, when $\mu \in \{-1, 1\}^n$, we get a $O(\sqrt{n})$ speed of divergence (see the derivation in the appendix), while reverse mode gives a $O(n)$ speed of divergence. We find here another 'curse of dimension': as the dimension of the parameter grows, we expect forward gradients to perform increasingly worse than true gradients.

6 EXPERIMENTS

We conducted two series of experiments.

The first series, presented in section 6.1, involves so called test functions. This gives us an idea of the behaviour of optimization with forward gradients that is somewhat generalizable to many domains of application.

The second series, which we present in section 6.2, was conducted on Envision scripts used in production at Lokad. They give a very precise description of the applicability of forward gradients for the Lokad use case, but may not translate to other domains. In this second series, we are not only interested in convergence, but also in computational metrics, i. e. memory load and speed of execution.

6.1 EXPERIMENTS ON TEST FUNCTIONS

The litterature provides an extensive body of *test functions* meant to challenge and evaluate new optimization algorithms, see for instance [14] (Appendix B) and [26] for a curated list of such function, or [12] for a comprehensive survey. Baydin et al. test in [5] two such functions – Beale and Rosenbrock – against forward gradients, with vanilla SGD, and show positive results. However both functions are 2-dimensional, which, according to our work in subsection 5.2.1, should not make apparent the potential pitfalls of forward gradients.

Thus, this section is meant to provide additional 'unit testing' experiments against forward gradients. It is not trivial to conduct a meaningful comparison experiment between optimization algorithms. We did our best to follow the recommendations from "Best practices for comparing optimization algorithms"[6]. In particular, we use diverse test functions and we test convergence from several starting points.

6.1.1 *Specification*

Here we detail our experimental protocol.

6.1.1.1 *Functions used*

Out of reproducibility concerns, we brought Beale and Rosenbrock in our experiments. Note that Rosenbrock has a definition for arbitrary dimensions, although it was used only in dimension 2 in [5] (known then as the banana function).

For convergence plots, we also use the simple sphere function, and the hyperellipsoid function, which are both separable. Separability ensures the problem should be trivial for simple gradient descent, while forward gradient descent may still struggle in high dimensions. The hyperellipsoid function is similar to the sphere function but is highly anisotropic. This may be relevant to test, considering the shape of the minimal tangent law.

$$f_{Sphere}(\boldsymbol{x}) = \sum_{i=1}^{D} x_i^2$$
$$f_{Ellipsoid}(\boldsymbol{x}) = \sum_{i=1}^{D} i x_i^2$$

Mainly for illustration purposes, we add our own function that we call the spiral function, and is defined as

$$f_{Spiral}(x, y, z) = \log(1 + z^2 + (x - \sin(z))^2 + (y - \cos(z))^2)$$

The spiral function is three-dimensional, which is the highest that we can reasonnably represent the path of $\boldsymbol{\theta}$ during optimization. The figure on the title page is a trajectory plot of true gradient descents compared to forward gradient descent for the spiral test function.

Our main concern remains robustness of results, which demands a large set of test functions. We used the collection implemented by Thevenot[24], which regroup 78 test functions well known in the litterature, and include convex, non-convex, separable, non-separable, multimodal and non-multimodal functions.

For each of those functions that are defined in arbitrary dimension, we test them in dimensions 2, 10 and 100.

6.1.1.2 *Optimizers used*

We chose to test SGD, Clipped SGD, Adam and Adabelief. SGD yields the most bare view of the behaviour of forward gradients. However SGD is sensitive to variations of the learning rate, and prone to diverge, and hence we also include Clipped SGD as a regularized version of SGD. Adam is maybe the most popular optimizer in machine learning. Adabelief appears to exhibit better robustness to gradient noise than Adam, which is of course highly desirable with forward gradients.

For each such optimizer, we test a version that uses forward gradients, and one that uses true gradients.

We keep a fixed learning rate equal to 0.01. In Envision, this is the learning rate that is used under the hood, and the design choice of the language is to *not allow the supply chain scienctists to change it*. In other words, we voluntarily restrict any hyperparameter optimization to match the way it is restricted in Envision.

6.1.1.3 *Reproducibility*

Our 'unit testing' experiments were done in Python, which is more convenient that Envision, and is available to anyone. Our implementation and experimentations are publicy available on Github, it relies on the autograd library [15] for performing autodifferentiation.

6.1.2 *Results*

Here, we are only interested in the performance of the optimizers with respect to the number of gradient evaluation. In particular, we do not compare the CPU time of execution, nor the memory usage as we make no attempt in our implementation to optimize one or the other. Those metrics will be evaluated in 6.2 where our Envision implementation is used.

