Normandie Université

THÈSE

Pour obtenir le diplôme de doctorat

Spécialité INFORMATIQUE

Préparée au sein de l'université Rouen Normandie

Programmation différentiable pour l'optimisation sur des données relationnelles à grande échelle

Présentée et soutenue par

Paul PESEUX

Thèse dirigée par THIERRY PAQUET
et encadrée par MAXIME BERAR

ed 590 MIIS
Mathématiques, Information, Ingénierie des Systèmes

UNIVERSITÉ DE ROUEN NORMANDIE

litis

Mots clés : données relationnelles, requête relationnelle, programmation différentiable, différentiation automatique, descente de gradient stochastique, données catégorielles

Résumé en Français

Cette thèse de doctorat présente trois contributions dans le domaine de la programmation différentiable axée sur les données relationnelles. Les données relationnelles sont courantes dans des secteurs tels que la santé et la logisitque, où les données sont souvent organisées en tableaux structurés ou bases de données. Les approches traditionnelles de l'apprentissage automatique ont du mal às'appliquer sur de telles données, tandis que les modèles d'apprentissage automatique de type boîte blanche sont plus adaptés mais également plus difficiles à développer.

La programmation différentiable offre une solution en traitant les requêtes sur les bases de données relationnelles comme des programmes différentiables, permettant ainsi le développement de modèles d'apprentissage automatique de type boîte blanche qui peuvent travailler directement sur les données relationnelles. L'objectif principal de cette recherche est d'explorer l'application de l'apprentissage automatique aux données relationnelles en utilisant des techniques de programmation différentiable.

La première contribution de la thèse introduit une couche différentiable dans les langages de programmation relationnelle, autant d'un point de vue théorique que d'un point de vue pratique. Le langage de programmation Adsl a été créé pour effectuer la différentiation et transcrire les opérations relationnelles d'une requête. Le langage Envision a été enrichi d'une couche de programmation différentiable, permettant le développement de modèles exploitant les données relationnelles dans un environnement de langage de programmation relationnelle natif.

La deuxième contribution développe un estimateur de gradient appelé GCE, conçu pour les caractéristiques catégorielles surreprésentées dans les données relationnelles. GCE est démontré comme étant utile sur divers ensembles de données catégorielles et modèles, et a été implémenté pour les modèles d'apprentissage profond. GCE est intégré en tant qu'estimateur de gradient natif dans la couche de programmation différentiable d'Envision, facilité par la première contribution de cette thèse.

La troisième contribution développe un estimateur de gradient généralisé appelé Stochastic Path Automatic Differentiation (SPAD), qui tire sa stochasticité de la décomposition du code. SPAD introduit l'idée de rétro-propager une fraction du gradient pour réduire la consommation de mémoire lors des mises à jour des paramètres. La mise en œuvre de cette approche d'estimation de gradient est rendue possible par les décisions de conception lors de la différentiation d'Adsl.

Cette recherche a des implications significatives pour les industries reposant sur les données relationnelles, en débloquant de nouvelles perspectives et en améliorant la prise de décision en appliquant des modèles d'apprentissage automatique de type boîte blanche aux données relationnelles en utilisant des techniques de programmation différentiable.

Un exposé en 3 minutes et en français de cette thèse est disponible ici[1].

[1]https://youtu.be/oTirPItT5xk

Keywords: relational data, differentiable programming, automatic differentiation, stochastic gradient descent, categorical data

Résumé en Anglais

This PhD thesis, titled presents three contributions to the field of differentiable programming with a focus on relational data. Relational data is prevalent in industries such as healthcare and supply chain, where data is often organized in structured tables or databases. Traditional machine learning approaches struggle with handling relational data, while white box machine learning models are better suited but challenging to develop.

Differentiable programming offers a potential solution by treating queries on relational databases as differentiable programs, enabling the development of white box machine learning models that can directly reason about relational data. This research's primary objective is to explore the application of machine learning to relational data using differentiable programming techniques.

The first contribution of the thesis introduces a differentiable layer into relational programming languages, both theoretically and practically. The Adsl programming language was created to perform differentiation and transcribe relational operations of a query. The domain-specific language Envision has been augmented with differentiable programming capabilities, allowing the development of models that leverage relational data in a native relational programming language environment.

The second contribution develops a novel gradient estimator called GCE, designed for categorical features over represented in relational data. GCE is demonstrated to be useful on various categorical datasets and models and has been implemented for deep learning models. GCE is also integrated as the native gradient estimator in the differentiable programming layer of Envision, facilitated by the first contribution of this thesis.

The third contribution develops a generalized gradient estimator called Stochastic Path Automatic Differentiation (SPAD), which derives its stochasticity from code decomposition. SPAD introduces the idea of backpropagating a fraction of the gradient to reduce memory consumption during parameter updates. The implementation of this gradient estimation approach is made possible by the design decisions during the differentiation of Adsl.

This research has significant implications for industries relying on relational data, unlocking new insights and improving decision-making by applying white box machine learning models to relational data using differentiable programming techniques.

Contents

List of Tables

Introduction

Context

In many domains, the need for machine learning models to handle relational data is becoming increasingly important. This is particularly true in fields such as healthcare or supply chain, where data often comes in the form of structured tables or databases.

This PhD research project is initiated and funded by Lokad with the support of ANRT through a CIFRE contract. Lokad is a French company that specializes in supply chain optimization for businesses. Lokad is engaged in a diverse range of businesses, each with their unique set of supply chain challenges and constraints. They help businesses to forecast their demand, manage their inventory levels, and optimize their ordering and delivery processes. Their software solutions leverage data to create models and algorithms that provide real-time recommendations for businesses to optimize their supply chain operations [2]. To do so, Lokad heavily relies on machine learning models [3] that rely on on historical data to predict the future, which is known as the supervised setting. The data used in supply chain optimization is often relational, meaning that it consists of tables of data that are interconnected in various ways [4, 5]. For example, a business's inventory data might be connected to their sales data, which is in turn connected to their order data, etc. In order to retrieve the desired information, domain experts query these data structures.

Complex and interrelated datasets present a significant challenge when applying conventional machine learning techniques [6,7], such as deep learning, to optimization problems on relational data. Conventional deep learning methods are frequently over parameterized, and fail to properly handle categorical attributes, which can lead to suboptimal solutions for tabular data. Furthermore, these approaches typically involve black-box models, which can be technically challenging to adapt to specific problems. Additionally, most existing machine learning frameworks are not designed for relational data, which makes challenging their application to such databases. Lastly, large deep learning models need extensive resources and highly qualified people to train them, which might be prohibitive especially when the model has to train on new data on a daily basis [8].

In contrast, simple and white-box machine learning models can be tailored for individual problems encountered with relational data. These models can be explicitly designed to capture complex relationships between data points, making them more suitable for relational data. Moreover, white-box models are more interpretable, enabling domain experts to understand the model's inner workings and use that knowledge to make better decisions [9]. Finally, the use of such simple models enables easy daily training, which represents a breakthrough feature for a company like Lokad by making it reactive and constantly up to date. Developing white-box machine learning models for relational data is a demanding task, necessitating a deep understanding of both the domain and underlying data structures [10]. The primary challenge lies in creating tools that facilitate the construction and optimization of white-box models for relational data.

Differentiable programming offers a potential solution to this challenge. By writing differentiable programs on relational data as specific queries, it becomes feasible to build and optimize models by querying directly the relational database, thus embedding the optimization process in the database itself [11] This approach enables the development of white-box machine learning models that leverage the rich relationships between data points within a relational database. Applying white-box machine learning models to relational data has the potential to uncover new insights and enhance decision-making across various critical domains [12].

The primary objective of this research is to investigate the application of machine learning to relational data using differentiable programming techniques.

Relational data and machine learning

Relational data, which is data organized in tables, is a common way to represent complex and structured data in many industries, including healthcare, supply chain, and retail. In these industries, relational data is often the primary form of data that is collected and analyzed, and it plays a critical role in decision-making processes. This data is structured into tables with relationships between them that it is possible to query in order to retrieve the precise information we want.

However, in the modern machine learning research community, the focus has been primarily on unstructured data, such as text, image, and audio data [6]. A possible explanation for that is that unstructured data is often more abundant and easier to collect, while relational data is often more complex to handle and requires specialized tools to process. Additionally, many machine learning researchers come from computer science or related fields where they may not have been exposed to the importance of relational data in many industries. A possible solution to still apply conventional machine learning to relational data would be to transform its structure and group relational data into one raw big table. It might seem like a good idea at first glance because it would provide a unified view of all the data in one place, making it easier to work with. However, this approach has several drawbacks that make it a suboptimal solution [13].

Firstly, relational databases are designed to organize data in a way that minimizes redundancy and maximizes efficiency. By grouping all the data into one big table, we lose the benefits of normalization, which allows us to store data in a more compact and efficient way. This can result in large amounts of duplicated data, leading to increased storage requirements and slower query times. Secondly, working with a single large table can make it more difficult to maintain data integrity and consistency. In a normalized database, we can use the database characteristics to ensure that data is consistent and accurate. With a large table, it can be more difficult to enforce such constraints, leading to potential data quality issues. Finally, working with a large table can be more difficult for data analysts and domain experts to understand and work with. This can limit the ability to build effective models and make informed decisions based on the data. Therefore, while grouping relational data into one raw big table may seem like a tempting solution at first, it is not a good idea in practice due to the potential issues it can create. Instead, it is better to organize the data into normalized tables that can be efficiently queried and maintained, while also providing a clear and intuitive view of the data for analysts and domain experts.

Differentiable programming

Differentiable programming is a programming paradigm that has made machine learning so popular, and has allowed the advent of deep learning as well [1,14,15]. The main idea behind differentiable programming is that all functions that make up a model are differentiable, which allows for end-to-end optimization using gradient-based methods [16]. While programming deals with programs that are not directly differentiable, they implement functions that are. A method to differentiate a program is presented in Figure 1. Differentiable programming combines traditional programming with automatic differentiation, allowing the construction of mathematical models that can be optimized using gradient-based methods. Automatic differentiation works by decomposing complex functions into elementary operations and applying the chain rule [17,18] to calculate exact gradients [19].

In machine learning, differentiable programming is essential for production-level systems as it allows domain experts to highlight their knowledge in a way that can be integrated directly into the optimization process [20,21]. In traditional machine learning approaches, domain experts would use their knowledge to create features that would be fed into a model. However, this process is often time-consuming and can be limited by the creativity and expertise of the domain expert. In contrast, differentiable programming enables domain experts to directly incorporate their knowledge into the optimization process without having to manually compute the derivatives of their models.

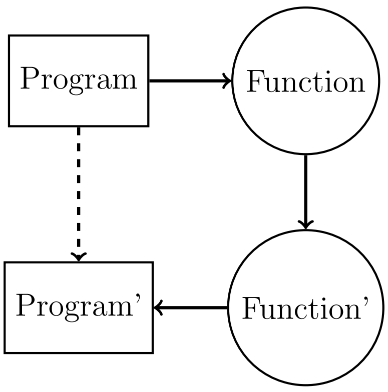


Figure 1.: Differentiable programming. A program implements a function, the derivative of which is mathematically defined. Consequently, the derivative of the program represents the program that corresponds to the function's derivative.

One of the key advantages of differentiable programming is that it allows to build complex models by composing simple building blocks [22] as it relies on automatic differentiation. The composition of these building blocks creates a model that can be optimized end-to-end using gradient-based methods. This approach allows us to build highly flexible and customizable models that can be adapted to a wide range of tasks and datasets. Domain experts can contribute their knowledge by designing differentiable building blocks that are specific to their field of expertise. These building blocks can then be easily integrated into a larger model using differentiable programming. This can lead to significant improvements in model performance and can also help to identify potential problems with the model, solvable by fixing the corresponding block. Differentiable programming can be viewed as an approach to adapt the model to the data, as opposed to altering the data to conform to the requirements of a predetermined and inflexible model [23].

Furthermore, differentiable programming allows for a more seamless integration between model development and production deployment. Since the differentiable program can be optimized using gradient-based methods, it can be easily incorporated into a larger system without the need for separate optimization and deployment stages.

In summary, differentiable programming is key in machine learning production as it allows domain experts to directly encode their knowledge into the optimization process, leading to improved model performance and a more seamless integration between development and deployment. It represents a paradigm shift in how machine learning systems are developed and deployed, enabling a tighter integration between domain expertise and machine learning algorithms.

If differentiable programming were to be implemented in relational languages, numerous challenges would arise. First, gradient-based methods are typically applied to numerical values, whereas relational data often contains categorical values. Special attention must be given to such data types while performing gradient descent on white-box models. Second, careful consideration must be devoted to the computational resources used while optimizing models created with differentiable programming tools. Although memory consumption for querying machine learning models is generally limited, optimizing them during the training phase can be computationally intensive. Efficient techniques, such as checkpointing, already exist [24]. This variable storage heuristic can result in significant memory savings, albeit with an increased computational time. By leveraging the construction of gradients facilitated by automatic differentiation systems in relational programming languages, we could develop a novel gradient estimator that requires less memory. Consequently, this would enable the execution of the same task more rapidly or facilitate the processing of larger datasets.

The aforementioned points highlight the need for differentiable tools in database systems and the importance of developing efficient machine learning approaches for relational data. Consequently, the title of this PhD thesis is:

A Differentiable Programming Approach for Optimization on Relational and Large Datasets.

Contributions

The first contribution of this PhD thesis is the introduction of a differentiable layer into relational programming languages, which is both theoretical and practical. The theoretical work involved in this contribution includes properly defining differentiation on relational queries, which requires the introduction of two main concepts: the PolyStar and the TOTAL JOIN operator. The PolyStar is a specific way of viewing the tree made up of the tables used in the query to be differentiated, which allows us to load only the minimum amount of data for each observation. The TOTAL JOIN operator is a novel join operator that enables us to construct the correct PolyStar relative to the query. These two concepts facilitate the derivation of a query by an automatic differentiation tool. To achieve this, we created our own programming language called Adsl, which is designed to perform differentiation and easily transcribe the relational operations of a query. Importantly, the derivative of a query can also be expressed as a query using

Adsl and its automatic differentiation system. The relational domain-specific language of Lokad called Envision has been augmented with differentiable programming capabilities through the design, differentiation, and implementation of Adsl. Envision is engineered by Lokad for the specific purpose of the predictive optimization of supply chains. Multiple models leveraging the relational aspect of data have been developed and demonstrated in a native relational programming language environment and ended up in production on a daily basis at Lokad.

The second contribution of this PhD thesis is the development of a novel gradient estimator, named GCE, that is designed for categorical features that are over represented in relational databases. This contribution was motivated by the underperformance of deep learning methods on relational data due to incorrect handling of such data in gradient estimation. The idea that "a non-existing gradient is not a zero gradient" led to the creation of GCE, which we demonstrate to be useful on several diverse categorical datasets and models. We provide an implementation for deep learning models, as well. Furthermore, GCE is integrated as the native gradient estimator in the differentiable programming layer of Envision, which was made possible by the first contribution of this PhD thesis.

The third contribution of this PhD thesis is the development of a generalized gradient estimator, where the stochasticity is derived from the code decomposition. Contrary to the conventional view of stochastic gradient descent, which focuses on observations and batches, we suggest backpropagating a fraction of the gradient to reduce memory consumption during parameter updates. We introduce Stochastic Path Automatic Differentiation (SPAD), a novel technique that can be regarded as a combination of dropout and layer freezing within neural networks. The implementation of this gradient estimation approach is also facilitated by the design decisions made during the differentiation of Adsl.

Organization of the manuscript

This manuscript is separated into four chapters. Chapter I presents the first contribution of this PhD thesis, i.e. the process of differentiating relational queries. We begin by discussing automatic differentiation foundations and its multiple implementation over widespread programming languages. Next, we delve into the fundamentals of relational queries, including relational algebra and query structures. Then, we propose a novel approach to differentiating relational queries by decomposing the query into its mathematical and its relational components. We then introduce Adsl, a dedicated programming language designed for this purpose, along with its key concepts and automatic differentiation capabilities. Adsl is a programming language specifically designed for automatic differentiation, exhibiting two key properties: each variable is assigned only once and is also accessed for reading exactly once. Finally, we discuss the implementation of Adsl, its inclusion in Envision, a domain-specific language, and showcase examples of how this approach can be applied in both toy and production settings. We end by providing mathematical insights on the kind of model that can be created from relational query. We emphasize the scan operator as an essential component of our tool of differentiable queries.

Chapter II presents an overview of stochastic gradient descent. We start by examining the basic concept of gradient descent, along with its analogy, notations, and convergence proof in smooth and convex cases. We also address its limitations, providing the motivation for incorporating stochasticity into the optimization process. Subsequently, we discuss various optimizers, including Vanilla, Adagrad, and Adam, and their individual properties. Next, we investigate the convergence guarantees associated with a generalized adaptive optimizer, shedding light on its effectiveness in optimization problems. Finally, we explore how stochasticity can be obtained from relational data and PolyStars.

Chapter III presents the second contribution of this PhD thesis, i.e. the gradient estimator for categorical features GCE. We begin by examining learning with categorical data, categorical models and one-hot-encoding. We also address the problems associated with stochastic gradient descent in the context of categorical features and the convergence guarantees of this technique. Next, we introduce a solution for gradient estimation in the form of GCE. We provide a formal definition of GCE, followed by a proof of its unbiasedness, and discuss its application to relational linear regression. GCE is based on the observation that not all categorical features appear in every row of a dataset. Consequently, the corresponding parameters should not be updated during every iteration. We also present experimental results and real-life applications of GCE, covering its use in deep learning, categorical models on public datasets, and production settings. Lastly, we explore the initialization of categorical and multiplicative models, starting with a two-feature example and extending the discussion to generalizations using Singular Value Decomposition.

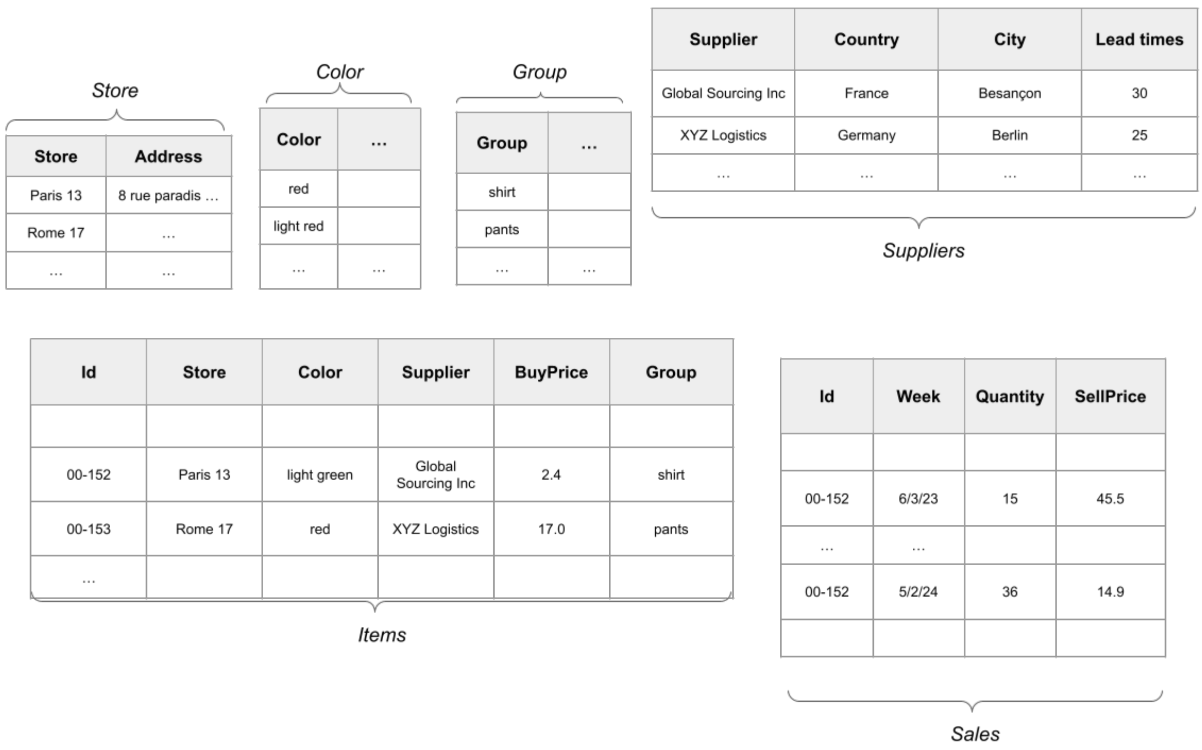
Chapter IV presents the third contribution of this PhD thesis, i.e. a gradient estimator based on code stochasticity. We also cover overfitting and memory consumption. We will start by discussing memory consumption in the context of gradient-based methods, including checkpointing techniques and Adsl's considerations on their selection. We will also discuss embedded artificial intelligence and its implications for memory management. Next, we will address the issue of overfitting the data and introduce the dropout technique as a solution to mitigate this issue. We examine the possibility to estimate the gradient by sampling random backpropagation paths. Moving beyond uniform distribution on backpropagation paths, we introduce SPAD and discuss its implementation generalization. We will also consider Adsl's approach to incorporating SPAD. Finally, in the last section, we present experiments that demonstrate the effectiveness of these techniques. We explore optimization functions and deep learning models to showcase the benefits of the methods discussed throughout the chapter.

In the conclusion of this doctoral research project, we provide a summary of the key results obtained. We reiterate our contributions that facilitated differentiation in relational queries. Furthermore, we recapitulate the two innovative gradient estimators introduced in this research: GCE and SPAD. Lastly, we discuss the potential avenues for future research in this area, highlighting the exciting prospects that this thesis has unveiled.

I. Differentiating relational queries

Introduction

In today's world, data recording and management play a crucial role across various domains, encompassing heterogeneous data types such as images, sounds, texts, and physical measurements. Among these diverse data types, relational data holds particular significance in domains like healthcare and supply chain management. For example, in these domains, patients may exhibit multiple health problems or take incompatible medications, while customers may purchase items from multiple providers. A retail supply chain database is given as an example in Figure I.1. To handle the inherent structure of relational data, theoretical and practical frameworks have been developed, and database systems are commonly used to query these structures and retrieve essential information.



Store

Store	Address
Paris 13	8 rue paradis ...
Rome 17	...
...	...

Color

Color	...
red	
light red	
...	...

Group

Group	...
shirt	
pants	
...	...

Suppliers

Supplier	Country	City	Lead times
Global Sourcing Inc	France	Besançon	30
XYZ Logistics	Germany	Berlin	25
...	...	...	...

Items

Id	Store	Color	Supplier	BuyPrice	Group
00-152	Paris 13	light green	Global Sourcing Inc	2.4	shirt
00-153	Rome 17	red	XYZ Logistics	17.0	pants
...					

Sales

Id	Week	Quantity	SellPrice
00-152	6/3/23	15	45.5
...	...		
00-152	5/2/24	36	14.9

Figure I.1.: Illustration of a retail supply chain database showing tables with shared attributes facilitating relationships between them.

With the advent of machine learning, experts from diverse fields are keen to harness this technology to tackle vital tasks like regression and classification, which hold great significance in healthcare and supply chain management. In order to demonstrate the potential of machine learning in relational programming languages, we present a straightforward example of a categorical model called relational linear regression. This model applies to data from Table I.1 and extends traditional linear regression by sharing the slope among all observations with the same category, while the intercept remains shared among all observations.

Category	Slopes
A	a_A
B	a_B
C	a_C

Cat table

Id	Category	x	y
01	B	1.1	3.4
02	A	2.4	2.7
03	A	1.6	1.4
04	B	3.7	9.7
05	C	4.2	5.9
...	...	...	...

Obs table

Table I.1.: Tables storing data. The Category attributes of the Obs and the Cat tables relate to the same information.

$$\hat{y}(cat, x) = a_{cat} \times x + b. \qquad (I.1)$$

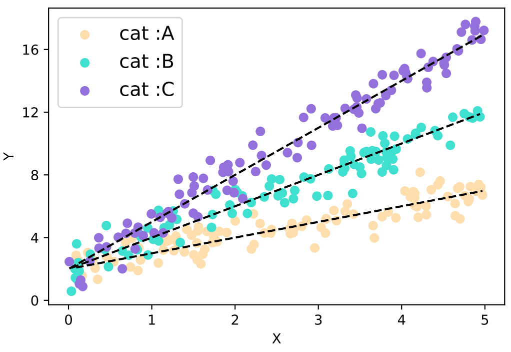


Figure I.2.: Relational linear regression applied with the cat attribute consisting of three different values: A, B or C.

For a continuous variable x predicting another one y, a linear regression has 2 parameters, while a relational linear regression of a categorical variable has $1 + n_s$ parameters, with n_s the cardinality of the possible attributes. Instead of only modeling the relationship between two variables, the underlying structure of the data is used to provide a more accurate representation of the input variables. Equation I.1 formalizes it while Figure I.2 represents a toy dataset where the relational linear regression with slopes shared by category fits the data very well. Such a simple model embraces the relational aspect of the data and is interpretable, every parameter has an understandable meaning. This inherently categorical model will serve as an example of what we aim to achieve through the differentiation of queries. We also present a more sophisticated model, which is in production at Lokad. Lokad faced the challenge of developing a retail forecasting model for one of its clients. The goal is to forecast item-level sales $y(item, week)$ for each week using a sales history of over 70 billion observations. For each item in the dataset, there are multiple categorical variables like the color, the store ... Lokad's experts crafted a categorical model, similar to the following equation[1] for each item i in the dataset:

[1]actual model details are kept confidential

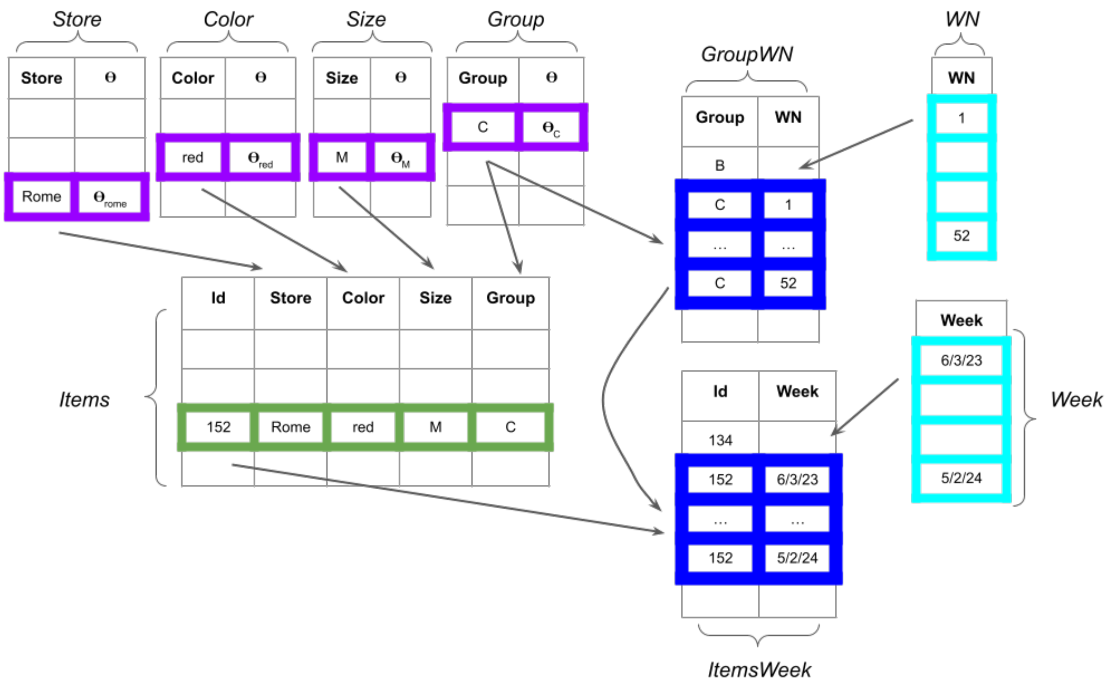
$$\hat{y}(i, week) = \theta_{store(i)} \times \theta_{color(i)} \times \theta_{size(i)} \times \Theta[group(i), WeekNumber(week)]. \tag{I.2}$$

Here, $\Theta[group(i), WeekNumber(week)]$ is a parameter vector that captures the annual seasonality for a specific group of items. It can be seen as a function mapping from the combination of groups and week numbers to real numbers:

$$\Theta : Groups \times [|1, 52|] \longrightarrow \mathbb{R}.$$

In this model, the values of the parameters $\theta_{store(item)}, \theta_{color(item)}, \theta_{size(item)}$ are determined by the category of the item. As a result, two *red* items will have the same value for $\theta_{color(item)}$. Once the model parameters ($Color.\theta$, $Store.\theta$, etc.) are defined and stored in the appropriate tables, it is straightforward to compute the sales estimation $\hat{y}(item, week)$ using any relational programming language. Tables can be easily queried, ensuring that each item has a unique corresponding color and, in turn, a unique θ_{color}. The relationships between these tables is presented in Figure I. However, relational languages lack a fully integrated automatic differentiation capability or operator, preventing in-database optimization of such models through gradient descent. The objective of this chapter is to establish a theoretical and a practical framework that enables the creation and optimization of these models within relational programming languages.