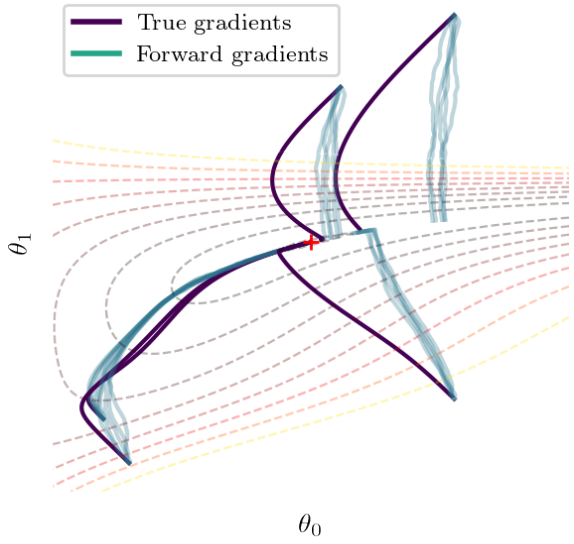
TRAJECTORY AND CONVERGENCE PLOTS We first start by providing some trajectory and convergence plots. They are not the most practical, as they do not allow comparisons across test functions, but nevertheless can be useful to form an idea of how optimizers behave.

Figure 5 reproduces the results from Baydin et al. with several initializations for θ. It can be seen that it depends on the initialization wether true gradient descent or forward gradient descent performs better, although forward gradients are never but marginally better.

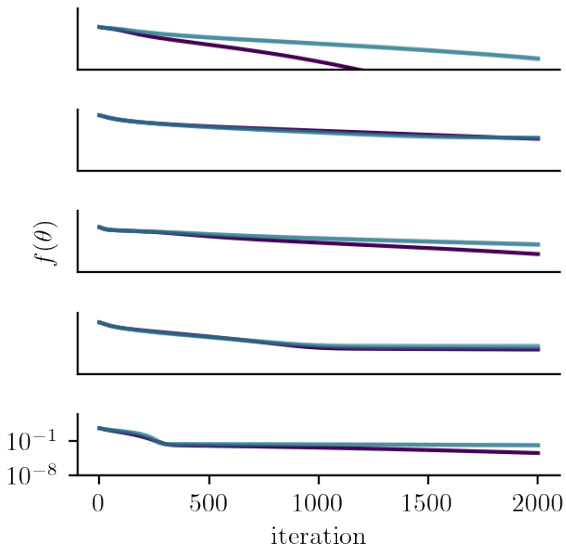
We can furthermore see how the convergence plots evolve as the function dimension grows, which we illustrate with the ellipsoid function in Figure 6. We can see that forward SGD does not seem to suffer from dimensionality. Indeed for low enough learning rates, the hyperellipsoid isopleths look straight and parallel on the path of gradient descent, which we showed in subsubsection 5.2.2.1 to be a case where forward gradients can readily replace true gradients. However, the same does not hold for Adam which displays convincingly how forward gradients struggle with high dimensionality.

PERFORMANCE PROFILE Beiranvand, Hare, and Lucet[6] recommend using performance profiles to report optimization experiments in a single graphic.

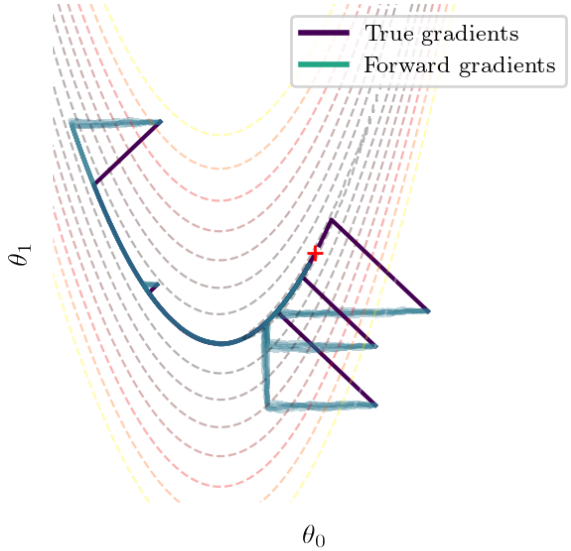
Let $\mathcal{P}$ be a set of problems (here test functions), and $\mathcal{S}$ a set of optimizers. Suppose our experiments produce a fixed-target metric $t_{p,s}$ for each problem p and solver s, here we use the number of



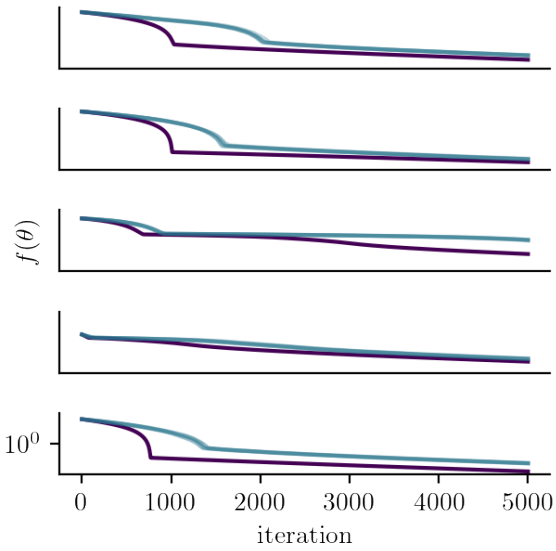
(a) Beale trajectory plot.



(b) Beale convergence plots for each initialization.



(c) Banana trajectory plot.

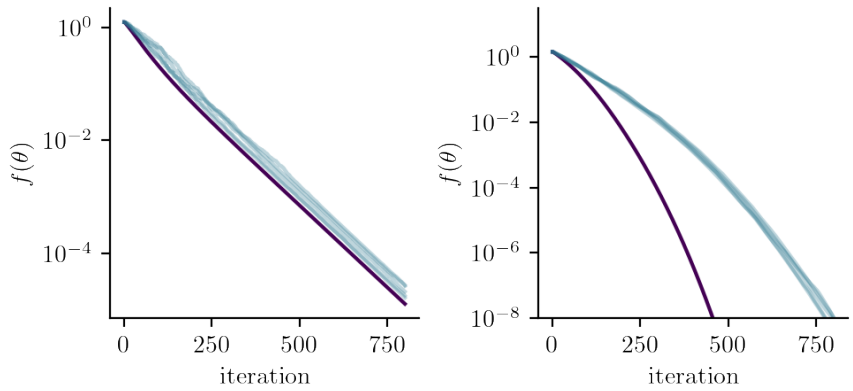


(d) Banana convergence plots for each initialization.

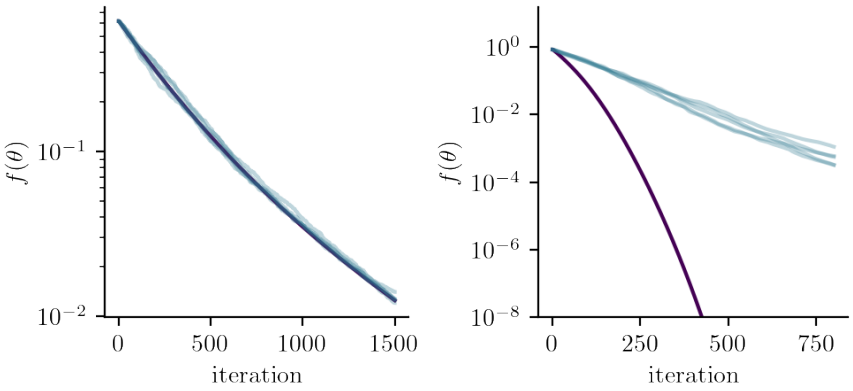
Figure 5: Vanilla gradient descent for the Beale and Banana test functions, with 5 random initializations.

function evaluation to reach the minimum up to $\epsilon = 0.1$. Then we define the performance ratio

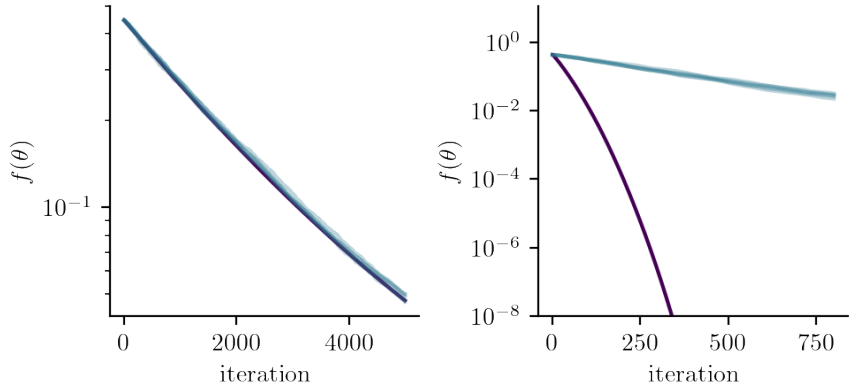
$$r_{p,s} = \begin{cases} \frac{t_{p,s}}{\min\{t_{p,s} \mid s \in \mathcal{S}\}} & \text{if convergence test passed,} \\ \infty & \text{otherwise} \end{cases}$$



(a) Dimension 2



(b) Dimension 10



(c) Dimension 100

Figure 6: Evolution of SGD (left half) and Adam (right half) with the hyper-ellipsoid of dimension 2, 10 and 100.

From this we derive the *performance profile*, which is a function of $\tau \geqslant 1$ for each solver :

$$\rho_s(\tau) = \frac{1}{|\mathcal{P}|} \text{size}\{p \in \mathcal{P} \mid r_{p,s} \leqslant \tau\}$$

That is, $\rho_s(\tau)$ is the proportion of problems where solver s is less than a factor τ away from the best solver.

In particular, $\rho_s(1)$ is the portion of time that s was the best solver, and $\rho(\infty)$ is the proportion that the solver managed to solve. In general, we are looking for solvers with consistently high ρ.

Performance profiles are convenient in that they display several information in a single graphic. However they treat all problems uniformly. In our case, we also would like to know how performance evolves with higher dimensionality. Thus, we provide in Figure 7 three profiles taken from problems with three different dimensions, 2, 10 and 100.