More generally, there is currently no machine learning tool in database systems. This lack is problematic for several reasons. First, the data has to be transferred on frameworks that are not designed for relational data in order to perform machine learning on it. Then the output has to be transferred back to the database system which is costly and error prone [11]. Second, the machine learning model design responsibility goes to a machine learning expert who has less understanding of the problem than the domain expert. If a very complicated model like Transformers [25] is needed to perform the task, this responsibility transfer is mandatory but if a simple-but-well-designed model is enough, the domain expert is the appropriate person to do it. Such simple-but-well-designed models can be created by a domain expert even if he does not need to be involved in the optimization process of it. And this can be obtained thanks to differentiable programming, a paradigm in which a program can be differentiated. This derivative program gives direct access to gradient-based methods, detailed in Chapter II. To enable feasible machine learning in database systems, we have developed a dedicated programming language: Adsl, which is specifically designed for this purpose. Differentiating relational queries is now possible thanks to Adsl and the introduction of PolyStar and TOTAL JOIN, which provide a solid framework for this task.



Example of the relation between tables of the Lokad sales forecasting challenge

Example of the relation between tables of the Lokad sales forecasting challenge

Figure I.3.: Example of the relation between tables of the Lokad sales forecasting challenge

This chapter is organized as follows. First, we present differentiation techniques and implementations that allow gradient-based optimization. Second, we present the relational algebra theory to properly define how we aim to handle relational data. We also motivate our objective to perform machine learning on relational programming languages through differentiable programming and we observe the lack of appropriate tools to do it. The rest of the chapter is contributional. We describe our main approach to unlock differentiable programming on relational programming languages. Thus we introduce Adsl, a sub-language especially crafted for automatic differentiation on relational data. Naturally we also introduce its differentiation. Finally we present the practical application of differentiable programming on relational data through Envision, a domain specific language for supply chain. We illustrate it by designing different models that strongly take into account the relational structure of the data.

The main contributions of this chapter are the following. We have unlocked differentiable programming on relational programming languages. To do so we introduce the notion of PolyStar to carefully describe the relationship between the tables used in a query. We also introduce the TOTAL JOIN operator which lets us carefully construct our query to differentiate. We also design Adsl, A Differentiable Sub Language for differentiation on relational data, we implement it through Envision that is the first relational programming language enabling differentiable programming.

I.1. Differentiation

Machine learning has led to significant advances in recent decades, with the development of various model structures capable of achieving high performance on tasks such as classification, numerical regression, policy learning, and data generation [26–29]. The success of these models is measured by their ability to minimize a specific function known as a loss function. Optimization techniques, such as gradient descent, are often used to minimize the loss function by adjusting the model's parameters. Gradient descent works by following the opposite direction of the gradient in the hopes of decreasing the loss function step by step. Deep neural networks, which are a popular class of machine learning models, are optimized using gradient descent. The details of gradient descent are presented in Chapter II. However, prior to discussing the minimization process using gradient descent, it is essential to explain how this gradient can be obtained, which is done below.

I.1.1. Differentiation techniques

The derivative of a function represents the rate of change of that function at any given point. It provides information about how the function behaves locally, including whether it is increasing or decreasing, and the steepness of the curve at a particular point. The derivative of a scalar function f from $\mathbb{R} \longrightarrow \mathbb{R}$ is:

$$f'(x) = \frac{\partial f}{\partial x} = \lim_{h \longrightarrow 0} \frac{f(x+h) - f(x)}{h}. \tag{I.3}$$

It generalizes to higher dimension on $f : \mathbb{R}^p \longrightarrow \mathbb{R}^q$ and is thus called gradient $\nabla f_i \in \mathbb{R}^p$ for $i \leq q$. The list of all the gradients of a function are represented into the matrix J_f from Equation I.4. J_f is called the Jacobian matrix.

$$J_f(x) = \begin{bmatrix} {}^T\nabla f_1(x) \\ \cdots \\ {}^T\nabla f_q(x) \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_p} \\ \cdots & \cdots & \cdots \\ \frac{\partial f_q}{\partial x_1} & \cdots & \frac{\partial f_q}{\partial x_p} \end{bmatrix}, \tag{I.4}$$

where $A \mapsto {}^T A$ is the transposition operator.

There are three possibilities in order to compute the gradient and thus use these optimisation techniques: **Manual** or **Numerical** or **Automatic** differentiation.

Manual differentiation

The most basic way is to manually code the gradients of each function. One can craft the derivative by hand by following the chain rule and then code it. An example is given in Figure I.4. This is very costly in terms of human time and it is error prone. Hand coded differentiation has been a chosen approach and might be relevant in very specific cases in order to fasten the gradient execution of a given function [30]. It is also useful if for specific reasons, the users do not want to work with the true gradient of its defined function but with a modified one.

$$f(x) = \cos(x^2)$$
$$f'(x) = 2x \cos'(x^2) = - 2x \sin(x^2)$$

Figure I.4.: Manual Differentiation of $\cos x^2$ as taught in high school.

Numerical differentiation

Numerical differentiation is a method for approximating the derivative of a function at a particular point. It involves calculating the slope of the function over a small interval surrounding the point of interest, which is illustrated in Figure I.5.

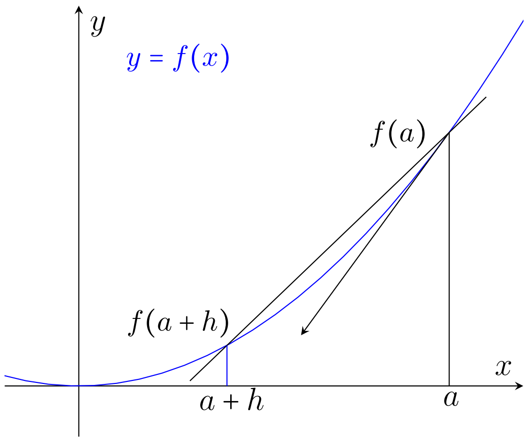


Figure I.5.: Illustration of numerical differentiation of f. The arrow starting from $(a, f(a))$ represents the tangent of the gradient of f at a, oriented in order to decrease f.

The most common methods for numerical differentiation are the finite difference method [31], which involves subtracting the function values at two points and dividing the result by the difference in their independent variable. These approximations become more accurate as the interval becomes smaller. Formula I.5 and I.6 respectively present the forward difference and the central difference methods.

$$[J_f(a)]_{i,j} \approx \frac{f_i(a + h_j e_j) - f_i(a)}{h_j} \tag{I.5}$$
$$[J_f(a)]_{i,j} \approx \frac{f_i(a + h_j e_j) - f_i(a - h_j e_j)}{2h_j}, \tag{I.6}$$

with e_j the unit vector in the j^{th} direction and h_j a step size. The forward differences need $p(q+1)$ evaluations of the function f while the central differences require $2pq$ ones. However, numerical differentiation can also be prone to errors and instability. First, the accuracy of numerical differentiation depends on the step size and the method used. For example, the finite difference method can introduce errors due to round off and truncation. Second, it is sensitive to the choice of step size, and the results can be highly dependent on the choice of the method and the behavior of the function being differentiated [32].

Third, numerical differentiation can be computationally expensive, especially for high-dimensional functions or for functions with multiple local extrema.

We now present another differentiation technique: automatic differentiation which is the foundation of differentiable programming.

I.1.2. Automatic differentiation

Automatic differentiation [19, 33] is a computational technique used to exactly and efficiently evaluate derivatives of functions expressed as computer programs. It computes derivatives by breaking down complex functions into elementary operations and applying the chain rule. The chain rule is a fundamental concept in calculus that governs the differentiation of composite functions. It states that if we have two functions f and g both from $\mathbb{R}$ to $\mathbb{R}$, one denotes $y = f \circ g$, where $g(x)$ represents an inner function and $f(u)$ represents an outer function, then the derivative of y with respect to x can be computed by multiplying the derivative of $f(u)$ with respect to u by the derivative of $g(x)$ with respect to x. Mathematically, the chain rule is expressed as:

$$\frac{\partial y}{\partial x} = \frac{\partial f}{\partial u} \times \frac{\partial g}{\partial x}. \tag{I.7}$$

This also applies to higher dimensions. When $g : \mathbb{R}^a \longrightarrow \mathbb{R}^b$ and $f : \mathbb{R}^b \longrightarrow \mathbb{R}^c$, then chain rule applies to the jacobians matrices $J_f \in \mathbb{R}^{c,b}$, $J_g \in \mathbb{R}^{b,a}$:

$$J_{f \circ g}(x) = J_f(g(x)) \times J_g(x). \tag{I.8}$$

The chain rule based on matrix multiplication applies irrespective of the complexity of the model, enabling differentiation of both simple linear regression and deep neural networks.

There are two main approaches to automatic differentiation: forward mode and reverse mode. Forward mode calculates derivatives with respect to each input variable in a single pass, while reverse mode computes the gradient of the output with respect to all input variables simultaneously. Reverse mode, also known as backpropagation in the context of neural networks, is widely used in machine learning and deep learning applications for optimizing model parameters values.

One can also think about Symbolic differentiation, which is a method that involves algebraic manipulation of a function's mathematical expression in order to obtain an exact representation of its derivative. However this can be considered as equivalent to automatic differentiation [34] (even if there is still debate on the subject [1]).

Automatic differentiation is applied on an intermediate representation of a program introduced as the Wengert lists.

Wengert lists

A Wengert list [19] is a mathematical construct used in the field of automatic differentiation. It is a list of all the operations performed during a computation, along with the variables that participate in those operations. In other words it is the trace of an execution of a program.

It takes the form of a list of statements. The only present statements are assignment statements to variables, called Wengert variables. It is as basic as it can be. All loops are unrolled and every Wengert variable is scalar. There are no conditional statements, it just keeps the branch chosen at the execution. Here's an example of a Wengert list for a computation that includes a conditional statement. Let's consider pseudo code from Listing I.1

```
x <- ...
if x>0
    then y <- x
    else y <- -x
z <- y + 1
```

Listing I.1: Pseudo code of a program containing a conditional.

The corresponding Wengert list for this computation depends on the value of the inputs, notably to determine which conditional branch to choose. It would be as follows in Listing I.1.

```
x = -3          (Input variable)
a = 0           (Input variable)
c = (x > a)     (Intermediate variable representing the condition)
w1 = - x        (Intermediate variable)
z = w1 + 1      (Output variable)
```

Listing I.2: Wengert list of a program containing a conditional. A Wengert list being the execution trace of the program, it requires a value for its input. We have arbitrarily chosen $x = -3$.

In this example, x is the input variable, z is the final output variable, c, and $w1$ are intermediate variables that store the results of intermediate computations. The Wengert list includes an additional intermediate variable c to represent the condition $x > 0$, which determines the rest of the list even though it seems unused in the following. The variable z is computed using the conditional branch of the current execution. With another value for input x, the trace of the execution would not be the same.

We present the two main approaches to perform automatic differentiation on a program. Automatic differentiation on Wengert lists computes the gradient of the output with respect to the input variables. Automatic differentiation can be implemented in two modes: reverse or forward [35]. Both rely on the chain rule, depicted in Equation I.7, but they do not run through it in the same direction.

In the subsequent sections, we present a formalization of the two distinct approaches for applying the chain rule to differentiate a program, known as reverse and forward modes. We begin by discussing the reverse mode.

Reverse mode

In the reverse mode of automatic differentiation we start from an initial cotangent $u \in \mathbb{R}$ in the output space. For an intermediary Wengert variable ω, its adjoint is defined as:

$$\overline{\omega} = \frac{\partial f}{\partial \omega}.u, \tag{I.9}$$

$\overline{\omega}$ can be seen as the sensitivity of the output $f(x) \in \mathbb{R}$ in the cotangent direction with respect to ω. One can choose 1 as an initial cotangent which simplifies to $\overline{\omega} = \frac{\partial f}{\partial \omega}$. For the input parameters, the choice of $u = 1$ gives us what we are looking for. An illustrating example is given on Listing I.3.

Let's note $\Omega_L = (\omega_1, \dots, \omega_L)$ the intermediary Wengert variable that directly use ω as input of intermediary functions $(f_1 \dots f_L)$:

$$\Omega_L = (\omega_1, \dots, \omega_L) = (f_1(\omega, \dots), \dots, f_L(\omega, \dots)).$$

Let's note $H : \mathbb{R}^L \longrightarrow \mathbb{R}$ such that $f(\omega) = H(\Omega_L)$. Then the chain rule gives us:

$$\overline{\omega} = \frac{\partial f}{\partial \omega}.u = \frac{\partial H(\Omega_L)}{\partial \omega}.u = \underbrace{{}^T(\nabla_{\Omega_L} H)}_{\in\mathbb{R}^{1,p}} \underbrace{\frac{\partial \Omega_L}{\partial \omega}}_{\in\mathbb{R}^{p,1}}.u = ({}^T(\nabla_{\Omega_L} H).u)\frac{\partial \Omega_L}{\partial \omega} = \sum_{l=1}^{L} \overline{\omega_l}\frac{\partial \omega_l}{\partial \omega} = \sum_{l=1}^{L} \overline{\omega_l}\frac{\partial f_l}{\partial \omega}.$$

This is called reverse accumulation as the adjoint of ω accumulates all the incoming adjoints of its children in the execution graph. One can notice that the expression of $\overline{\omega}$ uses the $\frac{\partial f_l}{\partial \omega}$: the inputs of the f_l are thus needed. Thus the Wengert list of the adjoint program starts by the statements of the original one to compute these inputs. The storage of the intermediate values is not tackled by the Wengert list representation but this is detailed in Section IV.1.1.

```
x = −3              (Input variable)
a = 0               (Input variable)
c = (x > a)         (Intermediate variable representing the condition)
w1 = − x            (Intermediate variable)
z = w1 + 1
z̄ = 1               (Initial cotangent)
w̄1 = z̄              (Chain rule)
x̄ = w̄1 ∗ (−1)  (Output variable)
```

Listing I.3: Reverse mode automatic differentiation of the Wengert list. In blue, the additional statements to obtain the gradient.

Forward mode

In the forward mode of automatic differentiation, we start from an initial tangent $v \in \mathbb{R}$ and the input $x \in \mathbb{R}$ both in the input space. For an intermediary Wengert variable ω that is formally computed from the input x by f_ω ($\omega = f_\omega(x)$), the tangent of ω is defined as:

$$\dot{\omega} = \frac{\partial \omega}{\partial x}.v.$$

Let's decompose the target function as $f_\omega = H_\Omega \circ H_x$ with $H_x : \mathbb{R} \longrightarrow \mathbb{R}^L$ and $H_\Omega : \mathbb{R}^L \longrightarrow \mathbb{R}$. Let's note $H_x(x) = \Omega_L = (\omega_1, \dots, \omega_L)$.

$$\begin{aligned}\dot{\omega} &= \frac{\partial \omega}{\partial x}.v = \frac{\partial f_\omega}{\partial x}.v = \frac{\partial H_\Omega \circ H_x}{\partial x}.v \\ &= \frac{\partial H_\Omega}{\partial H_x} \circ H_x.\frac{\partial H_x}{\partial x}.v \\ &=^T (\frac{\partial \omega}{\partial \omega_1}, \dots, \frac{\partial \omega}{\partial \omega_L}).(\dot{\omega_1}, \dots, \dot{\omega_L}) \\ &= \sum_{l=1}^{L} \frac{\partial \omega}{\partial \omega_l} \dot{\omega_l}.\end{aligned}$$

This is called forward accumulation and is the equivalence of reverse accumulation but for forward mode instead. An example on Wengert lists can be found in Listing I.4.

```
x = ...          (Input variable)
$\dot{x} = \frac{\partial x}{\partial x} = 1$
a = 0            (Input variable)
$\dot{a} = 0$
c = (x > a)
$\dot{c} = 0$
w1 = - x
$\dot{w1} = -\dot{x}$
z = w1 + 1
$\dot{z} = \dot{w1}$          (Output variable)
```

Listing I.4: Forward mode automatic differentiation of the Wengert list. In blue, the additional statements to obtain the gradient. In contrast to reverse mode, the additional statements are intertwined with the original ones.

Higher dimensions and mode choice

In a more general case, we consider $f : \mathbb{R}^p \longrightarrow \mathbb{R}^q$. Then the same rules apply for both reverse and forward mode but with matrices and vector multiplication rather than only scalar chain rule.

In reverse mode the objective is to compute the vector Jacobian product (VJP) $({}^T J_f(x))u$, $u \in \mathbb{R}^q$ being the cotangent. The traditional orthogonal basis $\{e_i\}_{i \leq q}$ of $\mathbb{R}^q$ is often employed as initial cotangents:

$$\forall i \leq q, \quad ({}^T J_f(x))e_i =^T [\frac{\partial f_1}{\partial x_i} \dots \frac{\partial f_q}{\partial x_i}] =^T \nabla f_i(x)$$

This computation can be achieved in a single pass for each cotangent. However, if the goal is to compute the entire Jacobian matrix instead of just the VJP, which would require q passes, with q different initial cotangents.

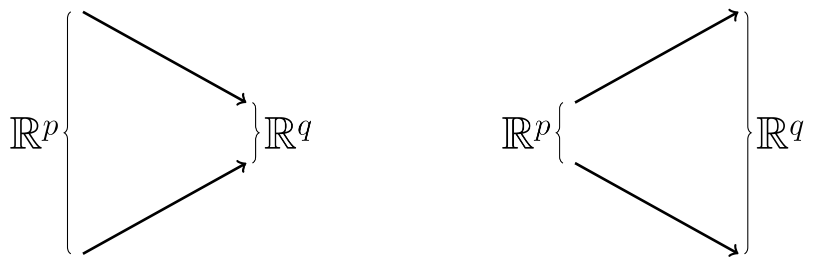
In forward mode the objective is to compute the Jacobian vector product (JVP) $J_f(x)v$, $v \in \mathbb{R}^p$ being the cotangent. The traditional orthogonal basis $\{e_j\}_{j \leq p}$ of $\mathbb{R}^p$ is often employed:

$$\forall j \leq p, \quad J_f(x)e_j =^T [\frac{\partial f_1}{\partial x_j} \dots \frac{\partial f_q}{\partial x_j}] =^T \nabla f_j(x)$$

This computation can be achieved in a single pass. However, if the goal is to compute the entire Jacobian matrix instead of just the JVP, which would require p passes, with p different initial tangents.

In the problem we are addressing, which is optimization in general, we primarily encounter the scenario where $q = 1$. This is a fundamental characteristic of minimization, as it allows us to minimize scalar quantities but not vectors. Specifically, when q equals 1, the vector space $\mathbb{R}^q$ is reduced to a one-dimensional space, which is the only case where a total order can be established.

In such cases, the reverse mode requires only one pass, whereas the forward mode requires p passes with p tangents. Consequently, when p is significantly larger than q, the forward mode becomes infeasible in practice.



(a) Reverse mode suited (b) Forward mode suited

Figure I.6.: Automatic differentiation mode choice depends on the data dimension.

In the following we will focus on the automatic differentiation as it does not require any human intervention and is exact: it gives the program that implements *exactly* the gradient of a function.

Remark 1. *In the following we assume that all the considered functions are differentiable, even though differentiation is still possible in a wider area [36].*

Remark 2. *In this chapter, our focus is solely on differentiation, and we do not delve into the development of gradient descent, which is enabled by this technique. The theory of gradient descent is presented in Chapter II, and it is extensively employed in Chapters III and IV.*

I.1.3. Existing automatic differentiation systems

To facilitate the implementation of gradient-based algorithms, automatic differentiation is crucial. Many subsets of programming languages can be automatically differentiated, as it is a widely studied subject [1, 37–39]. The most well-known libraries that rely on automatic differentiation include PyTorch [40] and TensorFlow [41].

There are two main approaches for implementing automatic differentiation. The first approach is to take an existing programming language and try to make it differentiable [37–40, 67]. In this method, the language is modified for optimization through gradient descent. However, by doing so one has to implement the automatic differentiation system for each programming language separately. The second approach is to create a specialized programming language designed specifically for automatic differentiation, making it fully differentiable [56, 65, 66, 68]. Then, the compilation between these specialized languages and target languages needs to be implemented. More precisely, one can separate five different techniques to implement automatic differentiation on programming languages.

Language	Tool	Type	Mode	Reference
AMPL	AMPL	INT	Forward, Reverse	[42]
C, C++	ADIC	ST	Forward, Reverse	[43]
	ADOL-C	OO	Forward, Reverse	[44]
C++	Ceres Solver	LIB	Forward	
	CppAD	OO	Forward, Reverse	[45]
	FADBAD++	OO	Forward, Reverse	[46]
	Mxyzptlk	OO	Forward	[47]
C#	AutoDiff	LIB	Reverse	[48]
F#, C#	DiffSharp	OO	Forward, Reverse	[49]
Fortran	ADIFOR	ST	Forward, Reverse	[50]
	NAGWare	COM	Forward, Reverse	[51]
	TAMC	ST	Reverse	[52]
Fortran, C	COSY	INT	Forward	[53]
	Tapenade	ST	Forward, Reverse	[37]
Haskell	ad	OO	Forward, Reverse	
Java	ADiJaC	ST	Forward, Reverse	[54]
	Deriva	LIB	Reverse	
Julia	JuliaDiff	OO	Forward, Reverse	[55]
	ForwardDiff	INT	Forward	[55]
	Zygote	ST	Reverse	[38]
LLVM	Enzyme	LIB	Reverse	[56]
Lua	torch-autograd	OO	Reverse	
MATLAB	ADiMat	ST	Forward, Reverse	[57]
	INTLab	OO	Forward	[58]
	TOMLAB/MAD	OO	Forward	[59]
Python	ad	OO	Reverse	
	autograd	OO	Forward, Reverse	[60]
	Chainer	OO	Reverse	[61]
	Jax	OO	Forward, Reverse	[62]
	PyTorch	OO	Reverse	[40]
	Tangent	ST	Forward, Reverse	[63]
Scheme	R6RS-AD	OO	Forward, Reverse	
	Scmutils	OO	Forward	[64]
	Stalingrad	COM	Forward, Reverse	[65]
SQL	GradDesc	COM	Forward	[11]
Taichi	DiffTaichi	ST	Reverse	[66]

Table I.2.: Survey of automatic differentiation implementations. This is based on [1] and updated. One can notice that there is no reverse-mode available on relational programming languages.

Compiler-based

Compiler-based automatic differentiation (COMP) is integrated into the compiler itself. The compiler analyzes the code and generates the derivative computations as part of the compilation process. This allows for efficient differentiation without the need for runtime overhead.

Interpreter-based

In interpreter-based automatic differentiation (INT), the differentiation is performed during the interpretation phase of the code execution. The interpreter analyzes the code on the fly and computes the derivatives as needed. This approach provides flexibility and dynamic control flow, as the differentiation can adapt to the program's execution. However, it can introduce runtime overhead.

Library-based

Library-based automatic differentiation (LIB) involves the use of dedicated libraries or frameworks that provide functions and APIs for automatic differentiation. These libraries typically offer a range of differentiation operations and can be used alongside existing programming languages.

Operator Overloading-based

In operator overloading-based automatic differentiation (OO), the mathematical operators of a programming language are overloaded to automatically compute the derivatives of functions. This allows for the computation of derivatives without explicitly programming the differentiation rules.

Source Transformation-based

Source transformation-based automatic differentiation (ST) involves modifying the source code of a program to include differentiation operations explicitly. The program code is transformed to include the computation of derivatives, either manually or through automated tools. This approach allows for fine-grained control over the differentiation process but requires modifications to the original code.

Each approach has its own advantages and considerations, and the choice depends on factors such as the programming language, the desired level of control, performance requirements, and the specific use case of automatic differentiation. A comprehensive survey of automatic differentiation systems is presented in Table I.2, which is an updated version of [1]. This table aims to provide an overview of various implementations and is not intended to be exhaustive. It highlights the diverse range of automatic differentiation tools available, each specifically developed to suit particular frameworks or facilitate integration with specific operations relevant to the field or target hardware.

Following the presentation of differentiation and its generic automatic implementations, our focus now shifts towards relational queries, which constitute the target of our differentiation efforts.

I.2. Relational query

I.2.1. Industrial context

Lokad specializes in handling client supply chain data, which is typically structured as tables with relational links between them. To ensure data cleanliness and organization, the most commonly used tools are relational database systems. However, when it comes to optimization tasks, Lokad has encountered challenges due to the temptation to rely on external machine learning algorithms written in Python for example. This reliance on multiple frameworks has resulted in friction and inefficiencies for Lokad. The forecasting challenge presented in the introduction of this section illustrates the challenges faced by Lokad. Lokad could extract the necessary data from the relational database into external Python frameworks such as scikit-learn [69] to develop and optimize machine learning models. This process not only introduces additional complexity but also requires data movement, transformation, and synchronization between the database and the external frameworks.

The limitations of this approach become evident when working with large-scale datasets. The frequent necessity to transfer and transform data between the relational database and external frameworks introduces substantial overhead and becomes a performance bottleneck [11]. When delivering results to clients on a daily basis, it is imperative that the entire process can be completed within a day. Moreover, maintaining consistency and synchronization between the database and the external frameworks can be challenging, particularly when dealing with real-time or near real-time data updates. From an industrial standpoint, even a few hours of unavailability can have significant and detrimental consequences.