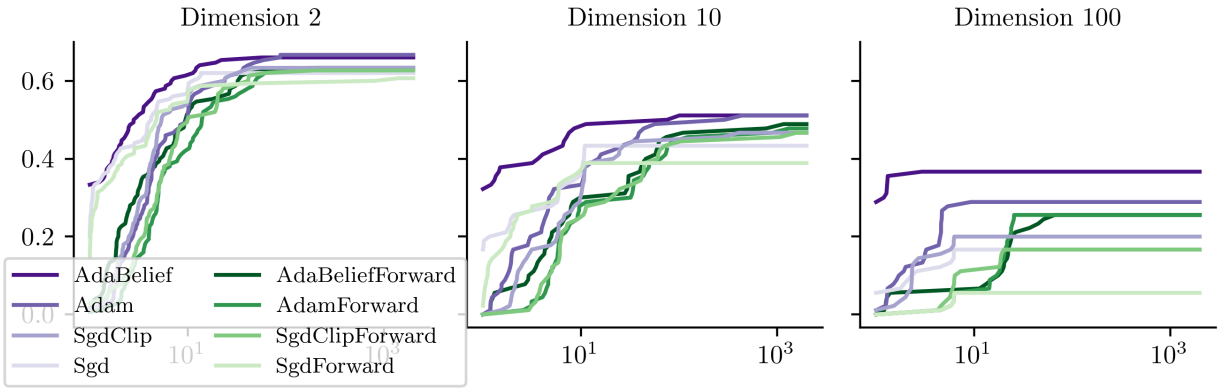


Figure 7: Performance profiles across sets of problems with fixed dimensionality.

Although we should be prudent with conclusions about 2-by-2 comparisons with performance profiles, we can observe the effect of higher dimensions, especially in the region $\tau \leqslant 10$. That is, in dimension 100, any optimizer that uses forward gradients are almost always at least 10 times slower than the best optimizer that uses true gradient (which almost always is Adabelief). Note than on top of this, all optimizers converge less often in high dimensions (the SGD based optimizers seem to be the most impacted).

Moreover, we can see that for forward gradient based optimizers, the derivatives of the performance profile in 1 gets flatter as the dimension increases $\rho'(1) \approx 0$. This means that not only are they almost never optimal (as $\rho(1) \approx 0$), but they also become poor approximators of the optimal solver in high dimensions. In contrast, Adam with true gradients has often good performance, albeit never optimal.

6.2 ENVISION PRODUCTION SCRIPTS

The Envision language is developped internally at Lokad and is unavailable to the general public. As such, we had straightforward access to all the scripts ever written in Envision, and the data they use. This enabled us to conduct the same experiments as in section 6.1 but

using the actual production scripts instead of test functions.

Lokad provides to the supply chain scientists frozen scripts called *templates* which implement a certain supply chain model. An *instance* of the template corresponds to choosing hyperparameters for the model and feeding data to it.

For a given template, there can be up to dozens of instances, which we treat as separate problems in our experiments. However note that the impact of using forward gradients is somewhat consistent across instances of a same template. Notably, we will see that forward gradient descent fails to converge on many problems, but these correspond to only a handful of models. Similarly, there are several instances where a lower loss is reached with forward gradients, but they all come from the same instance, i. e. the same underlying loss function.

6.2.1 *A typical supply chain model in Envision*

Our goal is not to understand precisely what model each script is implementing. Nevertheless, it may be of interest to the reader to have a broad understanding of the typical structure of the supply chain problems.

At Lokad, supply chain tasks often amounts to

1. Forecasting, e. g. what will the demand look like next year
2. Produce recommendations for retail prices, stocks, etc.

Autodifferentiation is used only for the first step. We describe briefly the simplest model used, called the *forecaster*.

Suppose our company sells different models of chairs $c_1, \ldots, c_m$. Using the available history of sells, the forecaster learns K *profiles* p_k, with $K \ll m$. A profile is a vector of size 52, which represent demand levels during each week of the year. The forecaster also learns, for each chair model, a trend, a mean level, and a sparse profile-selection vector s of size K. That is, we say that the demand of chair i at time t is

$$\text{trend} \cdot t + \text{level} + s^T p(t)$$

All these coefficients are learned jointly through autodifferentiation.

One significant subtelty is that although the total number of parameters may be very large (as m often is), the loss is only ever computed at a sell history observation, of the form 'x chairs of model c_i were sold at week w'. Such stochastic loss involves only a handful more than $52 \cdot K$ parameters, and this remains true for all other forecasting models used at Lokad.

From the work summarized in section 4.2, this means that the stochastic loss can be considered to have a much lower dimension than the full loss. This is especially relevant for forward gradients which we showed to work better in lower dimensions. While we found production scripts which had as high as 5M parameters, the stochastic loss never exceeded 530 parameters.

6.2.2 *Convergence*

Production scripts are fixed-cost problems, i.e. they are allowed a fixed number of epochs to minimize a loss. In such case, performance profiles as used in section 6.1 are not suitable. Instead, we display an *accuracy profile*, which again is recommended by Beiranvand, Hare, and Lucet[6]. Below we motivate and define what accuracy profiles are.

A natural measure for the quality of an algorithmic output is $f(\bar{x}) - f(x^*)$, where $\bar{x}$ is the best solution found by the algorithm, and x^* is one optimal point. We can "normalize" this measure by the starting accuracy, so that it makes some sense to compare them across problems, with $\frac{f(\bar{x})-f(x^*)}{f(x_0)-f(x^*)}$. Finally, the logarithm of this quantity is generally prefered, which we note $f_{acc}^{p,s}$ for solver s and problem p.

This accuracy forms the basis for accuracy profiles – however to be computed it requires to know the optimal solution, which is of course unavailable in production scripts. Instead, we will replace x^* by the best value found across solvers, and impose an upper bound on f_{acc} to avoid artefacts stemming from the best solver appearing exactly optimal. That is, we define

$$\gamma_{p,s} = \begin{cases} -f_{acc}^{p,s}, & \text{if } -f_{acc}^{p,s} \leqslant M \\ M, & \text{otherwise} \end{cases}$$

and we choose $M = 10$.

Then in the same manner we defined performance profiled, we define the accuracy function for solver s

$$R_s(\tau) = \frac{1}{|\mathcal{P}|}\text{size}\{\gamma_{p,s} \mid \gamma_{p,s} \geqslant \tau,\ p \in \mathcal{P}\}$$

Figure 8 shows such accuracy profiles for the Envision scripts. It gives evidence that forward optimizers reach hardly ever the best loss (they all tend to 0 when the accuracy cut-off tends to M). Moreover, the profile of the AdaBelief optimizer (with true gradients) stay close to 1 for almost all τ. In other words, even when it is not optimal, AdaBelief reaches a good enough approximation of the best loss.

On a side note, even though this was not the goal, our experiments showed that Lokad could prefer AdaBelief to Adam, which is the

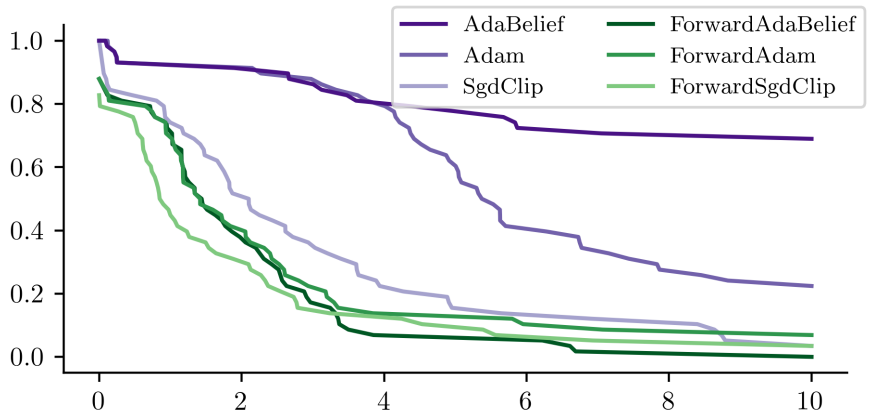


Figure 8: Accuracy profiles for the Envision scripts at Lokad

current default optimizer. In fact, we also observed that not only is Adabelief almost never worse than Adam, it also typically converges faster.

6.2.3 *Computational costs*

Remember our end goal: does forward mode use less memory and run faster than reverse mode ? We measured the computational overhead of both methods, and report here the results.

MEMORY FOOTPRINT Our work presented in subsection 4.1.2 allows to measure a tight upper bound on the memory load of an autodifferentiated piece of code. Figure 9 displays the ratio of memory usage for every production scripts. Although it can be observed that forward mode is never worse than reverse mode, the gains are hardly ever noticeable. The degenerated cases identified in Figure 2 simply do not occur in the wild.

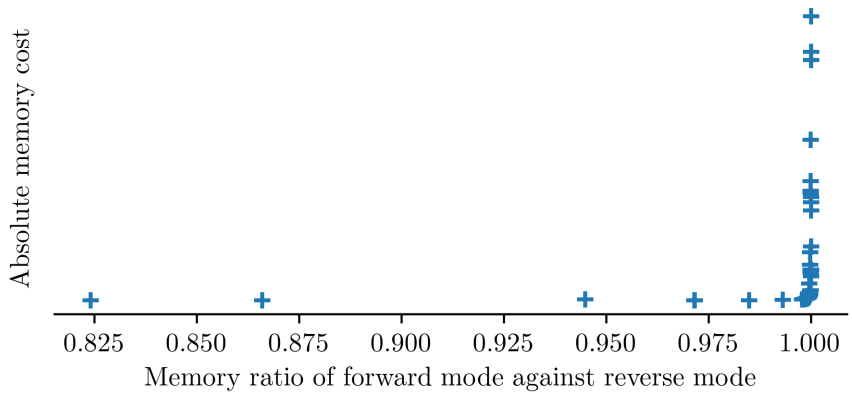


Figure 9: Memory comparison between forward and reverse mode AD.

CPU TIME The same comparison can be done for CPU time, which we display in Figure 10. At first, it appears that forward mode is not consistently faster than reverse mode. There are in fact two factors at play: the execution time for the autodifferentiated loss function, but also the sampling time for the initial tangent, for forward gradients. When the loss function is fast to execute, and its dimension is sufficiently high, the sampling overhead can take over.

This is what happens in the cluster with a time ratio > 1, which actually stems from a single template; meaning that it is only a single model that forward gradients take more time on. Moreover, there are optimization opportunities to sample much faster that we have not implemented – notably, we can trade random number generation time for the quality of randomness, which we do not need to be very high.