To address these challenges and improve the efficiency of optimization tasks, Lokad recognizes the value of developing machine learning models directly within the relational database system itself. By enabling differentiable programming in the database, Lokad aims to leverage the inherent relational structure of the data, perform computations at the source, and minimize data movement and synchronization overhead. Data transfer is often a significant time-consuming part, as measured in [11] for machine learning tasks. To support this statement, we argue that in many production models at Lokad that deal with categorical data, most of the resources are used to load and prepare the data, while the optimization part is minor in terms of computing. To emphasize this point, we have evaluated the CPU time of several production runs at Lokad using the developed optimization framework. Our findings indicate that, on average, 42% of the CPU time is devoted to gradient descent optimization, while the remaining time is allocated to data processing tasks, such as loading, cleaning, rendering, and exporting. The median value for the proportion of CPU time dedicated to gradient descent optimization drops to 34%. In essence, while Lokad minimizes data transfer by consolidating all computations, including data processing and optimization, within a single framework, data handling remains the primary source of resource consumption. One possible explanation for this phenomenon is that iterative optimization methods applied to the same dataset tend to be more effective than performing a pipeline of disparate operations on continuously changing data.

The ratio mentioned above is not applicable to tensor-based systems that run deep neural networks, as the high resource consumption is often beyond the capabilities of small companies. However, machine learning models, especially deep learning models, excel in modeling highly complex functions that primarily learn relevant features in large parameter spaces (millions or tens of millions of parameters). They learn a final decision or regression function that is generally of much lower complexity. It is expected that the optimization of such models requires a substantial amount of data and computational resources. On the other hand, white box models developed in supply chain must closely align with the semantics of the input data. Furthermore, the optimized functions do not exceed the complexity of the underlying database schema. There is no need to learn hidden representations (features) from the input data. Therefore, it is not surprising to observe the time ratio between computation and transfer when addressing optimization problems in supply chain.

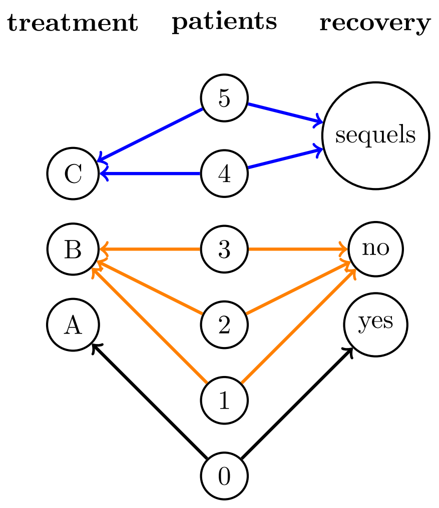
In following sections, we will delve into the properties of relational data compared to tensor data and introduce relational algebra.

I.2.2. Specificity of relational data

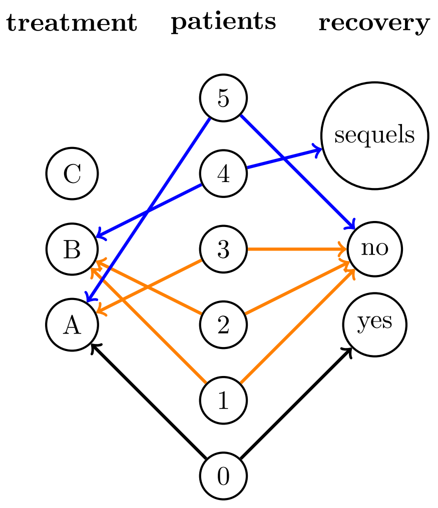
To highlight the difference between strongly structured relational data and tensor like image, we present two different data and their modification with 40% of noise.

On one hand, Figure I.7 represents two instances of the same tabular data storing medical treatment information on different patients: I.7a is the clean data while I.7b is the 40% noise version. In the clean data, we can hint that:

- treatment A cures patients,
- treatment B does not cure patients,
- treatment C cures patients with sequels.



(a) No noise.



(b) 40% noise.

Figure I.7.: Relational data and noise. Figure I.7a represents clean toy relational data and Figure I.7b depicts its polluted version. Such kinds of relational data do not support this level of noise.

On the other hand, Figure I.8 represents two instances of the image: I.8b is the clean data while I.8b is the 40% noise version. One can acknowledge that the image data is still readable with noise while the tabular data does not bring the same information at all with noise. On the noise version of the tabular data, the previous hints do not apply anymore. An error on a specific pixel has almost no importance while a treatment error can have a massive impact.

(a) No noise. (b) 40% noise.

Figure I.8.: Image data and noise. Figure I.8a represents a clean image of a logo and Figure I.8b depicts its polluted version. Such kinds of image data do not support this level of noise, as the main information is still present.

This aims to show that relational data deserve a specific treatment, especially to perform machine learning on it. One of the primary objectives of this initial section is to establish a sound theoretical framework for managing relational data. Specifically, we seek to differentiate a query that arises from combining data from multiple tables, which motivates the need to introduce the principles of relational algebra theory.

I.2.3. Relational algebra

Relational algebra [70] is a theoretical framework for handling and querying structured data in a relational database system. It provides a set of operators that allow for efficient data manipulation, combining tables, filtering data, performing calculations, and aggregating information. By understanding the principles of relational algebra, one can effectively work with relational data and optimize database operations. The majority of database system implementations follow this framework. In this work, we focus on the portion of relational algebra that is required for differentiation. To maintain consistency with the terminology used in other sections, we use the term *table* instead of *relation*.

Definition 1 (table)**.** *A table* $T = (\{a_1 \dots a_m\}, t_{i\leq n})$ *is a list of* m *attributes* $\{a_1 \dots a_m\}$ *and a finite set of m-tuples* $t_{i\leq n}$*. Each attribute takes its value within a specific discrete or continuous set.*

Definition 2 (primary key)**.** *The primary key of a table* T *is a subset* $\{a_i\}_{i\in I}$ *of the table's attribute such that the tuples values restricted to this subset are unique throughout the table:*

$$\forall t, t' \in T, \quad t_{\{a_i\}_{i\in I}} = t'_{\{a_i\}_{i\in I}} \Leftrightarrow t = t'.$$

Every table has a primary key.

Definition 3 (foreign key)**.** *A foreign key is a set of attributes* $\{b_i\}_{i\in I}$ *from table* T_1 *related to a primary key* $\{c_j\}_{j\in J}$ *from* T_2 *such that*

$$\forall t \in T_1, \exists t' \in T_2; \quad t_{\{b_i\}_{i\in I}} = t'_{\{c_j\}_{j\in J}}.$$

By extension a foreign key is a column or a set of columns in one table that refers to the primary key of another table. The purpose of a foreign key is to enforce referential integrity, which requires that the values in the foreign key column(s) match the values in the primary key of the related table. By establishing a relationship between two tables, a foreign key ensures that data is consistent and accurate.

Relational algebra provides a set of operators that can be applied to tables in a database. A query is a composition of these operators applied to a list of tables. The output of the query is also a table.

PROJECTION

A projection is an operation that consists of selecting a subset A of the attributes of the table T and thus reduces the size of the tuples. It is denoted as follows:

$$\Pi_{A \subset \{a_1 \dots a_m\}}(T) = (A, t_{i \leq n}).$$

The projections can be extended into generalized projections that apply a tuple-to-tuple function on every element of the table. Given a function $f : t \longrightarrow t'$, The following operator maps f to every tuple of T:

$$\Pi_f(T) = (A, f(t_i)_{i \leq n}).$$

The list of raw functions supported in the map operation is tiny but operator compositions allow us to build all the usual functions. The list of map operation can be split into two sub lists: the logical operators like $\wedge$, $\neg$, $\vee$ ... and the mathematical ones like x^n, $\cos x$, $\sin x$, e^x, $\ln x$ ... In the following we will split the queries into their relational and mathematical aspects. The mathematical aspect of the queries relies on the mathematical map operations.

SELECTION

A selection σ_ϕ is an operation that reduces the table T by reducing the number of tuples. The selection keeps the tuples satisfying a predicate ϕ that is a boolean function on the tuples space:

$$\sigma_\phi(T) = (\{a_1 \dots a_m\}, t_{i \leq n \quad and \quad \phi(t_i)}).$$

JOINS

There exists multiple ways to group tables into one, depending on what we want to obtain.

The **Cartesian Product** is a very simple way to join two different tables $R = (\{r_1, \dots, r_n\}, \rho_{i \leq n_R})$ and $S = (\{s_1, \dots, s_m\}, \zeta_{j \leq n_S})$. This is defined in Formula I.10:

$$R \times S = (\{r_1, \dots, r_n, s_1, \dots, s_m\}, \{\rho_i \cup \zeta_j \mid i \leq n_R; j \leq n_S\}). \tag{I.10}$$

The **Natural Join** $\bowtie$ between two tables R and S is the selection of the Cartesian Product to the tuples $r \in R$ and $s \in S$ that have a common value on their shared attributes.

The **Inner Join** $\bowtie_\theta$ between two tables R and S behaves like the Natural Join but the selection is applied on a given predicate θ rather than the simple matching of the values on the shared attributes:

$$R \bowtie_\theta S = (\{r_1, \dots, r_n, s_1, \dots, s_m\}, \{\rho_i \cup \zeta_j \mid i \leq n_R; j \leq n_S \text{ and } \theta(\rho_i \cup \zeta_j)\}). \tag{I.11}$$

There are many other join operators that are not described here as not used in the following.

AGGREGATION

The aggregation operation is the application of a given function agg on the tuples of a given attribute a of a table T noted as follows:

$$G_{agg(a)}(T) = \underset{i \leq n}{agg} \quad t_i[a]. \tag{I.12}$$

The function agg has to be defined on sets of elements of the type of the given attribute. The most common functions are Sum, $Count$, $Maximum$, Any ...

NULL

NULL values represent missing or unknown data. They are used to indicate the absence of a value or the inability to provide a valid value for a particular attribute or field in a table. There are situations where certain data cannot be obtained or are not defined for a particular record. For example, if a customer did not provide his phone number, the corresponding field can be marked as NULL to indicate the unavailability of that information. NULL values are crucial for maintaining data integrity in a database. They allow for the distinction between an empty value and a NULL value, ensuring accurate representation of missing or unknown data. They allow for conditional statements that involve checking for the presence or absence of data. For example, filtering records where a particular attribute is NULL can help identify missing data for further analysis or data cleaning. In certain cases, NULL values propagate through mathematical operations. When performing calculations or aggregations involving NULL values, the result is often NULL. This behavior helps maintain consistency in data computations. Overall, NULL values provide a standardized and explicit way to handle missing or impossible data in a database, allowing for more accurate data representation, querying, and analysis.

There exist multiple statistical techniques in order to fill NULL values in a database system. In the following, we do not consider NULL values. We suppose that all the data cleaning and preprocessing is done before the query differentiation. It is to be noticed that the TOTAL JOIN operator later introduced in Section I.3.3 cannot create NULL values by design. Without NULL values as input and with operators that do not create ones, this assumption is realistic.

Remark 3. *There exists a distinction between the theoretical relational algebra that has been presented and the various implementations of relational programming languages. The most renowned example is SQL (Structured Query Language), which will serve as a reference for implementation examples throughout the following sections.*

I.2.4. Query

A query $\mathcal{R}$ is the result of applying a combination of operators on input tables. As described in Section I.2.3, these operators take tables as input and output tables, which can be combined to create a single table known as the query result. Join operators are essential for working with relational data and are typically implemented in relational programming languages to optimize performance, as they can be resource-intensive. Formally a query is the composition of relational operators on existing tables.

In the context of relational linear regression using data from Table I.1, we consider the attributes of Table Obs as $\{id; c; x; y\}$ and the attributes of Table Cat as $\{c'; s\}$. The relational linear regression can be represented as follows:

$$\Pi_{f_b}(Obs \bowtie_{c=c'} Cat), \tag{I.13}$$

where $f_b(id, c, x, y, c', s) = s \times x + b$.

Having presented the theoretical framework of relational algebra, we advocate that performing machine learning within this framework would be a crucial achievement towards the introduction of optimization and machine learning techniques in many industrial application domains that bear on database systems. The following paragraph provides an overview of the recent attempts toward this goal published in the literature, and allows us to identify our contribution.

I.2.5. Existing automatic differentiation on relational queries

Data are heterogeneous, but most machine learning systems rely on tensor types such as Pytorch or Tensorflow. This is perfectly suited for many applications dealing with images for example, but this is not the best framework for relational data, while relational algebra (see Section I.2.3) is. For example, Pandas library [71] proposes an API to be requested with SQL queries as it is the appropriate way to handle relational data. Of course tensor data libraries propose to handle relational data but are not primarily designed for it. Programming languages dedicated to automatic differentiation [56, 65, 66, 68] are not designed for relational programming languages interoperability either. DiffTaichi [66] is designed for physical simulators, [68] and Stalingrad [65] serve theoretical analysis of automatic differentiation. Enzyme [56] performs automatic differentiation directly on the LLVM compiler which is mostly used by deep learning libraries.

When it comes to relational programming languages, there are only a few attempts since machine learning tools are largely absent from database systems. Some works, such as [11], have begun to address the subject by differentiating a portion of the SQL language. For example [11] proposes to support traditional linear regression, as presented in Listing I.5.

```
WITH means AS
    (SELECT avg(x) AS mean_x, avg(y) AS mean_y FROM datapoints),
    sums AS (SELECT
                sum((x - mean_x) * (y - mean_y)) AS numerator,
                sum(power(x - mean_x, 2)) as denominator
                FROM datapoints, means),
    a AS (SELECT a, numerator / denominator AS value FROM sums),
    b AS (SELECT
            b,
            mean_y - a.value * mean_x AS value FROM means,
            a)
SELECT * FROM b union SELECT * FROM a;
```

Listing I.5: Simple linear regression in SQL presented in [11]. Unlike relational linear regression, which involves multiple slopes and intercepts, the approach described in this listing focuses on a single slope and intercept.

This holds great promise and proves to be highly valuable, particularly when an expert needs to implement a straightforward model like linear regression within a larger set of relational operations.

While this is promising, the authors still describe it as an architectural blueprint. A follow up work [72] introduced a declarative machine learning language that can be translated both into SQL and Python as target platforms. [73, 74] have also worked in this direction. Their aim is to replicate the gradient-based methods used in tensor-based systems as [75] claims that their machine learning pipeline in SQL showed comparable performance to traditional machine learning frameworks. They even reproduced small neural networks in SQL as a proof of concept. Their primary objective is to reduce data transfer costs, which is crucial. From our perspective, another interesting challenge lies in enabling machine learning to fully leverage the relationship between input and model parameters. For instance, the implementation of relational linear regression or the retail forecasting model described by Equation I.2 cannot be seamlessly achieved using these frameworks. It could be done by facilitating the construction of relational models that take into account the specificity of this input data. Our perspective is summarized in Table I.3, which outlines our viewpoint on the appropriate tool to use in function of the model one wants to use on relational data.

Model	Suited place	Has been done by
Linear regression	Relational programming languages	[11, 72, 75]
Deep learning	Deep learning tensor libraries	[40, 41] . . .
Relational linear regression	Relational programming languages	not yet

Table I.3.: This table outlines the recommended tools according to us for performing machine learning on relational data.

Database systems serve as the appropriate framework for structured data. Though database systems are widely used to manage relational data, they lack integrated machine learning tools. This deficiency can be attributed to the lack of widespread automatic differentiation systems in database systems. Indeed, as presented in Section I.1.2, modern machine learning relies on gradient methods; without an automatic differentiation system, machine learning is thus not feasible. This lack of widespread tools for handling relational data may explain why tabular datasets are referred to as "the last unconquered castle" for deep learning by [6], whereas database systems are proven sufficient to express an end to end machine learning framework with data preprocessing, model training and its validation [75].

Furthermore, the database systems community and the machine learning community remain distinctly separate [76]. The absence of widespread automatic differentiation hinders the convergence of these two communities, which is detrimental, especially in domains such as supply chain and healthcare where field experts often work with database systems. Their expertise is invaluable for designing predictive models for related tasks. Allowing these experts to construct their models using differentiable programming tools would result in white box models that would be fully explainable. The following work aims to bridge the gap between these two communities by proposing the first framework for building *categorical models* (see Section III.1.2) directly within database systems. The programming language dedicated to automatic differentiation is open sourced[2] and Lokad complete pipeline is available[3].

We have highlighted the need of differentiating relational queries to perform important tasks on relational data. Current solutions show promise; however, they do not fully exploit the potential of data relationships. These solutions aim to replicate models, such as linear regression or classical deep networks [73,75], developed on tensor-based frameworks within database systems. Therefore, our aim is to build a framework where differentiable programming is a first-class citizen in relational programming languages. Doing so we will be able to replicate the standard models but also create new ones that leverage the relational aspect of the data, like relational linear regression presented in I.5.3. We will first present our approach to constructing this framework, which involves separating the query into its relational and mathematical components.

I.3. Differentiable programming on relational queries

The rest of this chapter is based on [77]. This work has been the subject of my talks at VLDB 2021 and the MODE Workshop on Differentiable Programming for Experiment Design 2022.

In the following section, we develop our brand new theoretical set up for query differentiation. This will allow us to construct complex decision or regression functions within a database system and optimize them using gradient descent. The main idea is to split the operations of a given query $\mathcal{R}$ into two parts. First there are the relational operations on the tables T_s and their relations that just propagate gradients. Second, there are the mathematical operations f that create complex gradients following automatic differentiation rules. This separation is presented in Figure I.9.

$$\begin{tikzcd}\mathcal{R} \arrow[r] & T_s, f\end{tikzcd}$$

Figure I.9.: Query splitting into its relational and its mathematical parts.

Differentiable programming is represented in Figure 1. Applying this representation to differentiation of relational queries lead to Figure I.10 where the query is split into its mathematical function f and the tables on which the query occurs.

Thanks to our approach, we obtain the gradient of a query as another query, which is a significant contribution of our work and offers multiple benefits.

Firstly, all the available optimizations for queries are applicable: the resulting gradient query benefits from the same support as the original one. This remark pertains to the optimizations available in the database system during both compilation and execution time of the relational query.

Secondly, differentiable programming is highly programmatic. When external tools call relational queries, returning the gradient as a query enables composability. In Section

[2]Adsl library can be found at https://github.com/Lokad/Adsl
[3]https://try.testing.lokad.com/

$$\begin{tikzcd}
\mathcal{R} \arrow[r] & T_s, f \arrow[d] \\
\mathcal{R}' & T_s, f' \arrow[l]
\end{tikzcd}$$

Figure I.10.: Path to differentiation. The direct differentiation of $\mathcal{R}$ to $\mathcal{R}'$ does not exist yet so we introduce a novel way to do it.

I.4.4, we present the reverse and forward modes of automatic differentiation, making this work compatible with any differentiable programming framework, regardless of the automatic differentiation mode implemented.

I.3.1. Notations

We consider the supervised learning set up with a given set of training labeled data $\mathcal{Z} = \{z_i = (X_i; y_i); i = 1 \dots n\}$, with the feature vectors $X_i \in \mathbb{R}^p$ and the scalar targets $y_i \in \mathbb{R}$.

We aim to find the best parameter $\theta^\star \in \mathbb{R}^p$ to minimize the loss $F_{\theta^\star}$ on the whole dataset:

$$\begin{aligned} f: \quad & \mathbb{R}^p \times \mathcal{Z} \longrightarrow \mathbb{R} \\ & \theta, (X, y) \longrightarrow f_\theta(X, y) \end{aligned} \tag{I.14}$$

$$\begin{aligned} \theta^\star &= \underset{\theta \in \mathbb{R}^p}{\arg\min} \quad F_\theta \\ &= \underset{\theta \in \mathbb{R}^p}{\arg\min} \sum_{X, y \in \mathcal{Z}} f_\theta(X, y) \\ &= \underset{\theta \in \mathbb{R}^p}{\arg\min} \sum_{i=1 \dots n} f_\theta(X_i, y_i). \end{aligned}$$

Our strategy to get closer and closer to $\theta^\star$ is to perform gradient descent as detailed in Chapter II. To do so we need an access to the gradient of f, we describe how to do it when the whole function is in fact a query. Note that in this case, the data X_i are entries of the tables T_s. One of the main challenges is to access these entries without loading the full tables for each computation of the gradient of f. All the following are oriented in this direction.

We apply these notations to the relational linear example. Let's assume that the Observation table (properly defined as the observation table in Section I.3.4) is of size n, the Category table of size c and that we use the norm l_2. It reduces to finding the best vector $A^\star \in \mathbb{R}^c$ in the Categories table and the best scalar $b^\star$ such as:

$$A^\star, b^\star = \underset{A \in \mathbb{R}^c, b \in \mathbb{R}}{\arg\min} \quad \sum_{i=1 \dots n} (y_i - A[cat_i]x_i + b)^2.$$

I.3.2. A loss query is relational and mathematical

Relational

We have presented relational operators in Section I.2.3. In order to fit the minimization presented above, the query, which is a composition of relational operators, has to end by a SUM-aggregation of a projection of the loss attribute. It takes the form of Formula I.15.

$$G_{SUM(loss)}(\Pi_{loss}(Observations)). \tag{I.15}$$

For a given query $\mathcal{R}$, the construction of the observation table and its loss attribute involves multiple tables T_s that form the relational aspect of the query.

Regarding the relational linear regression, the relational aspect of the query corresponds to the relationship between the Observation and the Categories table, properly defined in Section I.3.3.

Math

When retrieving information from a database through a query, the use of mathematical concepts is usually limited. However, when designing a predictive model as a query, mathematical operations become crucial. Although the set of available mathematical operators is limited, it enables the construction of highly useful ones. For instance, the raw operator *exp* can be used to build the *softmax* operator σ, as shown in Equation I.16.

$$\sigma(Z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}. \tag{I.16}$$

One can find the list of the operations we have implemented (and their derivative) in our query differentiation system in Table I.5. For a given query $\mathcal{R}$, the composition f of these mathematical operators forms the mathematical aspect of the query.

Regarding the relational linear regression, the mathematical aspect of the query corresponds to the standard linear regression from Equation I.1, ignoring the slopes being distinct by categories.

In order to effectively separate these two aspects of a query, we will introduce the PolyStars in the subsequent section. The construction of the PolyStars will be facilitated by a novel join operator known as the TOTAL JOIN.

I.3.3. TOTAL JOIN operator

In order to enable the construction of relational models such as relational linear regression, it is essential to have guarantees regarding the relationships between tables. In the context of relational linear regression, where we seek a slope parameter for each category, we aim to store the slope values within the Category table. To ensure this, it is necessary to establish that for each tuple in the observations table, there exists one and only one corresponding tuple in the Category table. If there were more than a corresponding tuple in the Category table there would be ambiguity in the parameter value. If there were no corresponding tuple in the Category table, it would lack the parameter value. The introduction of the concept of TOTAL JOIN in the following provides the necessary guarantee to fulfill this requirement.

Many database management tools are a sort of implementation of the model presented in I.2.3. In SQL, multiple joins types are possible: (INNER) JOIN, LEFT (OUTER) JOIN, RIGHT (OUTER) JOIN, FULL (OUTER) JOIN ... We introduce a novel join operator that helps us to simplify the table tree: TOTAL JOIN. T_1 TOTAL JOIN T_2 ON $\langle\theta\rangle$ is the same semantic as T_1 INNER JOIN T_2 ON $\langle\theta\rangle$ (presented in Section I.2.3) with the additional constraint that for each line of T_1, there is *exactly* one line of T_2 that corresponds. To make a successful T_1 TOTAL JOIN T_2 ON $\langle\theta\rangle$ it is sufficient that θ columns are a primary key (from Definition 2) in T_2 and a foreign key in T_1, but it is not necessary. This is true because a primary key is a non ambiguous way to select a unique tuple in the related table.

```
SELECT *
    FROM A
    INNER JOIN B
        ON A.K = B.K
```

Listing I.6: INNER JOIN.

```
SELECT *
    FROM A
    TOTAL JOIN B
        ON A.K = B.K
```

Listing I.7: TOTAL JOIN.

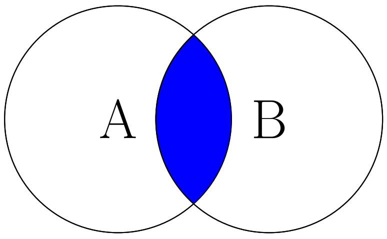


Figure I.11.: INNER JOIN representation. Without any restriction on the tables, there might be values for the K attribute in the B table that are not present in the A table.

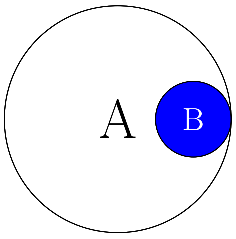


Figure I.12.: TOTAL JOIN representation. With restrictions on the tables, all the lines of B are concerned by this join operation.

This novel join operator does not create different tables than the INNER JOIN operator but it gives guarantees on the relationships between the tables and thus allows us to easily construct *PolyStars*, defined Section I.3.4. The difference is highlighted in Figures I.11 and I.12. This operator also allows us to avoid dealing with NULL values as presented in Section I.2.3 with strict conditions on the table relationships.

I.3.4. PolyStar

In this section, we introduce the definition of the *PolyStar*, which is a way to see the graph of the different tables used in the query we aim to differentiate. This is a key concept in order to isolate the relational part of a query from its mathematical one, which is the core of our differentiation approach.

Let's start by the introduction of the *Polytree* [78], on which our *PolyStar* relies on.

Definition 4 (Polytree)**.** *A Polytree is a directed acyclic graph whose underlying undirected graph is a tree.*

An example is given Figure I.13. For instance, genealogical trees can be considered as Polytrees. The direction of the edges is determined by the parental relationships, and the structure is evidently acyclic since no individual can be both an ancestor and a descendant of the same person.

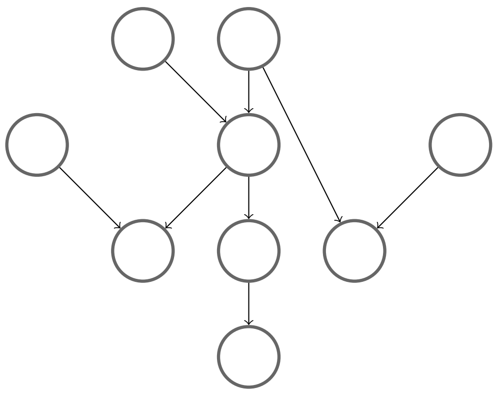


Figure I.13.: A generic Polytree.

Let T_s be the set of tables used in a query. Let's introduce the relationship "$T_A \longrightarrow T_B$" when the primary key of T_A is a foreign key in T_B. It is said that T_A broadcasts into T_B.
A simple way to create such T_A and T_B in SQL is presented in Listing I.8.

```
CREATE table TA AS
    SELECT foreignKey AS primaryKey
    FROM TB
    GROUP BY foreignKey
```

Listing I.8: Creating implicit broadcast.

In the following, any "$\longrightarrow$" between tables means *"broadcasts into"*.

Definition 5 (Cross Edge)**.** *A cross-edge is a pair of edges in a graph* $(A \longrightarrow B, C \longrightarrow B)$ *which indicates that B comes from a **cross** operation between A and C.*

Here is a simple way to create such a cross edge in SQL, which is the implementation of the Cartesian Join presented in Formula I.10:

```
CREATE table B AS
    SELECT * FROM A
    CROSS JOIN C
```

Listing I.9: SQL cross edge creation.