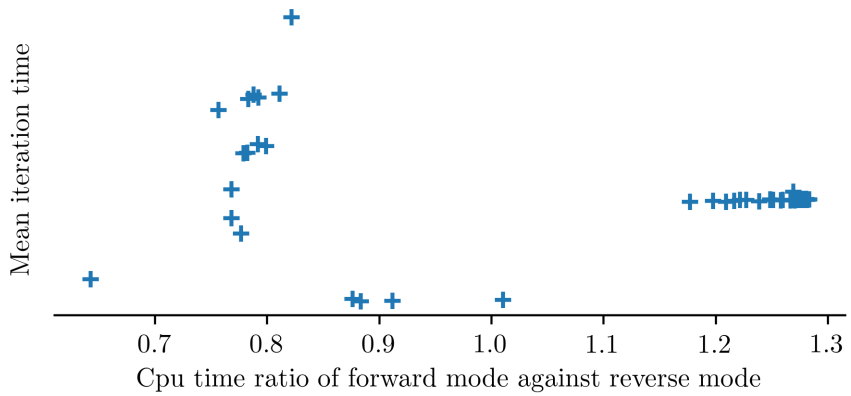


Figure 10: Runtime comparison between forward and reverse mode AD.

6.3 BATCH MODE

We showed, both theoretically, and experimentally, that forward gradients are poor approximators, albeit unbiased, of true gradients. We could sample at each step several forward gradients and use the sample mean instead, to obtain an estimator that is more narrowly centered around the true gradient; but of course this is non sense, as we would lose all computational benefits from using forward mode.

However, this idea can be recycled in conjonction with batch mode: a batch of forward gradients is a better approximators of the batch gradient than a single forward gradient of a single gradient. We get a better approximator for free.

Unfortunately, our experiments on Envision scripts using batches were unconclusive. We identified two shortcomings:

1. Reasonable batch sizes are typically too low (the convergence of the batch mean to the true gradient is proportional to the root of the batch size).

2. More importantly, two observations typically share only a handful of parameters (recall the last paragraph of subsection 6.2.1). In such case, batch mode provides barely any benefits.

Still, we believe that further exploratory work should be conducted. In particular, in the specific case of Envision, SIMD parallelisation may allow to obtain 8 forward gradient samples at each step instead of 1, for free.

7 CONCLUSION

Our work has exposed both theoretical and experimental shortcomings of forward gradients. Specifically, we identified high dimensional settings as an issue, and confirmed this result with extensive experiments.

For Lokad, it is clear that the benefits on the computational load does not justify their use. Worse, their propensity to diverge and general unstability are a bad omen for any future use in an industrial context, where reliability is key.

Still, our work should not be without interest for Lokad, as our experiments showed opportunities to use more efficient optimizers than Adam. Notably, we readily recommend Adam be replaced by AdaBelief.

On the theoretical side, we believe there are still many open roads for future work.

Most notably, we have not conducted any deep learning experiments, where Baydin et al. reported encouraging results for forward gradients. We believe more experiments are needed to evaluate how well they compare to true gradients on more diverse architectures. More importantly, it remains mysterious to us as to why forward gradients would not struggle in such a high dimensional setting as neural networks are, and we trust that further work on this topic would bring a better theoretical understanding.

Alternatively, one could try to design optimizers specifically suited for forward gradients. For instance, the first moment in Adam, which acts as a predictor of what the gradient is going to be, could be used to bias the forward tangent distribution. Although our early experiments did not pan out, we are interested to explore this idea further.

A

DERIVATIONS

A.1 PROOF OF PROPERTY 2

We provide here the proof for the Property 2

Proof.

$$\begin{aligned}\mathbb{E}\left[\|\nabla f(\boldsymbol{\theta}) - g(\boldsymbol{\theta})\|^2\right] &= \sum_i \text{bias}(g_i(\boldsymbol{\theta})) + \mathbb{V}(g_i(\boldsymbol{\theta})) \\ &= \sum_i \mathbb{E}[g_i(\boldsymbol{\theta})^2] - \mathbb{E}[g_i(\boldsymbol{\theta})]^2 \\ &= \sum_i \|\nabla f(\boldsymbol{\theta})\|^2 - \nabla f_i(\boldsymbol{\theta})^2 \\ &= (n-1)\|\nabla f(\boldsymbol{\theta})\|^2\end{aligned}$$

□

A.2 HOW FAR DOES A RANDOM WALK GO ?