Definition 6 (PolyStar)**.** *Let's define a PolyStar* $P\star = (P, ot)$ *as a Polytree P with cross-edges and ot a node of P.*

A PolyStar is a Polytree with a special focus on a specific node of the graph called the observation node. This special focus gives a natural coloration of the graph.

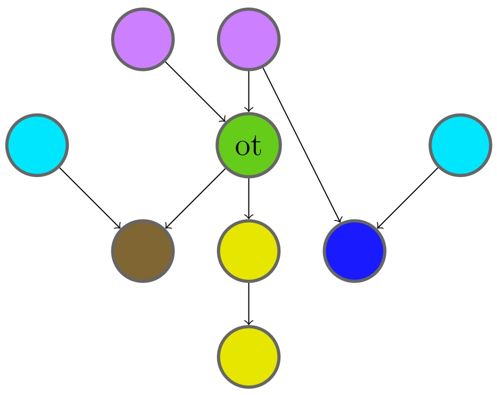


Figure I.14.: A PolyStar with its corresponding coloration thanks to the special focus on the *ot* node, which extends the PolyTree from Figure I.13.

From the special focus on a specific node ot, that we now call the observation node, we can classify the other nodes. Let n be a node of $(P, ot) = P\star$, we call:

- an *upstream* node, a node n of P such that $n \longrightarrow ot$.
- an *upstream-cross* node, a cross node n of P such that one of its parents is an upstream node.
- an *observation-cross* node, a cross node of P such that one of its parents is ot.
- a *downstream* node, a node d of P such that it is not an observation-cross node and that $ot \longrightarrow d$.
- a *full* node, all the remaining nodes of P.

These definitions uniquely define every node in the PolyStar, a generic example is given in Figure I.14. Two concrete examples on properly defined models are presented in Sections I.5.3 and I.5.4. As explained above, the PolyStar is introduced to fit on the tables used in the query. The names (upstream, downstream . . .) are introduced with the PolyStar, and are inspired by the flux' paths in the table tree induced by the query. Let's remember that the output of the query we aim to differentiate is a numerical vector (related to an attribute to fit with notations from Section I.2.3) in the observation table and we want to minimize its sum over the observation table, as presented in Formula I.15. The observation table corresponds to the training labeled data $\mathcal{Z}$ from Equation I.14. The loss function is, before aggregation, a vector in the observation table. Thus dimension of the inputs from tables can be defined from their relationships with the observation table. From the point of view of a line in the observation table, other tables do not need to be fully loaded and reduce to the following:

- An input from the observation table reduces to a scalar.
- An input from an upstream table reduces to a scalar.
- An input from an upstream-cross table reduces to a vector of the size of the right table used in the cross operation.
- An input from an observation-cross table reduces to a vector of the size of the right table used in the cross operation.

- An input from a downstream table reduces a vector of certain size.
- An input from a full table reduces a vector of the size of the full table itself.

In the relational linear regression example, there are two tables: Observations and Category. The Category table serves as an upstream table for the Observations table. In this context, for each line in the Observations table, the slope vector (a_A, a_B, a_C) from the Category table is reduced to a scalar parameter. Specifically, if the category of the row is A, the scalar parameter is a_A. Similarly, for categories B and C, the scalar parameters are a_B and a_C respectively. Furthermore, with this new perspective, Figure I can be observed, depicting how values from upstream parameters are loaded as scalars for each line of the observation table.

The PolyStar places a special emphasis on the observation node, which facilitates clear broadcasts between tables and allows for a lightweight semantic understanding of the relational aspect of the query. By freezing the relationships between tables, these clear broadcasts help prevent errors when joining tables. In tensor-based systems, the granularity of the loss at the observation table is implicit, as there is only one table. However, in our approach, we want to take full advantage of the relational aspect of the data and not flatten it into a single table. Therefore, the PolyStar is crucial in enabling us to properly define the quantity we aim to minimize.

With the relational aspect of the query now managed by PolyStar, and the query prepared for differentiation, we introduce the specialized programming language we have developed specifically for conducting automatic differentiation operations. Subsequently, it will become feasible to perform differentiation of relational models, such as relational linear regression expressed as queries, directly within the database system itself.

I.4. A dedicated programming language: Adsl

I.4.1. Presentation of the language

We introduce Adsl[4], which is **A** **D**ifferentiable **S**ub **L**anguage that is intended to lower relational programming languages, i.e. to translate them into another one that is closer to the machine.

Although the implementation of the compilation between Adsl and each programming language is necessary for this method, the automatic differentiation system only needs to be implemented once. It means that we have opted for the library approach from Section I.1.3, as illustrated in Figure I.15.

Adsl has been created to enable the expression of a query's gradient as another query. Adsl is a language where automatic differentiation is a first-class citizen. It is closed by differentiation: the adjoint, i.e. the derived program, of an Adsl program is also a differentiable Adsl program. Adsl is a simple language that supports loops and conditional but one of its key characteristics is its ability to support aggregators and broadcasts between tables. This feature enables the expression of join operations and the creation of new tables within the language. The inclusion of broadcast and aggregator capabilities directly in the language allows us to generate the required operators in Adsl for effectively expressing the gradient of a query.

[4]Adsl library can be found at https://github.com/Lokad/Adsl

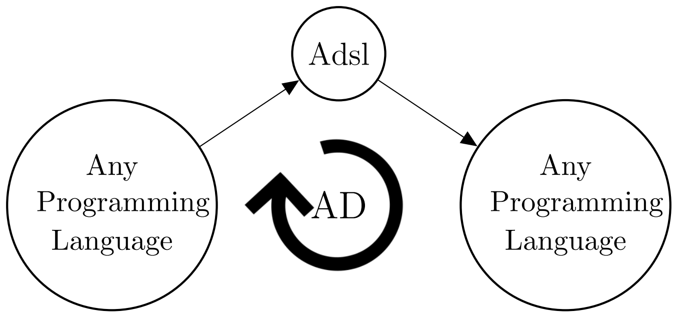


Figure I.15.: Adsl automatic differentiation schema.

As we crafted it from scratch, we designed it with specific properties, presented in Sections I.4.2 and I.4.3 that make Adsl differentiation as easy as possible. It also created a novel gradient estimator described in Chapter IV.

All of this relies on a specific treatment of Adsl's variables scope. The scope of a variable is the region within a program's source code where a variable is accessible and visible. The scope of a variable is determined by the rules of the programming language and affects how variables can be used and manipulated within the program. Formally, the scope of a variable can be defined as follows:

- Global scope: a variable declared outside any function or class has a global scope, which means it is accessible from any part of the code.
- Local scope: a variable declared within a function has a local scope. It is only accessible within the body of the function or method where it is declared. Once the function or method completes execution, the variable goes out of scope, and its value is lost.
- Block scope: a block-scoped variable is visible and accessible only within the block, like loops or conditionals, in which it is declared.

According to the definition below, an Adsl program is a list of Statements $\langle S \rangle$, whose grammar is defined in Grammar I.1.

$\langle\ S\ \rangle\quad ::=$	
$\mid\ \langle\ v \leftarrow e \rangle$	Variable assignment
$\mid\ \langle\ tup \leftarrow v \rangle$	Variable tupling
$\mid\ \langle\ \mathit{Cond}\ (\ v\quad \Psi\quad P_T\quad P_E\quad \Phi)\rangle$	Conditional
$\mid\ \langle\ \mathit{For}\ (\ \tau\quad \chi\quad rev\quad S\quad P\quad \Xi)\rangle$	Loop
$\mid\ \langle\ \mathit{Return}\ v\ \rangle$	Output of a program
$\langle\ e\ \rangle\quad ::=$	
$\mid\ \langle\ v\ \rangle$	Variable
$\mid\ \langle\ w\ \rangle$	Intermediate constant
$\mid\ \langle\ \oplus\quad tup \rangle$	Variable Addition
$\mid\ \langle\ \mathit{Call}\ op\ tup\ \rangle$	Function Call
$\mid\ \langle\ \mathit{Param}\ i\ \rangle$	Parameter access
$\mid\ \langle\ \mathit{Const}\ i\ \rangle$	Constant access
$\mid\ \langle\ v\quad \triangleleft\quad \beta\quad \rangle$	Broadcast Projector
$\mid\ \langle\ v\quad \triangleright\quad \alpha\quad \rangle$	Aggregation Projector
$\mid\ \langle\ \mathit{Pred}\ \rangle$	Predicate
$\langle\ \mathit{Pred}\ \rangle ::=$	
$\mid\ \langle\ \wedge\quad v\quad w \rangle$	And
$\mid\ \langle\ \vee\quad v\quad w \rangle$	Or
$\mid\ \langle\ \neq\quad v\quad w \rangle$	Inequality
$\mid\ \langle\ v \leq w \rangle$	

Grammar I.1.: Adsl expressions.

Most of the presented expressions are really simple and do not require further explanations. We do a special focus on the most interesting ones and illustrate them with pseudo code of simple examples below.

Parameter access versus Constant access

In Adsl, both parameters and constants can be assigned to a variable. This distinction is made to allow for different treatments of inputs with respect to the gradient computation, i.e., the parameters and the other inputs.

Example 1 (Traditional linear regression)**.** *Consider a simple linear regression on a set of n points $(x_i, y_i)_{i \leq n}$, with the goal of finding the slope a and intercept b that minimize the error:*

$$\sum_{i=1}^{n} (ax_i + b - y_i)^2 = \sum_{i=1}^{n} f_{a,b}(x_i, y_i).$$

In this example, a and b are the parameters and the (x_i, y_i) are the constant values. The Adsl program implementing the function $f_{a,b}$ is Adsl I.1.

We factorize the basic operations into a single Call. In this linear regression, the only relevant gradient are $\nabla_a f$ and $\nabla_b f$ as the values of the (x_i, y_i) data cannot be updated.

$$\begin{array}{l}
\langle a \quad \leftarrow \quad Param_0 \quad \rangle \\
\langle b \quad \leftarrow \quad Param_1 \quad \rangle \\
\langle x \quad \leftarrow \quad Constant_0 \quad \rangle \\
\langle y \quad \leftarrow \quad Constant_1 \quad \rangle \\
\langle z \quad \leftarrow \quad Call \quad f_{a,b}(x,y) \quad \rangle \\
\langle Return \quad z \quad \rangle
\end{array}$$

Adsl I.1.: Adsl program of linear regression.

Conditional

We present here how conditional statements are supported in Adsl.

$$\langle Cond(v \quad \Psi \quad P_T \quad P_E \quad \Phi)\rangle$$

v is the boolean branch variable while P_T and P_E are the *then* and *else* list of statements. Variables used in P_T or P_E enter a branch by a $\psi \in \Psi$ and exit by a $\phi \in \Phi$. The Ψ and Φ make the variable scope local in the appropriate branch. The duality between these two operators will be key in order to differentiate these statements. Formally it gives:

$$\begin{array}{ll}
\forall \psi(x, x_T, x_E) \in \Psi, & \textbf{\textit{if}} \quad v \quad \textbf{\textit{then}} \quad x_T \leftarrow x \quad \textbf{\textit{else}} \quad x_E \leftarrow x \\
\forall \phi(y_T, y_E, y) \in \Phi, & \textbf{\textit{if}} \quad v \quad \textbf{\textit{then}} \quad y \leftarrow y_T \quad \textbf{\textit{else}} \quad y \leftarrow y_E
\end{array}$$

Example 2 (Adsl conditional). *Let's consider the following pseudo code of conditional in Listing I.1.*
Its Adsl form is

$$\begin{array}{l}
\langle x_0 \leftarrow \dots \rangle \\
\langle Cond(v \quad \Psi \quad P_T \quad P_E \quad \Phi)\rangle \\
\langle z \leftarrow \oplus \quad y \quad 1\rangle
\end{array}$$

with

$$\begin{aligned}
v &= x_0 > 0 \\
\Psi &= [\quad \psi(x_0, x_T, x_E) \quad] \\
P_T &= [\quad y_T \leftarrow x_T \quad] \\
P_E &= [\quad y_E \leftarrow -x_E \quad] \\
\Phi &= [\quad \phi(y_T, y_E, y) \quad]
\end{aligned}$$

With such construction, the scope of every variable used in conditional statements is strictly local, which will be highly valuable for the differentiation process. There is no equivalence between these statements and the **SELECTION** operation of the relational algebra but both implement the *if* behavior.

Broadcasts and aggregators

Broadcasts and aggregators [79, 80] are a key feature of Adsl as it is intended to perform automatic differentiation of relational programming languages. A broadcast from a table to another duplicates the desired value in the corresponding line. If no broadcast is specified, the natural one applies: the broadcast implied by the foreign key of the target table matching the primary key of the input table. A simple example is given below in Table I.4 where *Category* is the primary key of the Cat table while it is a foreign key in the Obs table:

Category	θ
A	1.2
B	1.8
C	-2.2

Cat table

Id	Category
01	B
02	A
03	A
04	B
05	C

Obs table

Id	Category	θ
01	B	1.8
02	A	1.2
03	A	1.2
04	B	1.8
05	C	-2.2

$Obs.\theta \leftarrow Cat.\theta \quad \lhd \quad \beta$

Table I.4.: Broadcast from the Cat table to the Obs one.

Aggregators are the opposite of broadcasts where the values are aggregated (the default aggregator is the *sum*) into a smaller table. If no relationship between the tables is specified, the natural one applies: the aggregation implied by the foreign key of the input table matching the primary key of the target table. The ability to express new relational operations within Adsl is crucial for our strategy, as it enables us to create and incorporate the necessary operations required by the gradient query that may not be present in the original query.

Formally aggregators and broadcasts can be seen as a multiplication by a 0-1matrix, also called association table. $\langle w \leftarrow v \lhd \beta \rangle$ corresponds to the notation $W = M_\beta V$ where $M_\beta \in \{0;1\}^{n\times m}$ and W, V have the matching sizes. In the previous example the broadcast between the Cat table to the Obs gives:

$$\begin{pmatrix} 1.8 \\ 1.2 \\ 1.3 \\ 1.8 \\ -2.2 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1.2 \\ 1.8 \\ -2.2 \end{pmatrix}. \tag{I.17}$$

This matrix point of view is helpful to understand their behavior, but in practice Adsl broadcasts and aggregators are not implemented as matrices, which is one of the main difference with tensor approaches.

Loops

Loops are a key part of Adsl as they unlock complex model creation and are a major enhancement compared to [11]. Moreover loops are the most resource consuming operations, thus a special care has been devoted in its design. We present here how loop statements are supported in Adsl. Loop expressions are constructed as:

$\langle For(\tau \quad \chi \quad rev \quad S \quad P \quad \Xi)\rangle$

τ represents the variable indicating the size and the order in which the loop must be traversed. In other words, this is the table being iterated over. χ serves as the entry point for every external variable to be used inside the loop. This is achieved through broadcasts or simple variable assignments, ensuring the block scope of these variables. rev is a boolean variable indicating whether the loop is traversed in reverse order or not. S is a set of states allowing the persistence of certain variables across loop iterations. A state inherits a value defined in χ, and at the end of an iteration, the loop body updates its state ($\Lsh$ is used as notation). P consists of a list of statements that form the loop body. Ξ represents the exit point for variables constructed during loop iterations.

Example 3 (Adsl loop)**.** *Let's consider the following pseudo code*

```
X <- ...
acc = 0
Y = 0
for i from 0 to size(X) do
    acc = acc + x
    Y[i] = acc
return Y, acc
```

Listing I.10: Pseudo-code of a program containing a loop.

Its Adsl form is

$$\begin{array}{lll} \langle X_0 & \leftarrow & \ldots\rangle \\ \langle a & \leftarrow & 0\rangle \\ \langle For & (\chi \quad false \quad S \quad P \quad \Xi)\rangle & \\ \langle Return & Y\rangle & \\ \langle Return & acc\rangle & \end{array}$$

with

$$\begin{aligned} \chi &= [\quad x \leftarrow X \lhd \beta \quad ; \quad a_1 \leftarrow a \quad] \\ S &= [\quad a_1 \Lsh a_3 \quad] \\ P &= [\quad a_2 \leftarrow a_1 + x \quad ; \quad a_4, a_3 \leftarrow a_2 \quad] \\ \Xi &= [\quad Y \leftarrow a_4 \quad ; \quad acc \leftarrow a_3] \end{aligned}$$

The scan operator, also known as prefix-reduce (scan is the name popularized by the MPI library [81]), is a specific instance of a loop. Formally, a scan is the iteration of a function f over a vector A, with the persistence of a state starting from x_0. It is represented in Figure I.16.

By example, the scan operator can implement exponential smoothing [82] in its simplest

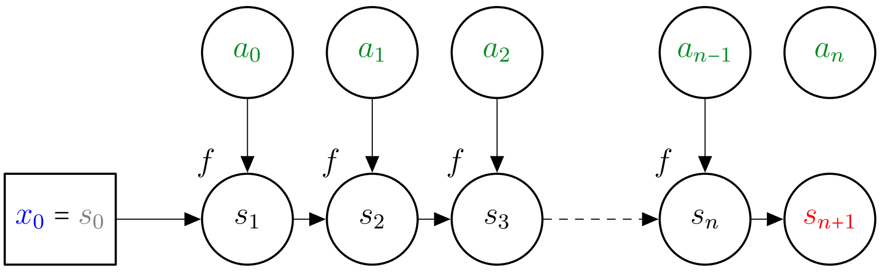


Figure I.16.: Scan Representation. The state s is initialized with x_0 and is updated while iterating the function f over the vector A.

form with $f_\alpha(s,x) = (1-\alpha)s + \alpha x$, where α is the smoothing factor:

$$\begin{aligned} s_0 &= x_0 \\ s_t &= f_\alpha(s_{t-1}, x_t). \end{aligned}$$

In this example, the starting point of the state is the first element of the vector iterated over, but it is not necessarily the case as presented in Section I.5.5 on a in-production example.

Having defined the expressions that constitute an Adsl program, we now turn our attention to the specific variable scope requirements that we aim for our program to fulfill.

I.4.2. SSA: Static single assignment form

Our goal is to enforce the strictest possible variable scope, which will ease the differentiation process for Adsl programs. Definition 7 aims to achieve this objective.

Definition 7 (SSA)**.** *Static Single Assignment form (SSA) is a property of an intermediate representation, which requires that each variable be assigned exactly once in the global scope, and every variable be defined before it is used.*

Adsl is SSA, which ensures us that the scope of a variable cannot be extended before one of its assignments as there is only one of them. SSA is a common property for programming languages and Zygote [38] also relies on it to implement automatic differentiation on Julia. We go beyond this property in order to completely close the scope of a variable by controlling its use, which also controls the scope of the adjoints.

I.4.3. SA: Single access

In addition to the SSA property, we add the Single Access property in Definition 8.

Definition 8 (SA)**.** *Single Access Form (SA) is a property of an intermediate representation, which requires that each variable be read no more than once even in the global scope.*

Adsl is also SA. By enforcing this property, we ensure that the scope of a variable cannot be extended after it has been read, as there is only one read operation allowed. Let's consider the function $f_1(x,y) = e^x \times (x+y)$. In Adsl it gives:

```
x ← Param_0                          x ← Param_0
y ← Param_1                          y ← Param_1
a ← e^x                              x_1, x_2 ← x
b ← ⊕   x   y        becomes         a ← e^{x_1}
z ← a × b                            b ← ⊕   x_2   y
Return   z                           z ← a × b
                                     Return   z
```

The meticulous handling of variable scopes in Adsl ensures that transforming an SSA program into an SSA-SA program is straightforward, provided that the tupling operation is available. This process involves listing the read operations for each variable within its scope and replacing all occurrences with an element of the tupling, dimensioned according to the number of read operations.

The usefulness of the SA property might not be immediately apparent. First let recall that Adsl programs are never directly written by programmers, they are compiled from another programming language. Writing an SA-form would be very painful. Second, it makes the use of every variable completely local, which simplifies its differentiation. Let us remember that Adsl is designed for differentiation thus all design choices we have made are in the direction of differentiation simplification. Finally the SA property leads to a novel gradient estimator detailed in Chapter IV.

Remark 4. *We have developed this approach in order to apply it for white box models on supply chains. A way to do interpretable machine learning is to restrict the number of parameters, i.e. way less than in deep learning, but with strong meaning and with multiple uses in the objective function. This approach might not seem relevant for deep neural networks with billions of parameters.*

In Section I.1.2 we have presented how automatic differentiation is applied on Wengert lists. We now explain how an SSA-SA Adsl program is an adequate representation of such a list.

I.4.4. Adsl and Wengert lists

$$f : \mathbb{R}^p \longrightarrow \mathbb{R}^q$$

In its SSA-SA version, an Adsl program can be directly converted into a Wengert list without any substantial modification of the representation as stated below. The following Theorem proves that Adsl has been appropriately designed to implement automatic differentiation.

Theorem 1 (Adsl can express Wengert lists)**.** *The trace of an SSA-SA Adsl program can be represented as another SSA-SA Adsl program.*

Proof. Let us consider an SSA-SA Adsl program P, which consists of a list of statements. Each statement $\langle S \rangle$ can be a tupling, a conditional, a variable assignment, a loop, or the output of the program. Statement by statement we construct the (very similar) Adsl program representing the trace.

• If $\langle S \rangle$ is a tupling or the output of the program, the statement itself serves as a valid representation of the trace.

• If $\langle S \rangle$ is a conditional $\langle Cond(v \quad \Psi \quad P_T \quad P_E \quad \Phi)\rangle$, the trace evaluates only the chosen branch. Without loss of generality, we assume that it is the first one. This conditional evaluation can be represented in Adsl using the following statements:

$$\begin{aligned}&\langle v_t \longleftarrow True\rangle\\&\langle Cond(v_T \quad \Psi \quad P_T \quad P_E \quad \Phi)\rangle\end{aligned}$$

The traces of the $\psi \in \Psi$ and $\phi \in \Phi$ are then reduced to a variable duplication to ensure their local scope. The Ψ and Φ formulations remain valid for representing the trace, even though only their *then* part will be utilized.

For the lists of statements, one can reason inductively for P_T, while P_E will not be used at all.

• If $\langle S \rangle$ is a variable assignment $\langle v \longleftarrow e\rangle$, the statement itself provides a valid representation of the trace. This holds true for all assignments, with the exception of broadcasts and aggregators. To understand why this also holds true, consider that broadcasts and aggregators are related to table relationships known at execution time. As a result, they can be regarded as a type of function call.

• If $\langle S \rangle$ is a loop $\langle For(\chi \quad rev \quad S \quad P \quad \Xi)\rangle$, the same loop can be employed to represent the trace. Variables entering the loop via χ, either through a broadcast or a duplication, are valid representations as previously discussed. The same logic is applicable for variables exiting the block scope via Ξ.

Similar to the conditional case, we can use induction to demonstrate that the list of statements P constitutes a valid representation of the trace.

For the states of S, they can be regarded as straightforward variable assignments at the beginning and the end of the list of statements P.

The Adsl program representing the trace of P is by construction SSA-SA regarding the minor modifications made to the already SSA-SA program, which concludes the proof. □

Adsl being able to express Wengert lists, as demonstrated by Theorem 1, is not a mere consequence of our Adsl design, but rather a driving factor. Given that Wengert lists are the primary objects used in automatic differentiation, we have developed Adsl as a language that closely aligns with Wengert lists, enabling efficient differentiation directly on it. In that sense, Theorem 1 plays a crucial role in the correctness of our approach based on the design of a dedicated programming language for automatic differentiation.

Automatic differentiation on Wengert lists computes the gradient of the output with respect to the input variables. In an Adsl, it translates into computing the gradient of the parameters introduced by the parameter assignment statement. Once the expression of Wengert lists into SSA-SA Adsl is established, we briefly discuss the two main approaches to perform automatic differentiation on a program: reverse of forward modes [35].

Reverse mode

Let apply the reverse mode of automatic differentiation on the SSA-SA form of the f_1 function:

$$\begin{array}{lcl}
\begin{array}{l}
x \leftarrow \mathit{Param}_0 \\
y \leftarrow \mathit{Param}_1 \\
x_1, x_2 \leftarrow x \\
a \leftarrow e^{x_1} \\
b \leftarrow \oplus \quad x_2 \quad y \\
z \leftarrow a \times b \\
\mathit{Return} \quad z
\end{array}
& \text{reverse mode gives} &
\begin{array}{l}
x \leftarrow \mathit{Param}_0 \\
y \leftarrow \mathit{Param}_1 \\
x_1, x_2 \leftarrow x \\
a \leftarrow e^{x_1} \\
b \leftarrow \oplus \quad x_2 \quad y \\
z \leftarrow a \times b \\
\overline{z} \leftarrow 1 \\
\overline{a}, \overline{b} \leftarrow \overline{z}b, \overline{z}a \\
\overline{x_2}, \overline{y} \leftarrow \overline{b} \\
\overline{x_1} \leftarrow \overline{a}e^{x_1} \\
\overline{x} \leftarrow \oplus \quad \overline{x_1} \quad \overline{x_2} \\
\mathit{Return} \quad \overline{x}, \overline{y}
\end{array}
\end{array}$$

Forward mode

When $p \gg q$ forward mode is prohibitive. Even though recent work seems promising [83] by limiting the number of Jacobian vector product computations, implementations by [84] on Adsl did not perform well in our experiment context (see Section III.3). For this main reason, we have selected reverse mode for implementing automatic differentiation in Adsl in production at Lokad.

I.4.5. Automatic differentiation of Adsl

The following is the reason for all the choices we have made while designing Adsl. Adsl is a programming language crafted for automatic differentiation. We have presented reverse and forward mode and for all the reasons depicted above, we have decided to implement reverse mode on Adsl. Our objective is to perform regression and classification with it, that means we are in the case of $q = 1$. Thus an Adsl program $P = [s_i \quad \text{for} \quad i = 0 \dots n]$ being a list of statements, the reverse mode implementation of automatic differentiation gives the adjoint program $\overline{P}$:

$$\overline{P} = \vec{P} \quad ; \quad \overleftarrow{P}.$$

The adjoint program $\overline{P}$ joins the forward pass $\vec{P} = [s_i \quad \text{for} \quad i = 0 \dots n \quad \text{if} \quad s_i \neq \mathit{Return}]$ and the backward pass $\overleftarrow{P} = [\overline{s_i} \quad \text{for} \quad i = n \dots 0]$. It is important to note that the forward pass $\vec{P}$ has no direct relationship with the forward mode of automatic differentiation.

In the following we present the adjoint $\overline{\langle s \rangle}$ of the existing Adsl statements $\langle s \rangle$. The adjoint of a statement $\langle s \rangle$ is the list of the statements that define the adjoint of the variables read in $\langle s \rangle$. This list often reduces to a simple statement, as summarized in Grammar I.2.

As presented in the reverse mode introduction, the adjoint program needs the variables computed during $\vec{P}$ to compute $\overleftarrow{P}$. To construct the adjoint of a program, we only use other Adsl statements, as it is close by differentiation.

Adjoint of a return

Every Adsl program ends with a return statement. By definition of the adjoint $\overline{\omega} = \frac{\partial f}{\partial \omega}$, we get $\overline{f} = \frac{\partial f}{\partial f} = 1$.

$$\overline{\langle Return \quad v\rangle} = \langle \overline{v} \leftarrow 1.0f\rangle$$

If the output is a non scalar vector we broadcast the final adjoint $\frac{\partial f}{\partial f} = 0$ into the appropriate vector. Nevertheless it does not happen in practice as we aim to perform gradient descent with the computed adjoint in order to minimize a function: one cannot minimize a vector.

Adjoint of a param

When we finally reach the parameter assignment in the differentiation process, we can output the computed adjoint as $\overline{\theta} = \frac{\partial f}{\partial \theta}$ is the targeted value.

$$\overline{\langle v \leftarrow Param \quad i\rangle} = \langle Return \quad \overline{v}\rangle$$

Adjoint of a tupling

The SA property of Adsl relies on the tupling operation.

$$\langle \overline{tup \leftarrow v}\rangle = \langle \overline{v_1 \dots v_t \leftarrow v}\rangle = \langle \overline{v} \leftarrow \oplus \quad \overline{v_1} \dots \overline{v_t}\rangle$$

Other automatic differentiation systems like Zygote for Julia [38] are not SA and rely on adjoint accumulation. Even though our approach increases the number of variables, it simplifies the compilation process. Moreover, the differentiation of a tupling statement is the foundation of the gradient estimator presented in Section IV.4.2.

Adjoint of a sum

The sum operation is not treated as a standard *call* operator in Adsl. This is due to its adjoint being a tupling statement. Given that tupling is unique in Adsl to support the SA property, special consideration is afforded to the sum operation:

$$\langle \overline{v \leftarrow \oplus \quad tup}\rangle = \langle \overline{v \leftarrow \oplus \quad v_1, .., v_t}\rangle = \langle \overline{tup} \leftarrow \overline{v}\rangle = \langle \overline{v_1}, .., \overline{v_t} \leftarrow \overline{v}\rangle$$

Adjoint of a Call

The adjoint of a call function can be found via the chain rule and the basic derivative formula. We give a list in Table I.5 but it does not need further explanations.

Function	Adjoint
$y = e^x$	$\overline{x} = \overline{y} \times e^x$
$y = \cos x$	$\overline{x} = \overline{y} \times \sin x$
$y = \sin x$	$\overline{x} = -\overline{y} \times \cos x$
$y = \ln x$	$\overline{x} = \frac{\overline{y}}{x}$
$y = x^p$	$\overline{x} = \overline{y} \times p \times x^{p-1}$
$y = x_a \times x_b$	$\overline{x_a} = \overline{y} \times x_b, \overline{x_b} = \overline{y} \times x_a$

Table I.5.: Adjoint of calls.

Adjoint of a conditional

The introduction of the Ψ and the Φ reveal all its usefulness while differentiating the conditional statement. We observe an elegant duality between these two operators as the adjoint of a ψ is a ϕ and vice versa. This can be understood as reverse mode back propagates adjoint through the code: when the variable *enters* the branch its adjoint exits its, when the variable *exits* the branch its adjoint enters its.

$$\langle\overline{Cond(\pi \quad \Psi \quad P_T \quad P_E \quad \Phi)}\rangle = \langle Cond(\pi \quad \overline{\Phi} \quad \overline{P_T} \quad \overline{P_E} \quad \overline{\Psi})\rangle$$

With $\overline{\Psi} = \overline{[\psi_i]} = [\overline{\psi_i}]$ and $\overline{\Phi} = \overline{[\phi_i]} = [\overline{\phi_i}]$ while $\overline{\phi(y_T, y_E, y)} = \psi(\overline{y}, \overline{y_T}, \overline{y_E})$ and $\overline{\psi(x, x_T, x_E)} = \psi(\overline{x_T}, \overline{x_E}, \overline{x})$.

As P_T and P_E are lists of statements, their adjoint is the reversed list of the statements' adjoint:

$$\text{with} \quad P_T = [s_i \quad \text{for} \quad i = 0 \dots n] \quad \text{then} \quad \overline{P_T} = [\overline{s_i} \quad \text{for} \quad i = n \dots 0]$$

$$\text{and} \quad P_E = [s_i \quad \text{for} \quad i = 0 \dots m] \quad \text{then} \quad \overline{P_E} = [\overline{s_i} \quad \text{for} \quad i = m \dots \quad 0]$$

Adjoint of broadcasts and aggregators

Let's recall that formally, broadcasts and aggregators are matrix multiplication and correspond to the notation $W = MV$, i.e. $W_i = \sum_{j=1}^m M_{i,j} V_j$. Applying the reverse mode on it gives

$$\begin{aligned} W_i &= \sum_{j=1}^{m} M_{i,j} V_j \\ \overline{V_j} &= \frac{\partial f}{\partial V_j} = \sum_{i=1}^{n} \frac{\partial f}{\partial W_i} \frac{\partial W_i}{\partial V_j} \\ &= \sum_{i=1}^{n} M_{i,j} \overline{W_i}, \end{aligned}$$

one can recognize the formula of the matrix multiplication by the transpose of M thus:

$$\begin{aligned} W_i &= \sum_{j=1}^{m} M_{i,j} V_j \\ \overline{V_j} &= \frac{\partial f}{\partial V_j} = \sum_{i=1}^{n} \frac{\partial f}{\partial W_i} \frac{\partial W_i}{\partial V_j} \\ &= \sum_{i=1}^{n} \overline{W_i} M_{i,j}. \end{aligned}$$

This directly gives the adjoint of broadcasts and aggregators as it is a specific case in relational manipulation of matrix manipulation:

$$\begin{aligned} \overline{\langle w \leftarrow v \quad \triangleleft \quad \beta \rangle} &= \langle \overline{v} \leftarrow \overline{w} \quad \triangleright \quad \beta^T \rangle \\ \overline{\langle w \leftarrow v \quad \triangleright \quad \alpha \rangle} &= \langle \overline{v} \leftarrow \overline{w} \quad \triangleleft \quad \alpha^T \rangle \end{aligned}$$

The transpose of a broadcast is an aggregator and vice versa. This matrix point of view can also justify the adjoint of a tupling that can be seen as the multiplication by the $\vec{1}$ vector: the multiplication by its transpose leads to a sum.

Adjoint of a scan

As presented in the introduction of the loop, the scan is a specific instance of this statement. We first present the adjoint of the scan operator to then generalize. To properly define the adjoint of a scan we write it as a Wengert list, i.e. we unroll the loop iteration. The generic unrolled trace of the execution of a scan is the following:

$$\begin{aligned} s_0 &= x_0 \\ s_1 &= f(s_0, a_0) \\ s_2 &= f(s_1, a_1) \\ s_3 &= f(s_2, a_2) \\ &\dots \\ s_n &= f(s_{n-1}, a_{n-1}) \\ s_{n+1} &= s_n \end{aligned}$$

In Adsl it gives

$$\langle For(A \quad \chi \quad false \quad S \quad P \quad \Xi) \rangle$$

with

$$\begin{aligned} \chi &= [\quad x \leftarrow x_0 \quad ; \quad a \leftarrow A \quad] \\ S &= [\quad x \looparrowleft s \quad] \\ P &= [\quad s \leftarrow \quad Callf \quad (x, a) \quad] \\ \Xi &= [\quad s_{n+1} \leftarrow s \quad ; \quad S \leftarrow s \quad] \end{aligned}$$

The scan adjoint is implemented in the JAX [62] automatic differentiation library but is barely documented. We provide a comprehensive way to formalize the adjoint of such operator. Applying reverse mode automatic differentiation on the unrolled trace of the scan gives the following:

$$\begin{aligned}
\overline{s_n} &= \overline{s_{n+1}} \\
\overline{s_{n-1}}, \overline{a_{n-1}} &= \overline{s_n}\,\overline{f}(s_{n-1}, a_{n-1}) \\
&\dots \\
\overline{s_2}, \overline{a_2} &= \overline{s_3}\,\overline{f}(s_2, a_2) \\
\overline{s_1}, \overline{a_1} &= \overline{s_2}\,\overline{f}(s_1, a_1) \\
\overline{s_0}, \overline{a_0} &= \overline{s_1}\,\overline{f}(s_0, a_0) \\
\overline{x_0} &= \overline{s_0}
\end{aligned}$$

One can also wrap it up in another Adsl loop statement:

$$\langle\overline{For(A \quad \chi \quad false \quad S \quad P \quad \Xi)}\rangle = \langle For(A \quad \chi' \quad true \quad S' \quad P' \quad \Xi')\rangle$$

with

$$\begin{aligned}
\chi' &= [\quad a \leftarrow A \quad ; \quad s \leftarrow S \quad ; \quad s'_{n+1} \leftarrow \overline{s_{n+1}} \quad] \\
S' &= [\quad s'_{n+1} \Lsh \overline{s} \quad] \\
P' &= [\quad w \quad \leftarrow \quad Call \quad \overline{f} \quad (s, a) \\
&\qquad s', \overline{a} \leftarrow \quad Mul \quad \overline{s} \quad w \quad] \\
\Xi' &= [\quad \overline{A} \leftarrow \overline{a} \quad ; \quad \overline{x_0} \leftarrow \overline{s} \quad]
\end{aligned}$$

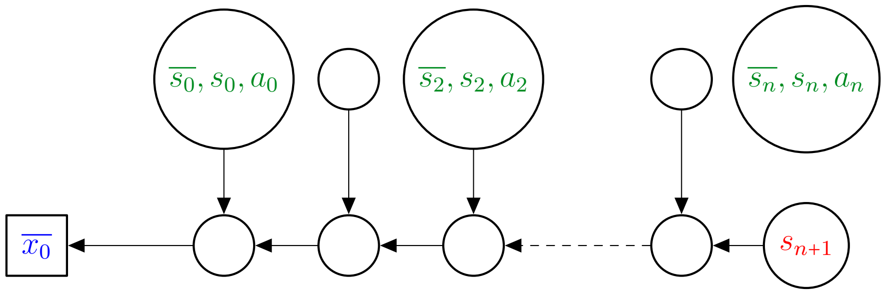

$\overline{s_0}, s_0, a_0$
$\overline{s_2}, s_2, a_2$
$\overline{s_n}, s_n, a_n$
$\overline{x_0}$
s_{n+1}

Figure I.17.: Adjoint of scan Representation. The state s is initialized with s_{n+1} and is updated while iterating the function $\overline{f}$ over the vectors $\overline{S}, S, A$ in reverse order.

The adjoint of a scan is thus another scan, Figure I.17 represents the backward pass on the scan to compute its adjoint. It highlights Adsl closure by differentiation.

Adjoint of loop

Now that the adjoint of a scan has just been detailed, we can tackle the more generic case of a loop in a very formal way.

Let consider an Adsl loop I.18:

$$\langle For(\tau \quad \chi \quad false \quad S \quad P \quad \Xi)\rangle \tag{I.18}$$

with

$$\begin{aligned}\Xi &= [\quad \xi_i : \xi(o_i, i_i, \beta_i) \quad]\\ S &= [\quad s_i : b_i \Lsh e_i \quad]\\ \chi &= [\quad \chi_i : \chi(i_i, o_i) \quad]\end{aligned}$$

Its adjoint is the constructed Adsl loop I.19:

$$\overline{\langle For(\tau \quad \chi \quad false \quad S \quad P \quad \Xi)\rangle} = \langle For(\tau \quad \overline{\Xi} \quad true \quad \overline{S} \quad \overline{P} \quad \overline{\chi})\rangle \tag{I.19}$$

with

$$\begin{aligned}\overline{\Xi} &= [\quad \overline{\xi_i} : \overline{\xi(o_i, i_i, \beta_i)} = \chi(\overline{i_i}, \overline{o_i}) \quad]\\ \overline{S} &= [\quad \overline{e_i} \Lsh \overline{b_i} \quad]\\ \overline{\chi} &= [\quad \overline{\chi_i} : \overline{\chi(i_i, o_i)} = \xi(\overline{o_i}, \overline{i_i}) \quad]\end{aligned}$$

Hard coded adjoints

There are some functions for which we have decided to hard-code their derivative. The set of round/ceiling/floor functions derivative is mathematically zero almost everywhere, while we would like to propagate gradient in it as it is globally increasing as depicted in Figure I.18. Thus we have hard-coded their gradient to be 1.

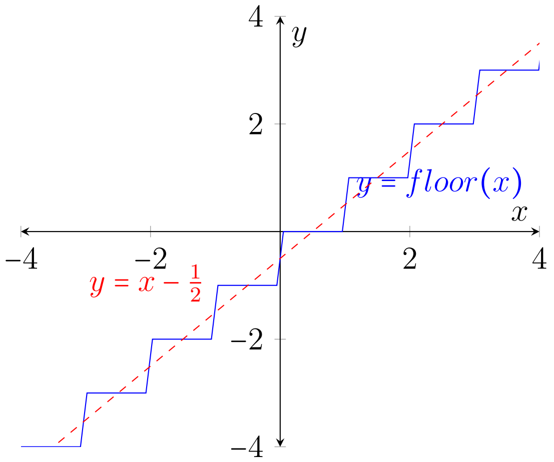


Figure I.18.: The floor function rounds down its input to the nearest integer. Its derivative is zero almost everywhere while its general trend is linear.

Many other differentiation libraries do the same trick in order to enable gradient based optimization with such functions. This is particularly important in supply chain that works with indivisible goods.

```
⟨v        ←   Param   0  ⟩
⟨w        ←   Param   1  ⟩
⟨z        ←   Call   random.normal   v   w  ⟩
⟨Return   z  ⟩
```

Adsl I.2.: Random call.

```
⟨v        ←   Param   0  ⟩
⟨w        ←   Param   1  ⟩
⟨zero     ←   0.0f  ⟩
⟨one      ←   1.0f  ⟩
⟨n        ←   Call   random.normal   zero   one  ⟩
⟨m        ←   Call   mul   n   w  ⟩
⟨z        ←   Call   mul   v   m  ⟩
⟨Return   z  ⟩
```

Adsl I.3.: Adsl program of reparameterization trick.

Adjoint of random functions

Adsl supports many random functions such as *random.normal* or *random.poisson*. Let consider that the random seeds are handled separately,

Let consider Adsl I.2.

Even though we do not have a appropriate adjoint for *random.normal*, we can apply the reparameterization trick from Equations I.20 and I.21, which rely on a rewriting of the gaussian random variable.

$$\frac{\partial .}{\partial \mu}\mathcal{N}(\mu,\sigma) = \frac{\partial .}{\partial \mu}[\mu + \sigma\mathcal{N}(0,1)] = 1 \tag{I.20}$$

$$\frac{\partial .}{\partial \sigma}\mathcal{N}(\mu,\sigma) = \frac{\partial .}{\partial \sigma}[\mu + \sigma\mathcal{N}(0,1)] = \mathcal{N}(0,1). \tag{I.21}$$

In Adsl, we need to rewrite the statements in Adsl I.3:

With this formulation, the differentiation of these statements is now possible. We can apply a similar trick for *random.uniform* but we did not find any suitable solution for the other random functions as *random.poisson*, *random.negativeBinomial* …

Adjoint of predicates conditional

In the previous paragraph **Adjoint of conditional**, we have described that gradient is not propagated through the predicate of a conditional: $\overline{\pi}$ is not used in any sense. This is the mathematically correct implementation of automatic differentiation but it has to be specified to users of differentiable programming on relational data.

Let consider the $isPos$ function that is 1 on $\mathbb{R}^{+}$ and 0 on $\mathbb{R}^{-\star}$, represented in Figure I.19. It is clear that its derivative is zero, thus the update gradient does not apply in order to find a global minimum. Differentiable programming users have thus to recall that gradient is not propagated through the predicate of a conditional. As a consequence, our paradigm is not suited to learn policies via gradient descent and is another extensively studied research subject [26, 85, 86].

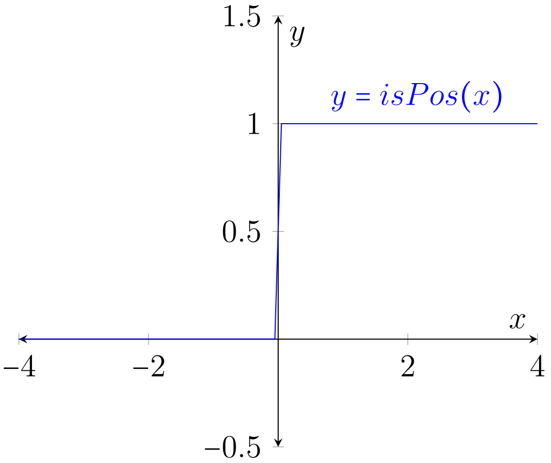


Figure I.19.: $isPos$ function. Its derivative is zero almost everywhere.

Adjoint of real valued function of an integer variable

Consider a function $f_z : \mathbb{Z} \to \mathbb{R}$. While this function is not defined on a continuous space, it is still possible to determine whether it is locally increasing or decreasing, as illustrated in Figure I.20. Additionally, composing this function with the floor function results in a real-valued function:

$$f_z \circ floor : \mathbb{R} \to \mathbb{R}$$

To properly define the local derivative of f_z, we could add such functions to the list of supported ones in $Call$ statements. Let $n \in \mathbb{Z}$ and $y = f_z(n)$. To define $\overline{n}$, we need to consider the previous and next values of f_z. If $f(n-1) \leq f(n) \leq f(n+1)$, then we propose $\overline{n} = \frac{f(n+1)-f(n-1)}{2}$, which is similar to the finite difference presented in Section I.1.1. This same approach applies if $f(n-1) \geq f(n) \geq f(n+1)$. However, it becomes different when $f(n)$ is a local extremum. Let's assume that n is a local minimum of f_z. If $\overline{y} \geq 0$, then we are on a local minimum of the objective function, and $\overline{n} = 0$. However, if $\overline{y} < 0$, then we are on a local maximum of the objective function, and we construct the gradient in the strongest direction. Specifically, if $f(n+1) \geq f(n-1)$, then $\overline{n} = f(n+1)$. Otherwise, $\overline{n} = -f(n-1)$. We sum this strategy in Figure I.21. It should be noted that this logic can only be implemented in reverse mode as it requires knowledge of the sign of $\overline{y}$. This consideration further justifies our decision to select this particular mode.

Writing this logic in Adsl involves many nested conditional and is not digest to read. Consequently we do not propose its statement version and stick to the previous description of the adjoint construction.

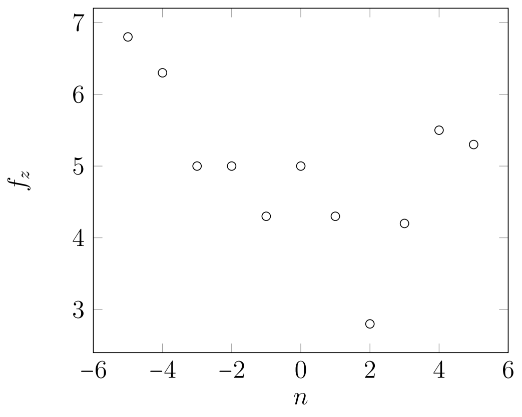
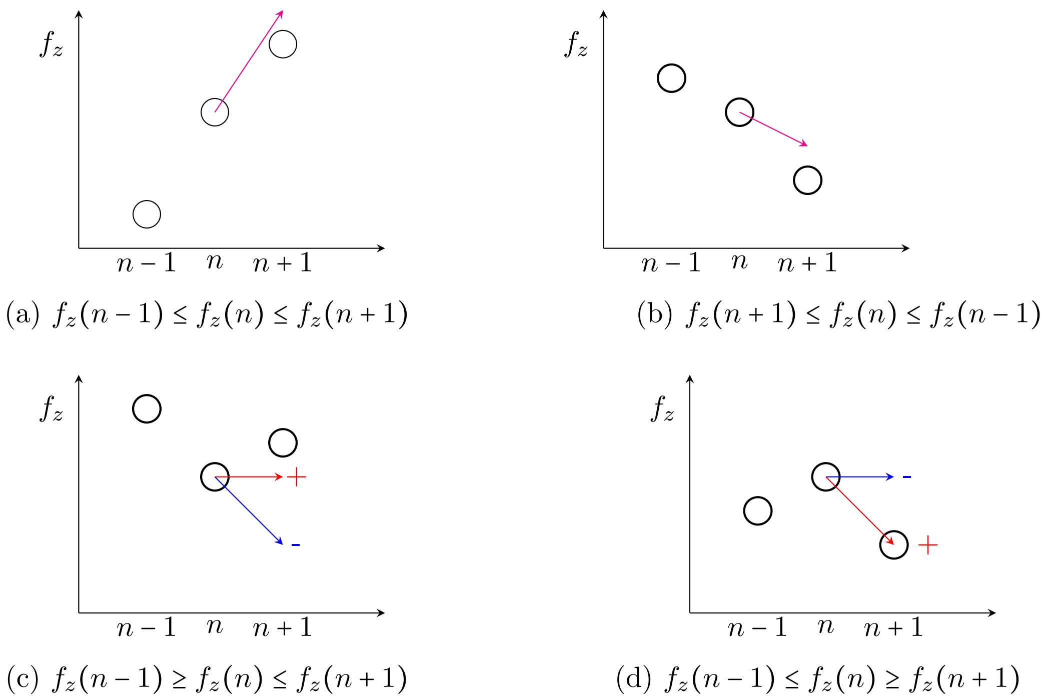


Figure I.20.: Representation of a real valued function of an integer variable.



(a) $f_z(n-1) \leq f_z(n) \leq f_z(n+1)$

(b) $f_z(n+1) \leq f_z(n) \leq f_z(n-1)$

(c) $f_z(n-1) \geq f_z(n) \leq f_z(n+1)$

(d) $f_z(n-1) \leq f_z(n) \geq f_z(n+1)$

Figure I.21.: Adjoint of the real valued function of an integer variable in function of the surrounding values and the backpropagated adjoint. The adjoint is represented by the slope of the gradient arrows. We display the arrows in red when $\overline{f_z(n)} \geq 0$, in blue when $\overline{f_z(n)} < 0$, in magenta otherwise.

Adsl closure by differentiation

All the previous adjoints are built with other Adsl statements and a comprehensive summary is given in Grammar I.2. The derivative of an Adsl program being another Adsl program has two main consequences.

First, this closure gives automatic access to higher order derivatives. Gradient based methods, like presented in Section II.1, often rely on the first order of the gradient but some higher one exists: [87] [88]. These methods are very costly in terms of resources but they can be implemented with our automatic differentiation system as we can apply the differentiation process to the adjoint program.

$$\frac{\partial^2 P}{\partial \theta^2} = \frac{\partial}{\partial \theta}\frac{\partial P}{\partial \theta}.$$

Second, the adjoint program being considered as a regular Adsl one, all the compiler optimizations like dead code elimination provided with Adsl are available. The compiled codes of the function and the derivative through the execution process follow the same pipeline. This is particularly useful in relational programming languages where query optimization is an extensive subject [89] [90]. The adjoint of a query being a query makes our automatic differentiation system composable with any other automatic differentiation architecture querying a database for example. For this reason we also implemented the forward mode to make it compatible with automatic differentiation systems relying on this mode.

$$\begin{array}{ll}
\langle S\rangle ::= & \\
& \mid \langle v \leftarrow e\rangle \\
& \mid \langle tup \leftarrow v\rangle \\
& \mid \langle Cond(v \quad \Psi \quad P_T \quad P_E \quad \Phi)\rangle \\
& \mid \langle For(\tau \quad \chi \quad rev \quad S \quad P \quad \Xi)\rangle \\
& \mid \langle Return \quad v\rangle
\end{array}
\qquad
\begin{array}{ll}
\overline{\langle S\rangle} ::= & \\
& \mid \langle \bar{e} \leftarrow \bar{v}\rangle \\
& \mid \langle \overline{v} \leftarrow \oplus \quad \overline{tup}\rangle \\
& \mid \langle Cond(\pi \quad \overline{\Phi} \quad \overline{P_T} \quad \overline{P_E} \quad \overline{\Psi})\rangle \\
& \mid \langle For(\tau \quad \overline{\Xi} \quad true \quad \overline{S} \quad \overline{P} \quad \overline{\chi})\rangle \\
& \mid \langle \overline{v} \leftarrow 1.0f\rangle
\end{array}$$

$$\begin{array}{ll}
\langle e\rangle ::= & \\
& \mid \langle v\rangle \\
& \mid \langle w\rangle \\
& \mid \langle \oplus \quad tup\rangle \\
& \mid \langle Call \quad op \quad tup\rangle \\
& \mid \langle Param \quad i\rangle \\
& \mid \langle Const \quad i\rangle \\
& \mid \langle b \quad \triangleleft \quad \beta\rangle \\
& \mid \langle a \quad \triangleright \quad \alpha\rangle \\
& \mid \langle Pred\rangle
\end{array}
\qquad
\begin{array}{ll}
\overline{\langle e\rangle} ::= & \\
& \mid \langle \overline{v}\rangle \\
& \quad \varnothing \\
& \mid \langle \overline{tup}\rangle \\
& \mid \langle Call \quad \overline{op} \quad tup \quad \overline{tup}\rangle \\
& \mid \langle Return \quad \overline{v}\rangle \\
& \quad \varnothing \\
& \mid \langle \bar{b} \quad \triangleright \quad \beta^T\rangle \\
& \mid \langle \bar{a} \quad \triangleleft \quad \alpha^T\rangle \\
& \quad \varnothing
\end{array}$$

Grammar I.2.: Summary of the Adsl Grammar I.1 and the corresponding adjoint for each statement. The adjoint for each statement is described in detail in the respective paragraph above. The reverse pass of the derivative for an Adsl program, represented as a list of statements, consists of the adjoint lists associated with each statement.

I.5. Application

In this section we present the results of the application of automatic differentiation in a domain specific language: Envision. To illustrate how Envision treats Differentiable programming as a first-class citizen we propose two examples of optimization thus enabled.

I.5.1. Envision: a domain specific language

Envision is a specialized Domain Specific Language (DSL) developed by Lokad that focuses on supply chain optimization. Its user-friendly design and targeted domain-specific features make it an effective tool for domain experts to model and manage complex supply chain scenarios. There is an open version of Envision[5] and its documentation is public[6].

It is a Python-like implementation of SQL. At a higher level, Envision combines aspects of array programming and relational algebra. It is designed for conducting operations on relational databases commonly used in enterprise systems. Additionally, Envision abstracts the need for loops, prioritizing performance and reliability, as observed in array programming languages. In Envision, each variable corresponds to a table and technically represents a vector, which is a collection of values with one value per row in the table. Consequently, when two variables belong to the same table, it becomes possible to execute operations on all rows simultaneously. Notably, the Envision compiler is predominantly written in F#. An illustration of the simplified[7] pipeline is presented in Appendix in Figure A.1.

Each variable in Envision belongs to a specific table. To enhance conciseness and clarity, the table name can be omitted when referring to the scalar table that plays a specific role in Envision, as highlighted in Listing I.11.

```
Scalar.greeting = "HelloWorld!"
/// is equivalent to
greeting = "HelloWorld!"
```

Listing I.11: First Envision script, which stores a text value into a created attribute of the Scalar table.