LEMMA 2: DIVERGENCE SPEED OF A RANDOM WALK

Let (X_i) be i. i. d. centered Rademacher variables. Then

$$\mathbb{E}\left[|\sum_i^n X_i|\right] = \sqrt{\frac{2n}{\pi}} + O(\frac{1}{\sqrt{n}})$$

Proof. We treat the case where $n = 2N$ is even to alleviate notations. The case of odd n is similar. First we compute

$$\begin{aligned}\sum_{k=0}^{N} \binom{2N}{k} k &= 2N \sum_{k=0}^{N-1} \binom{2N-1}{k} \\ &= 2N4^{N-1}\end{aligned}$$

and

$$\begin{aligned}2\sum_{k=0}^{N-1} \binom{2N}{k} + \binom{2N}{N} &= 4^N \\ \Rightarrow \quad \sum_{k=0}^{N-1} \binom{2N}{k} &= \frac{4^N - \binom{2N}{N}}{2} \\ \Rightarrow \quad \sum_{k=0}^{N} \binom{2N}{k} &= \frac{4^N + \binom{2N}{N}}{2}\end{aligned}$$

Then

$$\begin{aligned}
4^N \mathbb{E}\left[|\sum_i^{2N} X_i|\right] &= \sum_{k=0}^{N-1} \binom{2N}{k}(2N-k-k) + \sum_{k=N+1}^{2N} \binom{2N}{k}(k-(2N-k)) \\
&= 2\sum_{k=0}^{N} \binom{2N}{k}(2N-2k) \\
&= 4N\frac{4^N + \binom{2N}{N}}{2} - 2N4^N \\
&= 2N\binom{2N}{N}
\end{aligned}$$

Only remains now to estimate the asymptotic behaviour with Stirling:

$$\begin{aligned}
\mathbb{E}\left[|\sum_i^{2N} X_i|\right] &= \frac{2N}{4^N}\binom{2N}{N} \\
&\sim \frac{2N}{4^N}\frac{\left(\frac{2N}{e}\right)^{2N}\sqrt{4\pi N}}{\left(\frac{N}{e}\right)^{2N} 2\pi N} \\
&\sim \frac{2N}{\sqrt{N\pi}} = \sqrt{\frac{2n}{\pi}}
\end{aligned}$$

We can get the more precise result from the theorem by using one more term in the Stirling series, which we leave to the reader.

$$n! = \sqrt{2\pi n}(\frac{n}{e})^n \left(1 + O(\frac{1}{n})\right)$$

□

BIBLIOGRAPHY

[1] Francis Bach. *Lecture notes: Statistical machine learning and convex optimization.* 2016. URL: https://www.di.ens.fr/~fbach/orsay2016/lecture2.pdf.

[2] Lukas Balles and Philipp Hennig. "Dissecting Adam: The Sign, Magnitude and Variance of Stochastic Gradients". en. In: *Proceedings of the 35th International Conference on Machine Learning*. ISSN: 2640-3498. PMLR, July 2018, pp. 404–413. URL: https://proceedings.mlr.press/v80/balles18a.html (visited on 07/01/2022).

[3] Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. "Automatic differentiation of algorithms". en. In: *Journal of Computational and Applied Mathematics*. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations 124.1 (Dec. 2000), pp. 171–190. ISSN: 0377-0427. DOI: 10.1016/S0377-0427(00)00422-2. URL: https://www.sciencedirect.com/science/article/pii/S0377042700004222 (visited on 08/12/2022).

[4] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. "Automatic differentiation in machine learning: a survey". In: *arXiv:1502.05767 [cs, stat]* (Feb. 2018). arXiv: 1502.05767. URL: http://arxiv.org/abs/1502.05767 (visited on 05/09/2022).

[5] Atılım Güneş Baydin, Barak A. Pearlmutter, Don Syme, Frank Wood, and Philip Torr. "Gradients without Backpropagation". In: *arXiv:2202.08587 [cs, stat]* (Feb. 2022). arXiv: 2202.08587. URL: http://arxiv.org/abs/2202.08587 (visited on 05/06/2022).

[6] Vahid Beiranvand, Warren Hare, and Yves Lucet. "Best practices for comparing optimization algorithms". In: *Optimization and Engineering* 18.4 (Dec. 2017). arXiv:1709.08242 [math], pp. 815–848. ISSN: 1389-4420, 1573-2924. DOI: 10.1007/s11081-017-9366-1. URL: http://arxiv.org/abs/1709.08242 (visited on 08/16/2022).

[7] Richard Bellman. *Adaptive Control Processes*. Princeton University Press, 1961. ISBN: 978-0-691-07901-1. URL: http://www.jstor.org/stable/j.ctt183ph6v (visited on 08/18/2022).

[8] Clifford. "Preliminary Sketch of Biquaternions". en. In: *Proceedings of the London Mathematical Society* s1-4.1 (1871). _eprint: https://onlinelibrary.wiley.co
4.1.381, pp. 381–395. ISSN: 1460-244X. DOI: 10.1112/plms/s1-4.1.381. URL: https://onlinelibrary.wiley.com/doi/abs/10.1112/plms/s1-4.1.381 (visited on 08/12/2022).

[9] Benjamin Dauvergne and Laurent Hascoët. "The Data-Flow Equations of Checkpointing in Reverse Automatic Differentiation". en. In: *Computational Science – ICCS 2006*. Ed. by David Hutchison et al. Vol. 3994. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 566–573. ISBN: 978-3-540-34385-1 978-3-540-34386-8. DOI: 10.1007/11758549_78. URL: http://link.springer.com/10.1007/11758549_78 (visited on 05/10/2022).