In the general case, the table name precedes the variable name, and the dot (.) serves as a separator between them. In relational algebra the variable name is considered as an attribute of the table. One of the main specificity and advantages of Envision is that the tables and the relationships between them are reified. Primary and foreign keys characterize the relational structure of the tables processed by Envision. With the proper dimensions in place between the tables of interest, most operations can be performed with little syntactic overhead. This approach differs from the more traditional approach taken by query languages, joins between tables are thus implicit. The tables and their keys are handled as a whole, which makes TOTAL JOIN implicit between any tables T_1 and T_2 where the primary key of T_1 is a secondary key of T_2. Of course the other one can be requested with a specific semantic. This feature makes the Envision code writing very light and prevents many broadcasts errors. One of the key points is that all these relationships are known at compilation time and the user do not need to wait for the

[5]https://try.testing.lokad.com/

[6]https://docs.lokad.com

[7]Between Envision and its execution, there exist nearly 10 intermediate programming languages, all of which have been developed by Lokad.

execution errors to adapt its query. On the contrary, SQL implementations create many intermediate tables on the fly, every Envision table is reified and directly queryable.

In Listing I.12 we present how the Upstream table can be created from the Observations table thanks to a simple Aggregation of one of its non primary keys, which becomes a foreign one with the new table creation. This relationship between the two tables is reified and let the TOTAL JOIN be implicit.

```
/// Creation of the Upstream table based on the Category
/// vector from the table Observations:
table Upstream[Category] = by Observations.Category

/// Creation of a new attribute in the Upstream table
Upstream.NewAttribute = ...

/// Creation Observations.NewAttribute from a
/// TOTAL JOIN broadcast of Upstream.NewAttribute
Observations.NewAttribute = Upstream.NewAttribute
```

Listing I.12: Broadcasts in Envision.

I.5.2. Differentiable programming as a first-class citizen

Differentiable programming is a first-class citizen in Envision. We have seamlessly incorporated automatic differentiation within the Envision compiler, ensuring that its integration with Adsl is an integral part of the language's lowering process. In that sense, this approach aligns closely with the compiler-based approach discussed in Section I.1.3, further emphasizing the significance of automatic differentiation in Envision. By doing so, we statically detect all the differentiation related errors and display errors statically. It means that the user does not have to be (too) careful while writing Envision in order to get access to the differentiation as every static mistake is notified. This is very interesting as the main users of this (and the other) relational programming language, are supply chain practitioners that have a deep understanding of supply chain complexity but do not master all the gradient theory. Their expertise makes them the perfect users of Differentiable programming as they can implement relevant white box models to solve their daily problems. This position is quite new. There was an attempt to make Swift [67] one of the first popular programming languages with Differentiable programming as a feature, but the project was recently abandoned. As presented in Section I.3, a query is mathematical and relational. In that sense, any mathematical operation is supported in the optimization blocks. For the relational part, the only limitation is to satisfy the PolyStar form of Definition 6 for the used tables.

In the following we present two applications of differentiable programming through Envision. The first one is a relational linear regression example. The second one is a model used daily at Lokad to forecast demand.

I.5.3. Relational linear regression and Bayesian inference

Relational linear regression

Listing I.13 implements the relational linear regression model in Envision. The *autodiff* keyword opens the automatic differentiation block code. This block not only performs automatic differentiation of the query but also conducts end-to-end optimization by utilizing stochastic gradient descent thanks to the automatic access to the gradient query. In essence, *autodiff* offers more than just automatic differentiation.

As described in Section I.3.4, the observation node corresponds to the Observation table, where each variable in the block is observed at the Observation level. This implies that the vectors in the Observation table and the Upstream table can be broadcasted into scalar variables, as the Polystar rules apply in the *autodiff* blocks. It is important to note that the Category table, previously referred to as Table I.1, is now referred to as the Upstream table in order to maintain coherence with PolyStar terminology.

This example was presented in [77] applied on the highly categorical dataset of Chicago taxi rides [91].

```
 1  autodiff Observations epochs:10 learningRate:0.01 with
 2      params Upstream.a auto
 3      params b          auto
 4
 5      /// Relational
 6      a = Upstream.a
 7      X = Observations.X
 8      Y = Observations.Y
 9
10      /// Mathematical
11      prediction = a * X + b
12      error = prediction - Y
13
14      return error^2
```

Listing I.13: The autodiff block in Envision exemplifies the implementation of relational linear regression. In this block, stochastic gradient descent is executed for 10 epochs (line 1), utilizing the Adam optimizer with its default parameters (presented in Chapter II). Firstly, parameters are created, including the vector a in the Upstream table and the scalar b (lines 2-3). Subsequently, for each epoch, the block iterates over the Observations table. For every line in this table, the gradients of Upstream.a and b are computed, and Adam is employed to update them (performed on each line). The output of the block is the updated version of the 'Upstream.a' and 'b' parameters. It is worth noting that all other intermediate variables generated during the autodiff block are not preserved.

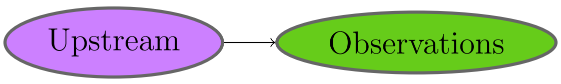


Figure I.22.: PolyStar of the relational linear regression example. By construction in the Listing I.12, the primary key of the Upstream table is a foreign key of the Observations table, which is a sufficient condition to allow the depicted broadcast. In other words, each line of the Observations table corresponds to a unique line in the Upstream table, allowing access to the appropriate line of the slope vector.

As presented in Section I.3 this query is both relational and mathematical. The relational aspect is the broadcast from the Upstream table into the Observations table and is easily presented in the PolyStar from Figure I.22. The mathematical part is the formula of the raw linear regression, and the choice of the l^2 norm.

In their works, [11, 74] suggest evaluating their in-database machine learning systems using regular linear regressions. However, we posit that a relational linear regression would be a more relevant example for assessing the performance of these proposed systems. This is because relational programming languages provide the simplest framework for constructing this type of categorical model. Implementing such a model in PyTorch or TensorFlow would not align naturally, whereas relational programming languages are well-suited, as demonstrated in Listing I.13. A standalone version of this code, ready for execution[8], is provided in Appendix Listing A.1. The additional version in this listing can assist readers who are interested in gaining a better understanding of Envision functionalities. One can also find the complete documentation of the autodiff block here[9].

Bayesian inference

Bayesian inference is a method of statistical inference in which Bayes' theorem is used to update the probability for a hypothesis as more evidence or information becomes available. It is an approach that allows us to quantify uncertainty in a principled way, incorporating prior knowledge and observed data to make predictions about unobserved quantities. In the context of machine learning, Bayesian inference has a number of desirable properties. Firstly, it provides a coherent framework for dealing with uncertainty, which is crucial in many real-world applications like supply chain [92]. For instance, in supply chain optimization, it is often the case that we have incomplete or noisy data, and Bayesian methods allow us to quantify and manage this uncertainty in a structured way. Furthermore, Bayesian methods naturally incorporate prior knowledge into the learning process, which has similarities with differentiable programming where domain experts play a crucial role.

In order to extend the work on relational linear regression, we will exploit the power of automatic differentiation to perform Bayesian inference. For a Bayesian linear regression model, instead of estimating point values for the parameters (slope and intercept), we aim to estimate their posterior distributions given the observed data. This requires us to define a prior distribution over these parameters. Common choices include Gaussian or Laplace distributions, which are both differentiable and hence suitable for automatic differentiation. Opting for the Gaussian distribution enables us to utilize its closed-form log-likelihood, which is inherently differentiable. This is possible as it can be readily implemented in pure Envision code, which is fully differentiable.

The posterior distribution of the parameters, given the observed data, is then proportional to the product of the likelihood of the data given the parameters and the prior distribution of the parameters. The likelihood is defined by the Gaussian distribution centered on the linear function of the inputs, parameterized by the slopes and the shared intercept, with a standard deviation.

This approach allows us to leverage the power of Bayesian inference in relational linear regression. Not only do we obtain a point estimate of the parameters, we also get a measure of uncertainty in the form of the posterior distribution. This can be crucial in many applications where it is important to quantify uncertainty in predictions. The use of automatic differentiation makes this approach scalable to large datasets and multiple

[8]at `https://try.lokad.com/s/Peseux-PhD-RLRegression`

[9]`https://docs.lokad.com/language/differentiable-programming/`

models. Listing I.14 illustrates the Bayesian relational linear regression model fit to the same toy dataset. Incorporating uncertainty through Bayesian inference is key in supply chain problems where a quantitative approach is a game changer [93, 94].

```
autodiff Observations epochs:10 learningRate:0.01 with
    params Upstream.a auto
    params b          auto
    params sigma      auto

    /// Relational
    a = Upstream.a
    X = Observations.X
    Y = Observations.Y

    /// Mathematical
    prediction = a * X + b

    /// Bayesian
    return - loglikelihood.normal(prediction, sigma, Y)
```

Listing I.14: autodiff block in Envision of the relational linear regression example in its Bayesian version.

I.5.4. Production example: forecasting retail

Differentiable programming on relational programming languages is promising and lets us build more complex models than relational linear regression.

We have successfully deployed to production the model presented in Figure I to forecast weekly sales of Celio, a large retail company. Sales forecasting is vital for businesses because it enables effective demand planning, inventory management, supply chain optimization, financial planning, sales and marketing strategies, risk management, resource allocation, performance evaluation, and informed decision making. It helps businesses optimize operations, allocate resources wisely, mitigate risks, and achieve growth and profitability.

Lokad experts have used the framework we have developed to create this model and applied it on a dataset that contains 3 years of history and concerns $100k$ different items. The dataset contains multiple categorical inputs for each item. The objective is to forecast sales at the item level as it is more important than at an aggregated level. It provides businesses with granular insights and enables more accurate demand planning and inventory management. By forecasting sales at the individual item level, businesses can understand the demand patterns and fluctuations specific to each product. This allows for better allocation of resources, optimized production, efficient inventory stocking, and targeted marketing strategies. It also helps identify top-performing products, optimize pricing strategies, and identify underperforming products that may require adjustments or promotions.

We recall the form of the model for each item i in the dataset:

$$\hat{y}(i, week) = \theta_{store(i)} \times \theta_{color(i)} \times \theta_{size(i)} \times \Theta[group(i), WeekNumber(week)].$$

The goal is to generate a vector in the ItemsWeek table that closely approximates

$$\begin{array}{lllllll}
\langle\theta_{store} & \leftarrow & Param & 0 & \rangle & & \\
\langle\theta_{color} & \leftarrow & Param & 1 & \rangle & & \\
\langle\theta_{size} & \leftarrow & Param & 2 & \rangle & & \\
\langle\Theta_{WN} & \leftarrow & Param & 3 & \rangle & & \\
\langle y & \leftarrow & Const & 0 & \rangle & & \\
\langle\theta^{WN}_{store} & \leftarrow & \theta_{store} & \lhd & \beta^{WN} & \rangle & \\
\langle\theta^{WN}_{color} & \leftarrow & \theta_{color} & \lhd & \beta^{WN} & \rangle & \\
\langle\theta^{WN}_{size} & \leftarrow & \theta_{size} & \lhd & \beta^{WN} & \rangle & \\
\langle\hat{y}^{WN} & \leftarrow & Call & mul & \theta^{WN}_{store} \quad \theta^{WN}_{store} \quad \theta^{WN}_{size} \quad \Theta_{WN} & \rangle & \\
\langle\hat{y} & \leftarrow & \hat{y}^{WN} & \lhd & \beta^{W}_{WN} & \rangle & \\
\langle E & \leftarrow & Call & minus & \hat{y} \quad y\rangle & & \\
\langle E^2 & \leftarrow & Call & square & E & \rangle & \\
\langle loss & \leftarrow & E^2 & \lhd & \alpha & \rangle &
\end{array}$$

Adsl I.4.: Adsl program of the categorical model depicted in Equation I.2. The broadcast β^{WN} simply duplicates a scalar value into the WeekNumber table, while the broadcast β^{W}_{WN} broadcasts the value into the Week table on the corresponding week number.

the existing ItemsWeek.Target values on historical data. Note that this model could be formalized as a small neural network with matrix multiplications but it would lose its simplicity of understanding. The Polystar associated with such a model is depicted in Figure I.23. We choose the *Items* table to be the observation table. Thus the tables *Store*, *Color*, *Size* and *Group* are considered as *upstream* tables because their primary keys are foreign keys of *Items*. The tables *WN* and *Week* have no relationships at all with *Items*, they are considered as *full* tables. *GroupWN* is a cross table between an *upstream* table (upstream-cross table) and a *full* table (*WN*), thus it is an *upstream-cross* table. The same logic also turns *ItemsWeek* into an *upstream-cross* table. The Adsl program of the model is written in Adsl I.4.

Such model has the following number of parameters:

$$|\,Store\,| + |\,Color\,| + |\,Size\,| + |\,Group\,| \times 52.$$

Let recall that our objective is to apply Stochastic Gradient Descent on the *observation* table. Hence for each line of the *observation* table, there corresponds one and only one line in the *upstream* tables thanks to the TOTAL JOIN operator. It motivates the natural broadcasts from line 8 in Listing I.15.

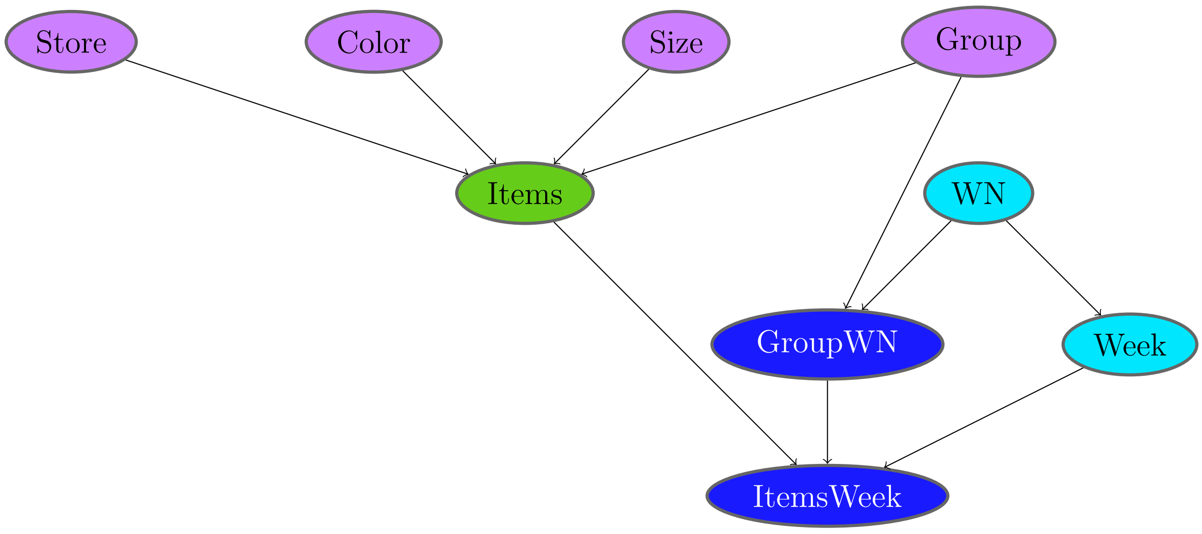


Figure I.23.: PolyStar from Celio's application. Items is the observation table and all the other tables are defined by their relationship with it. The upstream tables broadcast into Items as their primary keys are foreign keys for Items.

```
autodiff Items epochs:10 learningRate:0.01 with
    params Store.theta   in [epsilon ..] auto
    params Color.theta   in [epsilon ..] auto
    params Size.theta    in [epsilon ..] auto
    params GroupWN.theta in [epsilon ..] auto

    /// Relational
    thetaSt, thetaC, thetaSi = Store.theta, Color.theta, Size.theta
    ItemsWeek.thetaGroup = GroupWN.theta
    /// Mathematical
    ItemsWeek.Prediction = thetaSt * thetaC *
                           thetaSi * ItemsWeek.thetaGroup

    /// Relational
    Week.Prediction = ItemsWeek.Prediction
    /// Mathematical
    Week.Error = Week.Prediction - ItemsWeek.Target
    Week.Error2 = Week.Error^2

    /// Relational
    loss = sum(Week.Error2)
    return loss
```

Listing I.15: The autodiff block in Envision implements the multiplicative model on Celio data. This model is end-to-end differentiable, including the implicit broadcasts that are natively supported by Envision. The exhaustive code is proposed in Appendix Listing A.2

In order to retrieve the *color* parameter related to a specific item, one can use the TOTAL JOIN operator as presented in Listing I.16.

```
SELECT Color.theta
    FROM Items
    TOTAL JOIN Color
        ON Items.Color = Color.Color
```

Listing I.16: TOTAL JOIN to retrieve Color.θ in SQL code.

Advantages of this model

Such an approach has multiple advantages. First this is a white box model and its predictions can be explained as its parameters convey meaning. For the color vector $\theta_{color}(red) > \theta_{color}(blue)$ has the direct translation that the red clothes sell better than the blue ones. This is detailed in Section III.1.2. Second, such a model can be used to predict unseen items as the full vectors Store.θ, Group.θ and Size.θ are learned and a new item giving an unseen combination can be predicted yet. Third, it is very adaptive, which is one of the main advantages of differentiable programming. If one wants to take into account a new category as the type of the item (pants, shirts . . .), this is super easy and one does not need to restart the optimization from the beginning as it is a multiplicative model. For example one can easily take into accounts ambient factors like discounts or marketing campaigns by modifying the model. Thanks to automatic differentiation one can tweak the model without considering the generated gradient code. Lastly, the small number of parameters, compared to deep learning, makes their optimization relatively fast. Thus this model can be updated daily with the new data without consuming too many resources.

Performance

An alternative version of this model is implemented at Lokad. The model leveraged stochastic gradient descent optimization to process over 1.1 billion observations, iterating ten times on the full dataset. Even with 4,000 parameters to update, the optimization process took only 350 seconds, demonstrating computational efficiency.

In another retail forecasting model implemented for another client, which we cannot disclose due to confidentiality reasons, over a million parameters are updated in each epoch consisting of more than 100,000 observations. The process of each epoch takes approximately a dozen seconds. An even faster version based on parallelism [95] has been developed and unlocked optimization on even larger datasets.

I.5.5. Scan operator

In the production example, our approach involves creating an estimated vector of sales in the Week table and then comparing it week by week with the true sales vector. Although this is a valid approach, it has its disadvantages. Let's assume that sales are sparse and occur only once a year. Forecasting sales for this week with an error of one week is better than forecasting with an error of ten weeks. However, in both cases, the obtained loss is the same. An alternative method for computing the loss may be needed to address this issue.

If one can construct a cumulative vector of the sum of sales, comparing the cumulative vectors would yield the desired behavior: being one week late in the forecast would not be as penalized as being ten weeks late. When introducing the statements in Adsl, we identified the scan operator as a special case. It could potentially provide a solution to this issue. By scanning both the estimated and true sales vectors, one can compute the cumulative sum. An example of this approach based on an each block implemented in Envision is provided in Listing I.17.

```
autodiff Items epochs:10 learningRate:0.01 with
    params Store.theta   in [epsilon ..] auto
    params Color.theta   in [epsilon ..] auto
    params Size.theta    in [epsilon ..] auto
    params GroupWN.theta in [epsilon ..] auto

    /// Relational
    thetaSt, thetaC, thetaSi = Store.theta, Color.theta, Size.theta
    ItemsWeek.thetaGroup = GroupWN.theta

    /// Mathematical
    ItemsWeek.Prediction = thetaSt * thetaC *
                        thetaSi * ItemsWeek.thetaGroup

    /// Relational
    Week.Prediction = ItemsWeek.Prediction
    Week.Target = ItemsWeek.Target

    wcp = 0
    wt = 0
    Week.CumulativePrediction, Week.CumulativeTarget =
      each Week scan week
        keep wcp
        keep wt
        wcp = wcp + Week.Prediction
        wt = wt + Week.Target
        return (wcp, wt)

    /// Mathematical
    Week.Error = Week.CumulativePrediction - Week.CumulativeTarget
    Week.Error2 = Week.Error^2

    /// Relational
    loss = sum(Week.Error2)

    return loss
```

Listing I.17: Alternative version of autodiff block implementing the multiplicative model in Envision on Celio data. The scan operator is differentiable and enables the construction of a refined loss based on cumulative sum.

The distinction between the models presented in Listings I.15 and I.17 is subtle but can have a significant impact on the final results. We firmly believe that employing a white-box and customizable model, guided by domain experts, will be a game-changer in the industry. Such models will be more adaptable, and experts will no longer perceive modifications to a production model as an insurmountable task. This will foster a culture of continuous improvement. Our belief in this approach is strongly reinforced by the results we have witnessed at Lokad through the utilization of relational query differentiation.

I.5.6. Mathematical insights

On multiplicative models

Such model is multiplicative, we have tested other forms like the additive one inspired by the Prophet library [96]. The presented model is the most effective one we designed but one has to be careful implementing it. One of the main issues with such a multiplicative model is the risk of having parameters moving towards 0. If $\theta^n_{store} \sim \frac{1}{n}$ and $\theta^n_{color} \sim n$ then $\theta^n_{store} \times \theta^n_{color} \sim 1$. This kind of numerical divergence of the parameters but not of the prediction makes the model very unstable. Then we force parameters to be superior to a fixed ϵ: this is the sense of the boundaries in the parameter definition. This simple trick makes the learning way more stable. Moreover, well initializing the parameters of a model is a key point of the optimization success. In the present code, parameters are initialized by default which does not give great results in practice.Thus we develop a more elaborated approach in Section III.3.4.

On adaptive optimizers

Optimizers are properly introduced in Section II.3.1

To perform optimization, the gradient alone is not enough, an optimizer is also required. Optimizers are algorithms that update a parameter based on its gradient. Some of them, such as AdaGrad and Adam, are described as adaptive because they can handle multiple regimes of gradient values, making them ideal for use with differentiable programming in relational programming languages. This is especially important when models are built by domain experts who may have limited knowledge of gradient-based techniques. In such cases, it is important for non-problem related issues to be handled automatically, making the use of adaptive optimizers necessary.

The Envision optimization blocks use the Adam optimizer with its default values for parameter updating. After extensive testing, we found that this optimizer performed the best overall and showed good results for a variety of problems. One important observation we made while using this optimizer in production is that since the updates for Adam's parameters are approximately bounded by the learning rate, the parameters should be on the same scale. Specifically, we expect the amplitude (amp) between the minimum and maximum values of interesting parameters to respect the following relation for each parameter:

$$amp \leq 2 \times \alpha \times updates. \tag{I.22}$$

Here, α is the learning rate and $updates$ is the total number of updates. For a parameter in the observation table, $updates$ is equal to the number of epochs, while for raw scalar parameters it is equal to $size(Obs) \times epochs$.

In order to be sure that the scales of the parameters matches, one can apply scaling operators like affine transformation or exponential one:

$$\hat{y}'(i, week) = e^{\theta'_{store(i)}} \times e^{\theta'_{color(i)}} \times e^{\theta'_{size(i)}} \times \Theta[group(i), WeekNumber(week)]. \tag{I.23}$$

Conclusion

This chapter introduced automatic differentiation to relational programming languages and thus unlocked Differentiable programming on those specific languages.

After a theoretical presentation of relational algebra, we have demonstrated the crucial need of machine learning tools in database systems themselves as the existing solutions do not leverage the relational aspect of the data they work with. Our approach is to separate the relational and the mathematical aspect of a relational query in order to differentiate it. To do so we have crafted a dedicated programming language, Adsl. This has several advantages. First it removes all the language specific issues while implementing the differentiation system. Second, as we are free to design it, we did it in order to make anything related to differentiation easy. Thus Adsl has two simple properties: SSA and SA. It makes its differentiation very easy and natural. The SSA property is pretty common for programming languages but the SA property gives a specific form to the final gradient and has led to a novel gradient estimator described in Chapter IV. Finally Adsl's closeness by differentiation expresses the derivative of a query as another query. This property lets us express the gradient's query in the same environment as the original one: it benefits from the same optimizations before its execution. Moreover it makes it compatible with any programming language that can be compiled from and to. If a differentiable framework uses a query at some point, Adsl can return the derivative query that can be plugged into the creation of the model's gradient. In order to translate relational queries into Adsl programs, we have defined the PolyStar and the novel Join operator named TOTAL JOIN. These two new concepts are key in order to build a strong theoretical setup. Differentiation relational queries unlocks many possible applications, especially in order to build white box models. Thus from the practical side, we have compiled Adsl from and to Envision, that is a direct implementation of what we have just described. The solid theoretical work makes his execution very fast and reliable. This has led to white box models designed by supply chain experts that have outperformed black box models in daily production at Lokad. We have unlocked automatic differentiation on relational queries in order to optimize models through gradient descent. Applying gradient descent on relational data raises many issues that are tackled in the Chapter III.

To sum up, the main insights of this chapter are the following. First machine learning is wished for and possible in database systems. Second, the gradient of a query is also a query.

Now that we have a solid and implemented access to the gradient of a query, we present what we do with, i.e. stochastic gradient descent.

II. Stochastic gradient descent

The development of Adsl within a relational programming language like Envision serves the purpose of optimizing white box models using gradient descent. This chapter aims to provide a comprehensive understanding of Stochastic Gradient Descent (SGD) [97] and its practical applications in training machine learning models. This framework will fit the mathematical and relational decomposition of the query and will serve as the theoretical basis for Chapters III and IV.

Introduction

Gradient descent is a widely used optimization algorithm in machine learning that has played a significant role in recent advancements in training a variety of models. The algorithm is efficient, intuitive, and can be extended to different learning tasks through the use of different loss functions. An alternative version of this algorithm, SGD, performs gradient descent with an estimator of the full gradient and requires fewer resources. This feature makes SGD much more scalable than traditional gradient descent, as one of its versions only requires a small subset of the data to be loaded into memory at a time. Additionally, the chapter presents a unified perspective on the adaptive optimizers introduced by [98] and their utilization in the context of SGD. The proposed approach offers convergence guarantees for non-convex functions, leveraging the unbiasedness of the estimator.

The organization of this chapter is as follows: firstly, a brief analogy of gradient descent is presented, followed by the proper mathematical framework that supports this technique. The efficiency of the method is proved in the convex case and convergence guarantees are obtained under regularity assumptions. In the second part of the chapter, different approaches to introducing stochasticity in gradient descent are discussed. Finally we present a selection of highly effective optimizers suitable for performing in the given context and how they can be unified to give convergence guarantees under other regularity conditions.

II.1. Gradient descent

II.1.1. Analogy

We present a popular analogy that provides intuition for the mechanism of gradient descent. Consider a hiker on a mountain who wishes to descend to the valley. However, due to dense fog, she can only see a distance of one meter around her. As the valley is situated at a lower altitude than the mountain, she is likely to reach the valley if each of her steps follows the direction with the steepest descent around her. Although this method may fail if she ends up in a local minimum such as a mountain lake, assuming that the mountain topology has no local minima, she will reach the valley. The time

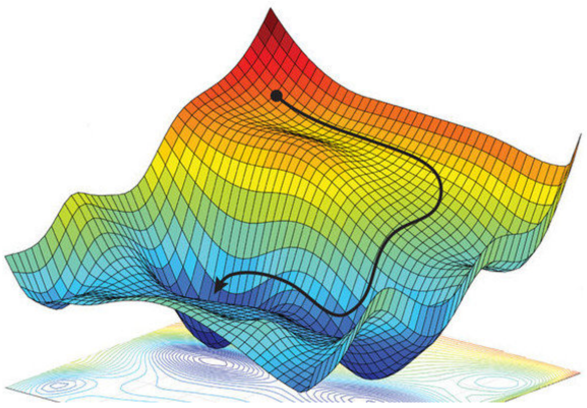


Figure II.1.: Gradient descent illustration on a representation of an arbitrary function $f : \mathbb{R}^2 \longrightarrow \mathbb{R}$. The parameter update following gradient descent allows the diminution of the objective function, as depicted by the decrease in the vertical location of $(x, y, f(x, y))$. Illustration directly taken from [99].

taken for her to reach the valley will depend on the length of her steps. If the steps are too small, she will require many steps to reach the valley. Conversely, if the steps are too big, she may head in the wrong direction as the direction with the steepest descent changes throughout the mountain.

In this analogy, the location of the hiker corresponds to the model parameters, her altitude represents the objective function, and each step decision represents the optimization algorithm. A comprehensive representation of gradient descent is thus given in Figure II.1. It may appear counterintuitive, but certain gradient descent methods do not strictly adhere to the direction with the steepest descent. This is due to the fact that selecting a smaller slope locally may be compensated for by repeating it multiple times.

This analogy is useful but now let's properly introduce gradient descent.

II.1.2. Notations

Let's consider F_θ the function from $\mathbb{R}^p$ to $\mathbb{R}$ that we aim to minimize. Our objective is to find $\theta^\star$, which we assume to exist:

$$\theta^\star = \underset{\theta \in \mathbb{R}^p}{\arg\min} \quad F_\theta.$$

All the work presented in Chapter I provides an automatic means to access the gradient of functions computed in relational programming languages. Section I.1.3 presented automatic differentiation systems on other environments like Pytorch [40] or Julia [39], thus we can assume that if F_θ is differentiable with respect to θ, i.e., $\nabla_\theta F$ exists, we can access to it. We can introduce gradient descent which is a first-order iterative optimization algorithm commonly used to find a local minimum of F_θ. The main concept behind this widely-used technique is to iteratively move the parameter in the opposite direction of the gradient $\nabla_\theta F$, which by definition is the direction of the steepest descent of the objective function.

The iterative step of gradient descent is:

$$\theta_{t+1} = \theta_t - \alpha_t \nabla_\theta F(\theta_t), \tag{II.1}$$

α_t is the learning rate. Using an appropriate learning rate is key in order to converge towards a minimum of the target function. The update, i.e. $\theta_{t+1}-\theta_t$, is called the stepsize.

In addition to its simplicity in formalization and comprehension, gradient descent provides convergence guarantees for smooth and convex optimization problems. These guarantees are presented in the subsequent section.

II.1.3. Convergence proof in the smooth and convex case

We recall the convergence proof of the gradient descent algorithm in the smooth and convex case and we follow [100] presentation. For this section, we suppose that F_θ is smooth and convex.

F_θ is L-smooth means that

$$\forall \quad \theta, \theta'; \quad F(\theta') \leq F(\theta) + \langle \nabla F(\theta); (\theta' - \theta) \rangle + \frac{L}{2} \|\theta' - \theta\|^2, \tag{II.2}$$

first note that the L-smoothness of F gives the uniqueness of $\theta^\star$. In this set up, a constant learning rate $\alpha_t = \frac{1}{L} \in \mathbb{R}^{+\star}$ gives us convergence properties.

Theorem 2 (gradient descent converges). *If F_θ is L-smooth and $\alpha_t = \frac{1}{L}$, then*

$$F(\theta_t) - F(\theta^\star) \leq 2L \frac{\|\theta_1 - \theta^\star\|^2}{t}. \tag{II.3}$$

Proof. F being L-smooth, one can apply II.2 with $\theta' = \theta_{t+1}$ and $\theta = \theta_t$:

$$F(\theta_{t+1}) \leq F(\theta_t) + \langle \nabla F(\theta_t); \frac{1}{L} \nabla F(\theta_t) \rangle + \frac{L}{2} \left\| \frac{1}{L} \nabla F(\theta_t) \right\|^2.$$

It allows us to bound the updates of F:

$$F(\theta_{t+1}) - F(\theta_t) \leq -\frac{1}{2L} \|\nabla F(\theta_t)\|^2. \tag{II.4}$$

One can note $\Delta_t = F(\theta_{t+1}) - F(\theta^\star)$. Then:

$$\Delta_{t+1} \leq \Delta_t - \frac{1}{2L} \|\nabla F(\theta_t)\|^2. \tag{II.5}$$

Convexity of F and triangular inequality gives us:

$$\Delta_t \leq \langle \nabla F(\theta_t); (\theta_t - \theta^\star) \rangle \leq \|\nabla F(\theta_t)\| \, \|(\theta_t - \theta^\star)\|,$$

which can be rewritten into:

$$\frac{\Delta_t^2}{\|(\theta_t - \theta^\star)\|^2} \leq \|\nabla F(\theta_t)\|^2. \tag{II.6}$$

II.5 and II.6 gives:

$$\Delta_{t+1} \leq \Delta_t - \frac{\Delta_t^2}{2L \|\theta_t - \theta^\star\|}.$$

$\|\theta_t - \theta^\star\|^2$ is a decreasing serie:

$$\begin{aligned}
\|\theta_{t+1} - \theta^\star\|^2 &= \left\|\theta_t - \theta^\star - \frac{1}{L}\nabla F(\theta_t)\right\|^2 \\
&= \|\theta_t - \theta^\star\|^2 - \frac{2}{L}\langle \theta_t - \theta^\star ; \nabla F(\theta_t)\rangle + \frac{1}{L^2}\|\nabla F(\theta_t)\|^2 && \text{(L-smoothness)} \\
&= \|\theta_t - \theta^\star\|^2 - \frac{2}{L}\langle \theta_t - \theta^\star ; \nabla F(\theta_t) - \nabla F(\theta^\star)\rangle + \frac{1}{L^2}\|\nabla F(\theta_t)\|^2 \\
&\le \|\theta_t - \theta^\star\|^2 - \frac{2}{L^2}\|\nabla F(\theta_t)\|^2 + \frac{2}{L^2}\|\nabla F(\theta_t)\|^2 && \text{(Co-coercivity)} \\
&\le \|\theta_t - \theta^\star\|^2 - \frac{2}{L^2}\|\nabla F(\theta_t)\|^2,
\end{aligned}$$

which permits to write:

$$\Delta_{t+1} \le \Delta_t - \frac{\Delta_t^2}{2L\|\theta_1 - \theta^\star\|}. \tag{II.7}$$

Multiplying II.7 by $\frac{-1}{\Delta_{t+1}\Delta_t}$ gives:

$$\frac{1}{2L\|\theta_1 - \theta^\star\|}\frac{\Delta_t}{\Delta_{t+1}} \le \frac{1}{\Delta_{t+1}} - \frac{1}{\Delta_t} \tag{II.8}$$

With $\frac{\Delta_t}{\Delta_{t+1}} \le 1$ from II.5, summing II.8 inequalities for $j \le t$ gives the final result:

$$F(\theta_t) - F(\theta^\star) \le 2L\frac{\|\theta_1 - \theta^\star\|^2}{t}.$$

□

The use of the convex setting is advantageous for proving convergence theorems because it lacks local minima, preventing the gradient descent algorithm from getting stuck. It should be noted that if the algorithm, as expressed in Equation II.1, reaches a local minimum where $\nabla F = \vec{0}$, it will remain trapped there indefinitely. By making stronger assumptions about F_θ, faster convergence can be demonstrated, though the details will not be discussed here.

However, gradient descent is not a universal solution for minimization problems. We will discuss its main limitations in the following section.

II.1.4. Limitations

Choosing the appropriate learning rate for gradient descent can be challenging due to several factors that can affect the optimization process. An ill-chosen learning rate can lead to slow convergence, oscillations, or even divergence of the algorithm. If the learning rate is too small, the algorithm will take very small steps, leading to slow convergence and possibly getting stuck in a local minimum. On the other hand, if the learning rate is too large, the algorithm may overshoot the optimal solution, causing oscillations or even divergence. It is important to note that the smooth parameter L from Equation II.2 is not always known.

Moreover gradient descent is not well-suited for discrete optimization problems due to its reliance on continuous gradients. Firstly, discrete optimization involves finding the best solution from a finite set of possible discrete choices, such as selecting the best combination of discrete variables or making discrete decisions. However, gradient descent is designed for continuous optimization, where the objective function and variables are continuous. Then gradient descent cannot handle the discrete nature of the variables, leading to invalid or nonsensical solutions. Secondly, gradient descent is a local search algorithm that aims to find the optimal solution by iteratively updating the parameters based on the local gradient information. However, in discrete optimization, the search space is often non-convex and contains multiple local optima. Gradient descent may get stuck in local optima and fail to explore the entire search space. Lastly, this class of problems often involve a combinatorial explosion of possible solutions. The search space grows exponentially with the number of discrete variables or the size of the problem. Gradient descent, which relies on calculating gradients and updating parameters iteratively, becomes computationally infeasible for large-scale discrete optimization problems.

To address discrete optimization problems, specialized optimization algorithms are used, such as integer programming [101], constraint programming [102] or genetic algorithms [103]. These methods are designed to handle discrete variables, non-differentiable functions, and explore the discrete search space more effectively than gradient descent.

Finally the main limitation of gradient descent is its computational cost in presence of a very large dataset. This motivates the following section.

II.2. Stochasticity

In the previous section we have presented the gradient descent algorithm. This is not very used in practice, SGD is often preferred.

II.2.1. Motivation

In real-world scenarios, the size of the dataset $\mathcal{Z}$ can be enormous. For instance, the CIFAR dataset used as an example consists of 60,000 32x32 color images, which corresponds to approximately 160 MB. This only concerns the data and does not even include the model. With respect to the Celio dataset discussed in Chapter I, the sales history comprises over 70 billion observations. Attempting to load full datasets and compute the exact gradient for a single gradient descent step is impractical due to the memory constraints [104]. This is known as the computational burden of the optimization problem. Furthermore, in complex models like neural networks, it is not always guaranteed that the optimization landscape is convex or smooth, leading to the possibility of local minima trapping the optimizer. As a result, an alternative approach is necessary to enable the learning of massive datasets. Rather than computing the exact gradient on the full dataset, one can use an estimator of the gradient, since the dataset itself can be viewed as observations of a wider phenomenon, dependent on the measurement span, for instance. Using an estimator $\hat{g}_t$ of the gradient might be an interesting solution [97]:

$$\theta_{t+1} = \theta_t - \alpha_t \hat{g}_t. \tag{II.9}$$

Let's consider the minimization problems presented in the first chapter. Our objective is to minimize the sum of an attribute constructed in the observation table of a PolyStar:

$$G_{SUM(loss)}(\Pi_{loss}(Observation)). \tag{II.10}$$

It means that the minimization problem can be written as

$$\begin{aligned}\theta^\star &= \arg\min_{\theta\in\mathbb{R}^p} \quad F_\theta \\ &= \arg\min_{\theta\in\mathbb{R}^p} \sum_{X,y\in\mathcal{Z}} f_\theta(X,y) \\ &= \arg\min_{\theta\in\mathbb{R}^p} \sum_{i=1\ldots n} f_\theta(X_i,y_i).\end{aligned}$$

Then the estimator of the gradient can be induced by decomposition of the dataset into batches of observations. Each batch of observation is small in terms of data size, which makes it possible to estimate it on very large datasets.

SGD is a direct extension of gradient descent that also provides convergence guarantees. We will discuss some of these guarantees in the following section.

II.2.2. Convergence of SGD

SGD operates based on an estimation of the objective function's gradient, rather than the exact gradient itself. It may be unclear how this optimization technique can effectively minimize a function that it does not have an exact access to. However, several groundbreaking works have demonstrated convergence guarantees with this gradient estimation, including [97, 98, 104–106].

Our objective is to prove the convergence of SGD following the update rule defined in equation II.9. We assume that we have a random variable χ that draws samples from $\mathcal{Z}$, denoted as χ_t. We then define the gradient estimator $\hat{g}_t$ using these samples:

$$\hat{g}_t = \nabla_{\theta_{t-1}} f_{\theta_{t-1}}(\chi_t). \tag{II.11}$$

We note

$$\mathcal{F}(\theta) = \mathbb{E}_\chi[f_\theta(\chi)]. \tag{II.12}$$

Theorem 3 (SGD convergence)**.** *Assume that the gradient of F exists and is bounded:*

$$\exists B > 0, \forall \theta \in \mathbb{R}^p : \sup_{X,y\in\mathcal{Z}} \|\nabla_\theta f_\theta(X,y)\|^2 \leq B. \tag{II.13}$$

Assume also that there exists $\mu > 0$ such that $\mathcal{F}$ is μ-strictly convex:

$$\forall \theta, \theta' \in \mathbb{R}^p \quad \mathcal{F}(\theta') \geq \mathcal{F}(\theta) + \langle \nabla\mathcal{F}(\theta); \theta' - \theta\rangle + \frac{\mu}{2}\|\theta - \theta'\|^2. \tag{II.14}$$

Then $\theta^\star = \arg\min_{\theta\in\mathbb{R}^p} \quad \mathcal{F}(\theta)$ exists and is unique and for any $\epsilon > 0$, there exists $\alpha \in \mathbb{R}^{+\star}$ such that

$$\limsup_{t\to\infty} \quad \mathbb{E}[\|\theta_t - \theta^\star\|^2] \leq \epsilon.$$

We have chosen proof from [107] for its simplicity.

Proof. First of all, the uniqueness of $\theta^\star$ is guaranteed by the μ-strictly convexity of $\mathcal{F}$.

For notation convenience, let's note

$$\delta_t = \mathbb{E}[\|\theta_t - \theta^\star\|^2]$$

Then we start by bounding δ_{t+1} in function of δ_t:

$$\begin{aligned}
\delta_{t+1} &= \mathbb{E}[\|\theta_t - \alpha_t \hat{g}_t - \theta^\star\|^2] \\
&= \mathbb{E}[\|\theta_t - \theta^\star\|^2] + \alpha^2 \mathbb{E}[\|\hat{g}_t\|^2] - 2\alpha \mathbb{E}[\langle \theta_t - \theta^\star ; \hat{g}_t \rangle] \\
&= \delta_t + \alpha^2 \mathbb{E}[\|\hat{g}_t\|^2] - 2\alpha \mathbb{E}[\langle \theta_t - \theta^\star ; \nabla \mathcal{F}(\theta_t) \rangle] && \text{(expectancy on } \chi) \\
&\leq \delta_t + \alpha^2 B - 2\alpha \mathbb{E}[\mathcal{F}(\theta_t) - \mathcal{F}(\theta_\star) + \frac{\mu}{2} \|\theta_t - \theta_\star\|^2] && (\mu - \text{strongly convex}) \\
&\leq \delta_t + \alpha^2 B - \alpha\mu \mathbb{E}[\|\theta_t - \theta_\star\|^2] && (\theta_\star \text{ is the minimum of } F) \\
&= (1 - \alpha\mu)\delta_t + \alpha^2 B.
\end{aligned}$$

This inequality can be rewritten as:

$$(\delta_{t+1} - \frac{\alpha B}{\mu}) \leq (1 - \alpha\mu)(\delta_t - \frac{\alpha B}{\mu}).$$

Taking the positive part of $\tilde{\delta_t} = \delta_t - \frac{\alpha B}{\mu}$ and choosing α such that $\alpha \leq \frac{1}{\mu}$:

$$(\tilde{\delta_{t+1}})_+ \leq (1 - \alpha\mu)(\tilde{\delta_t})_+.$$

We can iterate this inequality for $k > 1$:

$$(\tilde{\delta_{t+k}})_+ \leq (1 - \alpha\mu)^k (\tilde{\delta_t})_+.$$

That means that:

$$\limsup_{k \to \infty} \quad (\tilde{\delta_k})_+ = 0.$$

Then for any α smaller than $\frac{1}{\mu}$ and smaller than $\frac{\epsilon\mu}{B}$ we obtain

$$\limsup_{t \to \infty} \quad \delta_t \leq \epsilon,$$

which concludes the proof. □

When the dataset is made of a finite set of observations $X, y \in \mathcal{Z}$, a natural χ is a random uniform draw on these observations. Then $\mathcal{F}(\theta)$ reduces to the quantity we want to minimize.

Equation II.9 provides a method for updating the parameters based on the gradient estimation. However, there are several other approaches, known as optimizers, that can be used for this update. In the following section, we will introduce the most widely used optimizers and discuss the convergence guarantees they offer.

II.3. Convergence properties with adaptive optimizers

II.3.1. Optimizers

An optimizer is essentially an algorithm that iteratively adjusts the model's parameters to minimize the objective function. It uses the gradient of the objective function with respect to the model's parameters to determine the direction of the parameters adjustment, and the learning rate to determine the magnitude of the adjustment. The learning rate controls the step size of the optimizer, and a smaller learning rate results in slower but more accurate convergence, while a larger learning rate can result in faster convergence but may risk overshooting the optimal solution.

Different optimizers use different strategies for updating the parameters, such as momentum-based methods, adaptive learning rate methods, and stochastic gradient methods. The optimizer choice can have a significant impact on the performance of the model, as different optimizers may converge at different rates and to different local optima.

In the following, we review common optimizers and discuss the circumstances in which they perform optimally. This review is not exhaustive and the chosen optimizers are presented in chronological order of their historical development.

Vanilla SGD

Vanilla SGD is the first optimizer and is presented in Equation II.1. The advantage of the vanilla optimizer for gradient descent is that it is computationally simple and easy to implement, making it a popular choice for optimizing machine learning models. Additionally, it works well for many problems, especially those with relatively smooth loss surfaces. However, there are some disadvantages to the vanilla optimizer. One of the main issues is that it can converge slowly, especially in cases where the loss surface is not smooth or has sharp, narrow valleys. In such cases, the optimizer may oscillate around the minimum or get stuck in local minima. Additionally, the learning rate must be chosen carefully to balance convergence speed and stability, which can require some trials and errors. Finally, the vanilla optimizer for gradient descent does not include any adaptive techniques to adjust the learning rate during training, which can limit its effectiveness on some problems.

Polyak's momentum

Polyak's momentum is a popular modification of the stochastic gradient descent (SGD) algorithm that was introduced by [108]. The basic idea behind momentum is to add a fraction of the previous gradient estimate to the current one, which helps smooth out the direction of updates and accelerate convergence.:

$$\theta_{t+1} = \theta_t - \alpha g_t + \beta(\theta_t - \theta_{t-1}).$$

This allows the optimizer to continue moving in the same direction even when the current gradient changes direction, thereby reducing oscillations and improving stability. Polyak's momentum has become a widely used optimization algorithm in deep learning due to its ability to speed up convergence and improve the robustness of the optimization process.

Adagrad

Adagrad [109] is an optimization algorithm for gradient-based optimization that adapts the learning rate for each parameter based on the gradients' histories. The key idea behind Adagrad is to scale the learning rate for each parameter based on its historical gradient variance. This means that parameters that have large gradients will get a smaller learning rate and parameters that have small gradients will get a larger learning rate, thus allowing the optimizer to converge quickly while avoiding the problem of overshooting the minimum.

The algorithm maintains a per-parameter learning rate, which is updated based on the sum of the squares of the past gradients for that parameter. The update rule for Adagrad is as follows:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\epsilon + \sum\limits_{k<t} g_k^2}} g_t.$$

Adagrad's default hyperparameters are $\alpha = 0.001$ and $\epsilon = 10^{-8}$. One advantage of Adagrad is that it requires minimal hyperparameter tuning, as the learning rate is adaptively scaled based on the historical gradients. However, one limitation of Adagrad is that the pseudo learning rate $\frac{\alpha}{\sqrt{\epsilon + \sum\limits_{k<t} g_k^2}}$ continues to decrease as the sum of squares of the past gradients grows larger, which can cause the pseudo learning rate to become too small, making it difficult for the optimizer to escape from local minima.

Adam

Adam was introduced by [110]. One of the main contributions of this optimizer is the introduction of refined momentum m_t and r_t estimation of the gradient. It relies on the momentum by using a sort of moving average of the gradient instead of the computed gradient, similar to the Polyak's momentum.

$$\begin{aligned}
m_{t+1} &= \beta_1 m_t + (1 - \beta_1) g_t \\
r_{t+1} &= \beta_2 r_t + (1 - \beta_2) g_t^2 \\
\hat{m}_{t+1} &= \frac{m_{t+1}}{1 - \beta_1^{t+1}} \\
\hat{r}_{t+1} &= \frac{r_{t+1}}{1 - \beta_2^{t+1}} \\
\theta_{t+1} &= \theta_t - \alpha \frac{\hat{m_{t+1}}}{\sqrt{\hat{r_{t+1}} + \epsilon}}.
\end{aligned}$$

Adam's default hyperparameters are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

One of the interesting properties of Adam is that its stepsizes are approximately bounded by the learning rate. Indeed, in the default case with $(1 - \beta_1) = \sqrt{1 - \beta_2}$, we get $\frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}}} < \frac{\hat{m}_{t+1}}{\sqrt{\epsilon + \hat{v}_{t+1}}} < 1$.

For big gradients, as soon as $\beta_1 < \beta_2$, the stepsize is smaller than the learning rate, as explained in II.15:

$$\lim_{g_t \to \infty} \frac{|\hat{m}_{t+1}|}{\sqrt{\epsilon + \hat{v}_{t+1}}} = \frac{1 - \beta_1}{1 - \beta_1^{t+1}} |g_t| \times \sqrt{\frac{1 - \beta_2^{t+1}}{(1 - \beta_2) g_t^2}} = \frac{1 - \beta_1}{\sqrt{1 - \beta_2}} \frac{1 - \beta_1^{t+1}}{\sqrt{1 - \beta_2^{t+1}}} < 1. \quad \text{(II.15)}$$

This property makes Adam powerful regardless of the amplitude of the gradient. One advantage of Adam over Adagrad is that Adam uses both momentum and adaptive learning rates. The momentum helps the optimizer to continue moving in the same direction even when the gradients have become small or noisy. The adaptive learning

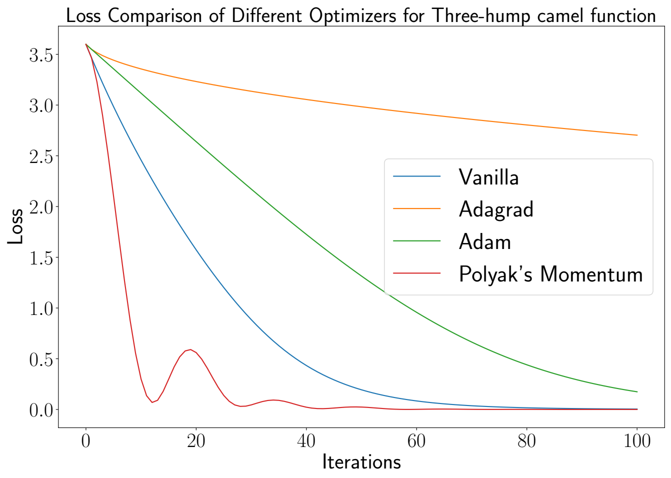


Figure II.2.: The Three-hump camel-back function was minimized using 4 optimizers, each demonstrating different rates of objective function reduction. Default parameters were employed for each optimizer, although this did not provide an advantage for the Adagrad and Adam optimizer due to their small learning rates of 0.01.

rates help the optimizer to automatically adjust the learning rate for each parameter based on the historical gradients. This combination of momentum and adaptive learning rates makes Adam more efficient in finding the minimum of the loss function compared to Adagrad. Additionally, Adam is less sensitive to hyperparameters, such as the learning rate, compared to Adagrad.

Due to these numerous advantages and after extensive benchmarking, Lokad has made the decision to adopt Adam as the default optimizer for all production optimization tasks. From an industrial standpoint, it is logical to limit the number of choices available to the domain expert, who is already in charge of the model design. Note that in production, apart from the learning rate, no hyperparameters can be modified by the expert.

Difference between optimizers

To highlight the difference between these optimizers, we have tried to minimize the Three-hump camel-back t_{hc} function:

$$t_{hc}(x,y) = 2x^2 - 1.05x^4 + \frac{x^6}{6} + xy + y^2.$$

The global minimum of this function can easily be proven as $t_{hc}(0,0)$. This function is also strictly convex as $\frac{\partial^2 t_{hc}}{\partial x^2} > 0 < \frac{\partial^2 t_{hc}}{\partial y^2}$. Consequently its minimization through gradient descent is easy and all the optimizers perform this task well. However we note different behavior in Figure II.2. In the case this strictly convex function, momentum-based optimizers such as Adagrad and Adam lose their primary advantage. There are no exotic gradient points that need to be mitigated by the gradient descent general trend.

II.3.2. Convergence generalization

In the following we formalize the two possible stochasticities, from the observation or from the function, into a generic setting. This setting is directly inspired from [98].

Our goal is still to minimize the function F. We assume there exists a random function $f : \mathbb{R}^p \to \mathbb{R}$ such that its gradient estimates without any bias the true gradient of F:

$$\forall \quad \theta \in \mathbb{R}^p; \quad \mathbb{E}[\nabla f(\theta)] = \nabla F(\theta),$$

we also assume that we have access to an oracle providing i.i.d. samples $(f_t)_{t\in\mathbb{N}^\star}$. We denote $\mathbb{E}_{t-1}[.]$ the conditional expectation knowing $f_1, \ldots, f_{t-1}$. The origin of these samples is inconsequential, whether they come from observation-related stochasticity, function-related stochasticity, or any other source as proposed in future Section II.4.1.

In this setting we do not consider the full gradient of F as an input but we only access a series of realizations of an unbiased estimator of it. As we did in the smooth and convex case, we need to make three assumptions on the function F we aim to minimize and on the samples.

First, F is bounded below by B:

$$\forall \theta \in \mathbb{R}^p; \quad F(\theta) \geq B, \tag{II.16}$$

this assumption makes sure that F can be minimized.

Second, the l_∞-norm of the gradient estimator is uniformly almost surely bounded, i.e there exists $M \geq 0$ such that:

$$\forall \theta \in \mathbb{R}^p; \quad \|\nabla f(\theta)\|_\infty \leq M. \tag{II.17}$$

Third, the true gradient is L-Liptchitz-continuous with respect to the l_2-norm:

$$\forall \quad \theta, \theta'; \quad \|\nabla F(\theta) - \nabla F(\theta')\|_2 \leq L \|\theta - \theta'\|_2 . \tag{II.18}$$

Under these three assumptions, we can obtain convergence properties for stochastic gradient descent. To achieve this, we do not utilize the basic parameter update from equation II.1, but instead employ the optimizer from Equation II.19 introduced by [98], which is a generalization of adaptive optimizers into a single one.

$$\begin{aligned} m_{t+1} &= \beta_1 m_t + g_t \\ r_{t+1} &= \beta_2 r_t + g_t^2 \\ \theta_{t+1} &= \theta_t - \alpha_t \frac{m_{t+1}}{\sqrt{r_{t+1} + \epsilon}}. \end{aligned} \tag{II.19}$$

Although the following holds for different values of β, we use this optimizer with $\beta_1 = 0$ and $\beta_2 = 1$ henceforth for the simplification of the proof. With these parameters, the optimizer reduces to Adagrad. If we choose the default value of Adagrad, this version is very close from the original one, especially after a few iterations.

The following proves the convergence of stochastic gradient descent in specific cases. Note that an uncareful hyperparameters adjustments will lead to non convergence even with this optimizer [111].