[10] Alexandre Défossez, Léon Bottou, Francis Bach, and Nicolas Usunier. "A Simple Convergence Proof of Adam and Adagrad". In: *arXiv:2003.02395 [cs, stat]* (Oct. 2020). arXiv: 2003.02395. URL: http://arxiv.org/abs/2003.02395 (visited on 05/12/2022).

[11] Michael Innes. *Don't Unroll Adjoint: Differentiating SSA-Form Programs*. arXiv:1810.07951 [cs]. Mar. 2019. URL: http://arxiv.org/abs/1810.07951 (visited on 08/12/2022).

[12] Momin Jamil, Xin-She Yang, and Hans-Jürgen Zepernick. "8 - Test Functions for Global Optimization: A Comprehensive Survey". en. In: *Swarm Intelligence and Bio-Inspired Computation*. Ed. by Xin-She Yang, Zhihua Cui, Renbin Xiao, Amir Hossein Gandomi, and Mehmet Karamanoglu. Oxford: Elsevier, Jan. 2013, pp. 193–222. ISBN: 978-0-12-405163-8. DOI: 10.1016/B978-0-12-405163-8.00008-9. URL: https://www.sciencedirect.com/science/article/pii/B9780124051638000089 (visited on 08/08/2022).

[13] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. arXiv:1412.6980 [cs]. Jan. 2017. URL: http://arxiv.org/abs/1412.6980 (visited on 07/29/2022).

[14] Mykel J Kochenderfer and Tim A Wheeler. "Algorithms for Optimization". en. In: (Mar. 2019), p. 520.

[15] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. "Autograd: Effortless Gradients in Numpy". en. In: (2015), p. 3.

[16] Charles C. Margossian. "A review of automatic differentiation and its efficient implementation". en. In: *WIREs Data Mining and Knowledge Discovery* 9.4 (2019). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1 e1305. ISSN: 1942-4795. DOI: 10.1002/widm.1305. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1305 (visited on 07/29/2022).

[17] Eric Moulines and Francis Bach. "Non-Asymptotic Analysis of Stochastic Approximation Algorithms for Machine Learning". In: *Advances in Neural Information Processing Systems*. Vol. 24. Curran Associates, Inc., 2011. URL: https://papers.nips.cc/paper/2011/hash/40008b9a5380fcacce3976bf7c08af5b-Abstract.html (visited on 08/05/2022).

[18] Y. Nesterov. "A method for solving the convex programming problem with convergence rate O(1/k^2)". en. In: *undefined* (1983). URL: https://www.semanticscholar.org/author/Y.-Nesterov/143676697 (visited on 09/02/2022).

[19] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in PyTorch". en. In: (2017), p. 4.

[20] Paul Peseux. "Programmation différentiable à grande échelle pour les données relationnelles". These en préparation. Normandie, 2020. URL: http://www.theses.fr/s245013 (visited on 08/08/2022).

[21] Paul Peseux, Maxime Berar, Thierry Paquet, and Victor Nicollet. "Stochastic Gradient Descent with gradient estimator for symbolical features". en. In: (2022), p. 31.

[22] Herbert Robbins and Sutton Monro. "A Stochastic Approximation Method". In: *The Annals of Mathematical Statistics* 22.3 (Sept. 1951). Publisher: Institute of Mathematical Statistics, pp. 400–407. ISSN: 0003-4851, 2168-8990. DOI: 10.1214/aoms/1177729586. URL: https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-22/issue-3/A-Stochastic-Approximation-Method/10.1214/aoms/1177729586.full (visited on 09/02/2022).

[23] Jeffrey Mark Siskind and Barak A. Pearlmutter. "Divide-and-Conquer Checkpointing for Arbitrary Programs with No User Annotation". In: *Optimization Methods and Software* 33.4-6 (Nov. 2018). arXiv: 1708.06799, pp. 1288–1330. ISSN: 1055-6788, 1029-4937. DOI: 10.1080/10556788.2018.1459621. URL: http://arxiv.org/abs/1708.06799 (visited on 05/10/2022).

[24] Axel Thevenot. *Optimization & Eye Pleasure: 78 Benchmark Test Functions for Single Objective Optimization*. en. Feb. 2022. URL: https://towardsdatascience.com/optimization-eye-pleasure-78-benchmark-test-functions-for-single-objective-optimization-92e7ed1d1f12 (visited on 08/30/2022).

[25] R. E. Wengert. "A simple automatic derivative evaluation program". In: *Communications of the ACM* 7.8 (1964), pp. 463–464. ISSN: 0001-0782. DOI: 10.1145/355586.364791. URL: https://doi.org/10.1145/355586.364791 (visited on 08/15/2022).

[26] Xin-She Yang. *Test Problems in Optimization*. arXiv:1008.0549 [math]. Aug. 2010. DOI: 10.48550/arXiv.1008.0549. URL: http://arxiv.org/abs/1008.0549 (visited on 08/11/2022).

[27] Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar C Tatikonda, Nicha Dvornek, Xenophon Papademetris, and James Duncan. "AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 18795–18806. URL: https://proceedings.neurips.cc/paper/2020/hash/d9d4f495e875a2e075a1a4a6e1b9770f-Abstract.html (visited on 07/01/2022).

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Final Version as of May 8, 2023 (`classicthesis` version 0.1).