Theorem 4 (Stochastic gradient descent converges). *For any $T \in \mathbb{N}^\star$, with τ a random index that uniformly draws into $[\![0..T[\![$ and $R = M + \sqrt{\epsilon}$, given the previous assumptions:*

$$\mathbb{E}[\|\nabla F(\theta_\tau)\|^2] \leq 2R\frac{F(\theta_0) - F^\star}{\alpha T} + \frac{1}{\sqrt{T}}(4pR^2 + \alpha pRL)\ln(1 + \frac{TR^2}{\epsilon}).$$

Notation To make the following proofs more readable, we note $G = \nabla F(\theta_{t-1})$; $g = \nabla f_t(\theta_{t-1})$, $v = v_t$, $\tilde{v} = \tilde{v}_t$, $v_\epsilon = v + \epsilon$ and $\tilde{v}_\epsilon = \tilde{v} + \epsilon$. By definition:

$$\mathbb{E}_{t-1}[g^2] \leq \tilde{v}_\epsilon, \tag{II.20}$$

and

$$g^2 \leq v_\epsilon. \tag{II.21}$$

Lemma 1 (adaptive updates approximately follow a descent direction). *For all $t \in \mathbb{N}^\star$*

$$\mathbb{E}_{t-1}[\frac{Gg}{\sqrt{v_\epsilon}}] \geq \frac{G^2}{\sqrt{\tilde{v}_\epsilon}} - 2R\mathbb{E}_{t-1}[\frac{g^2}{v_\epsilon}]. \tag{II.22}$$

Proof. By linearity

$$\mathbb{E}_{t-1}[\frac{Gg}{\sqrt{v_\epsilon}}] = \mathbb{E}_{t-1}[\frac{Gg}{\sqrt{\tilde{v}_\epsilon}}] + \mathbb{E}_{t-1}[Gg(\frac{1}{\sqrt{v_\epsilon}} - \frac{1}{\sqrt{\tilde{v}_\epsilon}})]. \tag{II.23}$$

Knowing $(f_i)_{i\leq t-1}$ does not give any information on G and g so the first term gives:

$$\mathbb{E}_{t-1}[\frac{Gg}{\sqrt{\tilde{v}_\epsilon}}] = \frac{G^2}{\sqrt{\tilde{v}_\epsilon}}.$$

For the second term:

$$\begin{aligned}
Gg(\frac{1}{\sqrt{v_\epsilon}} - \frac{1}{\sqrt{\tilde{v}_\epsilon}}) &= Gg\frac{\sqrt{\tilde{v}_\epsilon} - \sqrt{v_\epsilon}}{\sqrt{\tilde{v}_\epsilon}\sqrt{v_\epsilon}} \\
&= Gg\frac{\tilde{v} - v}{\sqrt{\tilde{v}_\epsilon}\sqrt{v_\epsilon}(\sqrt{\tilde{v}_\epsilon} + \sqrt{v_\epsilon})} \\
&= Gg\frac{\mathbb{E}_{t-1}[g^2] - g^2}{\sqrt{\tilde{v}_\epsilon}\sqrt{v_\epsilon}(\sqrt{\tilde{v}_\epsilon} + \sqrt{v_\epsilon})} \\
| Gg(\frac{1}{\sqrt{v_\epsilon}} - \frac{1}{\sqrt{\tilde{v}_\epsilon}}) | \leq | Gg | \frac{\mathbb{E}_{t-1}[g^2]}{\sqrt{v_\epsilon}(\tilde{v}_\epsilon)} &+ | Gg | \frac{g^2}{\sqrt{\tilde{v}_\epsilon}(v_\epsilon)} = A_1 + A_2.
\end{aligned}$$

For all $a, b \in \mathbb{R}$, $h_{a,b}(x) = \frac{a^2}{2}x^2 - abx + \frac{b^2}{2}$ is positive ($\Delta = 0$) on $\mathbb{R}$. Thus for all $x > 0$:

$$ab \leq \frac{a^2}{2}x + \frac{b^2}{2x}, \tag{II.24}$$

so let's apply Inequality II.24 on A_1 with

$$x = \frac{\sqrt{\tilde{v}_\epsilon}}{2}; a = \frac{| G |}{\sqrt{\tilde{v}_\epsilon}}; b = \frac{| g | \mathbb{E}_{t-1}[g^2]}{\sqrt{v_\epsilon \tilde{v}_\epsilon}}.$$

we get:

$$A_1 \leq \frac{G^2}{4\sqrt{\tilde{v}_\epsilon}} + \frac{g^2\mathbb{E}_{t-1}[g^2]^2}{v_\epsilon \tilde{v}_\epsilon^{3/2}}.$$

Thanks to II.20:

$$\begin{aligned}\mathbb{E}_{t-1}[A_1] &\leq \frac{G^2}{4\sqrt{\tilde{v}_\epsilon}} + \frac{\mathbb{E}_{t-1}[g^2]}{\sqrt{\tilde{v}_\epsilon}}\mathbb{E}_{t-1}[\frac{g^2}{v_\epsilon}] \\ &\leq \frac{G^2}{4\sqrt{\tilde{v}_\epsilon}} + R\mathbb{E}_{t-1}[\frac{g^2}{v_\epsilon}].\end{aligned}$$

Now lets apply Inequality II.24 on A_2 with

$$x = \frac{\sqrt{\tilde{v}_\epsilon}}{2\mathbb{E}_{t-1}[g^2]}; a = \frac{\mid Gg \mid}{\sqrt{\tilde{v}_\epsilon}}; b = \frac{g^2}{v_\epsilon},$$

we get:

$$A_2 \leq \frac{G^2}{4\sqrt{\tilde{v}_\epsilon}}\frac{g^2}{\mathbb{E}_{t-1}[g^2]} + \frac{\mathbb{E}_{t-1}[g^2]}{\sqrt{\tilde{v}_\epsilon}}\frac{g^4}{v_\epsilon^2}.$$

Thanks to II.20:

$$\begin{aligned}\mathbb{E}_{t-1}[A_2] &\leq \frac{G^2}{4\sqrt{\tilde{v}_\epsilon}} + \frac{\mathbb{E}_{t-1}[g^2]}{\sqrt{\tilde{v}_\epsilon}}\mathbb{E}_{t-1}[\frac{g^2}{v_\epsilon}] \\ &\leq \frac{G^2}{4\sqrt{\tilde{v}_\epsilon}} + R\mathbb{E}_{t-1}[\frac{g^2}{v_\epsilon}].\end{aligned}$$

Summing the inequalities gives:

$$\mathbb{E}_{t-1}[\mid Gg(\frac{1}{\sqrt{v_\epsilon}} - \frac{1}{\sqrt{\tilde{v}_\epsilon}}) \mid] \leq \mathbb{E}_{t-1}[\mid A_1 \mid + \mid A_1 \mid] \leq \frac{G^2}{2\sqrt{\tilde{v}_\epsilon}} + 2R\mathbb{E}_{t-1}[\frac{g^2}{v_\epsilon}], \tag{II.25}$$

II.23 and II.25 can thus complete the lemma proof:

$$\frac{G^2}{2\sqrt{v_\epsilon}} \leq \mathbb{E}_{t-1}[\frac{Gg}{\sqrt{v_\epsilon}}] + 2R\mathbb{E}_{t-1}[\frac{g^2}{v_\epsilon}].$$

□

Proof. (of Theorem 2)

Lets start with the smoothness of F between θ_{t+1} and θ_t:

$$F(\theta_t) \leq F(\theta_{t-1}) - \alpha\langle G.\frac{g}{\sqrt{v_\epsilon}}\rangle + \alpha^2\frac{L}{2}\left\|\frac{g}{\sqrt{v_\epsilon}}\right\|_2^2, \tag{II.26}$$

with the conditional expectation with respect to the t previous samples:

$$\mathbb{E}_{t-1}[F(\theta_t] \leq F(\theta_{t-1}) - \alpha\mathbb{E}_{t-1}[\langle G.\frac{g}{\sqrt{v_\epsilon}}\rangle] + \alpha^2\frac{L}{2}\mathbb{E}_{t-1}[\left\|\frac{g}{\sqrt{v_\epsilon}}\right\|_2^2]. \tag{II.27}$$

We have that $\sqrt{v_\epsilon} \leq R\sqrt{t}$ which gives:

$$\mathbb{E}_{t-1}[F(\theta_t] \leq F(\theta_{t-1}) - \frac{\alpha}{2R\sqrt{t}} \|G\|_2^2 + (2\alpha R + \frac{\alpha^2 L}{2})\mathbb{E}_{t-1}[\left\| \frac{g}{\sqrt{v_\epsilon}} \right\|_2^2], \tag{II.28}$$

this inequality holds for any $0 \leq t \leq T$, thus we can sum them and obtain:

$$\mathbb{E}[F(\theta_T]) \leq F(\theta_0) - \frac{\alpha}{2R\sqrt{T}} \sum_{t=0}^{T-1} \mathbb{E}[\|\nabla F(\theta_t)_2^2\|] + (2\alpha R + \frac{\alpha^2 L}{2}) \sum_{t=0}^{T-1} \mathbb{E}[\left\| \frac{g}{\sqrt{v_\epsilon}} \right\|_2^2]. \tag{II.29}$$

As $\ln(x) \leq x$ for $x \in \mathbb{R}^{+\star}$, for $a, b \in \mathbb{R}^{+\star}$, we have:

$$\frac{a}{b} \leq \ln(b) - \ln(a), \tag{II.30}$$

we can apply this inequality with $a = \nabla f_t(\theta_{t-1})^2$ and $b = \epsilon + v_t$:

$$\begin{aligned} \frac{\nabla f_t(\theta_{t-1})^2}{\epsilon + v_t} &\leq \ln(\epsilon + v_t) - \ln(\epsilon + v_t - \nabla f_t(\theta_{t-1})^2) \\ &\leq \ln(\epsilon + v_t) - \ln(\epsilon + v_{t-1}), \end{aligned}$$

summing it forms a telescoping series:

$$\sum_{t=0}^{T} \frac{\nabla f_t(\theta_{t-1})^2}{\epsilon + v_t} \leq \ln(1 + \frac{v_T}{\epsilon}). \tag{II.31}$$

Using it in Equation II.29 gives:

$$\mathbb{E}[F(\theta_T]) \leq F(\theta_0) - \frac{\alpha}{2R\sqrt{T}} \sum_{t=0}^{T-1} \mathbb{E}[\|\nabla F(\theta_t)_2^2\|] + (2\alpha R + \frac{\alpha^2 L}{2}) p \ln(1 + \frac{TR^2}{\epsilon}). \tag{II.32}$$

Rearranging the terms and bounding below $F(\theta)$ by $F^\star$ thanks to the first assumption concludes the proof. □

It should be noted that the theoretical application of these concepts is based on functions that satisfy the three assumptions. In practice, however, the models employed may not always fulfill these assumptions, as illustrated by a concrete example in Section III.3.3. In some cases, the model itself might not even be differentiable. For instance, rectified linear units (ReLU) [112] are frequently applied to neural network layers and are defined as $ReLu(x) = \max(x, 0)$. Fortunately, these conditions are sufficient for convergence but not strictly necessary.

The validity of Theorem 4 relies on the assumption of the existence of a random function that provides an unbiased estimation of the true gradient. Building upon the theoretical guarantees established with this framework, we will now introduce two practical techniques for obtaining such a function in practice.

II.4. Stochasticity generalization and applications

II.4.1. Alternative stochasticity origin

Estimating the gradient of the mathematical function using the entire dataset requires access to the full set of data $\mathcal{Z}$ and the corresponding mathematical function gradient ∇f. To introduce stochasticity in the gradient estimation process and reduce computational resources needed for parameter updates, a common approach is to divide the dataset into smaller batches. Alternatively, a lighter version of the mathematical function with reduced computational cost can also achieve the same objective.

In the following sections, we will discuss these two techniques for estimating the gradient and their suitability for use with the selected optimizer.

Observation

Given that the objective function is expressed as a sum over an observation table in Equation II.10, a natural way to estimate the gradient would be to compute it on a randomly selected subset S of fixed size from the observation table.

$$\hat{g}_S = \frac{|Z|}{|S|}\sum_{z\in S}\nabla_\theta f(z),$$

the size of S is called the batch size.

Thanks to the linearity of the gradient, such estimator is unbiased:

$$\begin{aligned}
\mathbb{E}[\hat{g}_S] &= \frac{|\mathcal{Z}|}{|S|}\mathbb{E}[\sum_{z\in S}\nabla_\theta f(z)] \\
&= \frac{|\mathcal{Z}|}{|S|}\sum_{z\in S}\mathbb{E}[\nabla_\theta f(z)] \\
&= \frac{|\mathcal{Z}|}{|S|}\frac{|S|}{|\mathcal{Z}|}\mathbb{E}[\nabla_\theta F] \\
&= \mathbb{E}[\nabla_\theta F].
\end{aligned}$$

Reducing the number of observations utilized in gradient computation is the most prevalent technique for decreasing the computational cost. This approach also enables handling streaming data, where the full set of observations is not available simultaneously. This method relies on dividing the observations into batches. Additionally, one can consider modifying the function $\nabla_\theta f$ to enable faster computations.

Function

One can define the linearized computational graph of f_θ, in which the nodes represent intermediate variables and the edges represent the mathematical operations. It is a specific representation of a computation graph that orders the nodes in such a way that each node's input operands appear earlier in the sequence than the node itself. This linearization ensures that the computation graph can be traversed and executed sequentially, simplifying the process of evaluating the function and its derivatives.

Definition 9 (LCG). *A linearized computational graph (LCG) is a directed acyclic graph $G = (V, E)$, where V is the set of nodes representing variables, operations, and functions, and E is the set of directed edges representing the dependencies between the nodes. The linearization of G is a sequence of nodes $L = (z_1, z_2, ..., z_n)$, where n is the number of nodes in V, and for each edge $(z_i, z_j) \in E$, $i < j$. This linear ordering ensures that all dependencies for a node are satisfied before the node itself is processed.*

In a LCG, each operation depends only on the output of the previous operation. This simplification allows for efficient calculations, as the operations can be computed one after the other, without the need to store all the intermediate values.

Our objective is to compute the derivative of the output node with respect to the input parameters; this can be written as a sum over all paths in LCG. It has been explained by [113] as the decomposition of the gradient on the contribution of the LCG paths from the parameter θ to the output node z. This is depicted in Figure II.3 and Equation II.33 with f_θ the function to minimize with respect to θ.

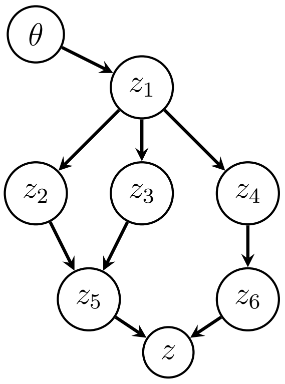
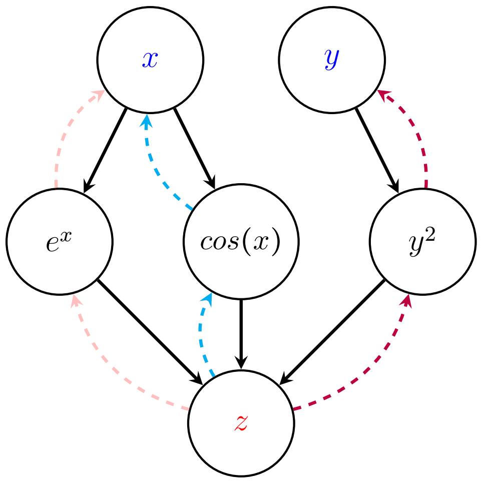


Figure II.3.: Example of the LCG of an objective function f_θ from the parameter θ to the output node z.

$$\nabla_\theta f = \sum_{\theta \longrightarrow z} \prod_{z_k \longrightarrow z_l} \frac{\partial z_l}{\partial z_k}. \tag{II.33}$$

$z_k \longrightarrow z_l$ represents a directed edge connecting two nodes, and z is the output node that represents f_θ. The total gradient is the sum of all the path contributions.

Thanks to this formula, any gradient can be written as a sum with this decomposition. One can draw random terms in this sum, which is equivalent to drawing random paths in the LCG. With the uniform distribution on the sum terms, this estimator is unbiased. We give an illustration of this technique on the function f_2 in Figure II.4.



$$\frac{\partial f_2}{\partial x} = \frac{\partial e^x}{\partial x} y^2 \cos x + \frac{\partial \cos x}{\partial x} e^x y^2$$
$$\frac{\partial f_2}{\partial y} = \frac{\partial y^2}{\partial y} e^x \cos x.$$

Figure II.4.: LCG of $f_2(x,y) = y^2 e^x \cos x$. The dashed lines represent backpropagation.

Remark 5. *Although LCGs and PolyStars share the characteristic of being graphs, it is important to note that they operate on different entities. LCGs are concerned with mathematical functions and their operations, whereas PolyStars focus on tables that store data and capture relationships between them. It is crucial to avoid confusion between these two concepts.*

With this approach, the stochasticity on the gradient does not come from the data which have been split into batches. The stochasticity comes from the gradient code itself and also reduces the needed resource as all the paths do not need to be computed anymore. This approach has been introduced by [114] and is generalized in Chapter IV.

Following the introduction of the two methods for obtaining a gradient estimator, we will apply them to the forecasting model presented in Equation I.2 in the first chapter. This will allow us to investigate their applicability in this specific case.

II.4.2. Gradient stochasticity applied on relational data and PolyStar

Firstly, we develop how the stochasticity obtained from the decomposition of the observations into batches applies on provided examples. Upon introducing the PolyStar in Definition 6, we highlighted a special table referred to as the observation table. All other tables involved in the relation query are determined by their relationship with the observation table. By doing this, we can apply SGD by partitioning the dataset using the rows of the observation table.

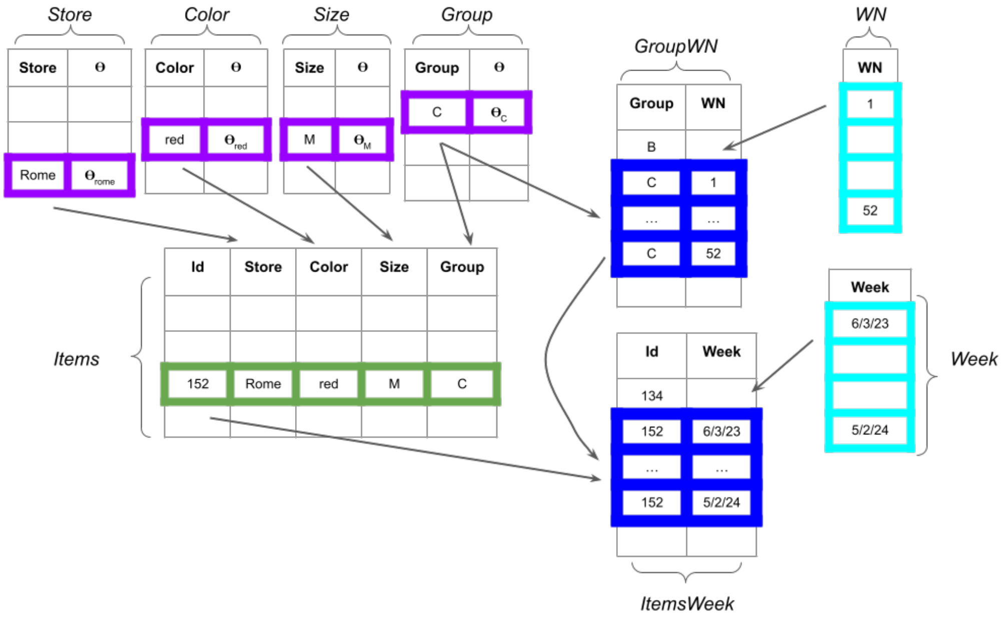


Figure II.5.: Data loading from the Celio dataset with model from Equation I.2. For each line of the Items table, one can access the corresponding line of the upstreams tables (Store, Color, Size, Group). The Full tables WN and Week are fully loaded, while the upstream-cross table GroupWN and observation-cross ItemsWeek are partly loaded.

The full data pertaining to a row of the observation table is explicitly defined through the PolyStar. To demonstrate this, we provide an example derived from the Celio model presented in Equation I.2, utilizing the same tables as the PolyStar defined in Figure I.23. In this example, *Store*, *Color*, *Size* and *Group* are the primary keys of the tables with the same name, while they are foreign keys of the Items table. Such relationships allow us to define a TOTAL JOIN between these tables and load the minimum required data to compute the loss and the gradient at each observation from the Items table.

A suitable framework implementation enables us to load only the minimal required data for each observation, as represented by the selected lines of Figure II.5. By doing so, the execution of SGD is accelerated, as data handling consumes a non-negligible portion of computing resources, as detailed in Section I.1.3. This perspective emphasizes the necessity of introducing a differential layer in relational programming languages, as accomplished in Chapter I.

Secondly, regarding the function stochasticity of the Celio model, the formal representation of the model under a LCG is given in Figure II.6. One remarks that there is one and only one path from each parameter to the final loss. Consequently the presented gradient decomposition as a sum does not apply in that case.

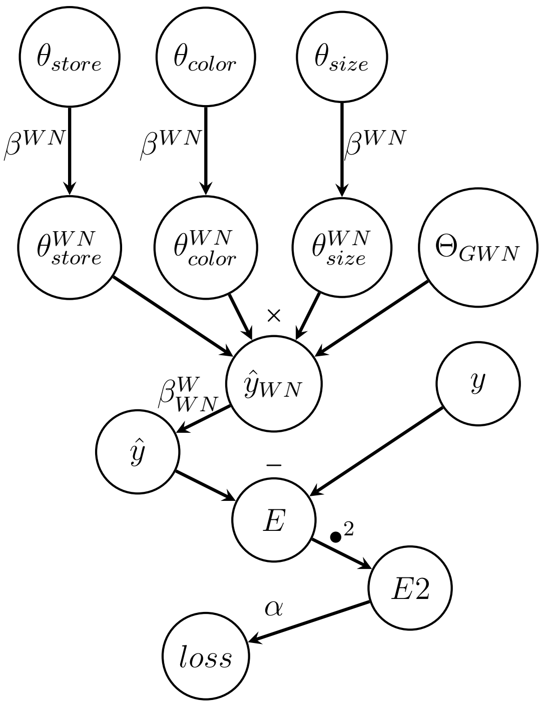


Figure II.6.: LCG of the Celio model presented in Equation I.2. It is an alternative representation of the Adsl statements of Figure I.4. The parameters nodes are θ_{store}, θ_{color}, θ_{size} and Θ_{GWN} whereas $loss$ is the output node.

Conclusion

In conclusion, this chapter aimed to provide a comprehensive understanding of SGD and its practical applications in training machine learning models. We have discussed the algorithm's efficiency and scalability, as well as its relationship with traditional gradient descent. We digged into the stochasticity aspect of this technique and proposed two different ways to obtain an estimator of the true gradient, one based on the observation the other on the function itself.

The chapter also presented a unified view of adaptive optimizers and their applications to SGD, offering convergence guarantees for non-convex functions based on the unbiasedness of the estimator.

We have explored the mathematical framework supporting gradient descent, its efficiency in the convex case, and convergence guarantees under regularity assumptions. Additionally, we delved into various approaches to introducing stochasticity in gradient descent and presented some of the best optimizers for this context, along with unifying them to provide convergence guarantees under other regularity conditions.

This chapter serves as a solid theoretical foundation for the subsequent chapters, which introduce novel gradient estimators and further explore the practical applications of SGD. This chapter will remain crucial for understanding and implementing efficient optimization algorithms in a variety of contexts, as in the following chapters.

III. Gradient estimator for categorical features

The work presented in this chapter is based on [115].

Introduction

Tabular data represent a considerable amount of modern data especially in the healthcare and industrial sectors. Machine Learning has been applied to those tabular data for decades for different tasks such as regression or classification. Boosting methods [116,117] are widely spread on these data and are still state of the art. After outstanding results on image, speech recognition or text, some attempts to apply deep learning models on tabular data have been published recently but with a limited impact on the state of the art as presented by [7]. Some deep learning approaches [118–120] have tried to adapt their architecture to the specificity of tabular data but none did succeed in overtaking classical machine learning models such as gradient-boosted tree ensembles [121].

One of the possible explanations is that deep learning excels on homogeneous data where embeddings can be learned [122, 123] via SGD presented in Chapter II to update their parameters by utilizing the underlying nature of the data like 2D spatial pixel neighborhood. In contrast, there is no such general underlying structure on tabular data. Domain transfer is not applicable in tabular data problems due to the absence of a common underlying structure. Unlike image or language processing, where knowledge and data from one problem can be leveraged to learn another problem with fewer annotated data, tabular data lacks this transferability. Each tabular problem possesses its unique structure and data characteristics, rendering the generalization of learnings across different problems impractical.

As tabular data often contains categorical data which are not numerical, the inputs of the model consists in the encoding of data. When the categorical data cardinality is limited, one-hot encoding is a good solution. Beyond its simplicity, it creates category-related parameters that are key for model explainability. In this encoding, only one feature is set to 1 (indicating the presence of the corresponding category), while all other features are set to 0. During the model optimization through gradient descent, the gradient is only present for the active feature, and all other features have zero gradient. The issue with this is that a zero gradient can halt the optimization process, as the optimizer updates all the parameters, including the ones corresponding to the inactive features. Considering the booleans 0 and 1 as numerical data by the optimization algorithm can be seen as improper usage, leading to potential negative consequences. This misuse deviates from their intended representation as logical values, and it can impact the optimization process in undesirable ways. This might explain the underperforming results of deep learning models on categorical data. This observation applies also on non deep models whose training rely on gradient descent. In this chapter we investigate this issue by directing our attention towards the categorical aspect of the data, which is the root cause of the problem, rather than model architectures.

Our main contributions are the following. First, we propose a modification of the standard training loss on categorical data with a related unbiased estimator. Second, we show that this new gradient estimator outperforms standard ones on different datasets. Third, we applied this technique to an in-production model using a private dataset that we are releasing as open source for this purpose. This dataset is substantial, with only a few publicly available datasets in the supply chain domain.

The chapter is organized as follows. After an overview of the recent works on tabular data we point out the issue of applying modern SGD on one-hot-encoded categorical data. Then we propose a novel gradient estimator and show that it is unbiased for a relevant loss on categorical data. We conducted several experiments on public and private datasets that demonstrate the superiority of our proposed gradient estimator over the classical gradient estimator. The chapter ends by a proposal on categorical model initialization when their underlying structure is multiplicative, which is based on singular value decomposition.

III.1. Learning with categorical data

III.1.1. Related works

As described in [7, 121], tabular data exhibit heterogeneity, characterized by high variability of data types and formats, in their underlying structure, unlike images. They involve categorical input attributes and have a strong structure that is unique to each tabular dataset. Modifying a categorical attribute in the input may lead to a complete change in the meaning of the corresponding data, whereas changing a pixel in an image does not fundamentally alter the image. This data kind distinction can lead to different results for the same deep learning architectures, as shown in the case of adversarial learning [124]. Even for simpler tasks such as binary or multiclass classification and regression, deep learning did not yet surpass tree models on tabular data as presented in [121, 125]. Tabular data is depicted by [6] as the last "unconquered castle" of deep learning and multiple works assess the crucial need of further development in this direction [126]. This holds even though various architectures such as MLP [127], ResNet [128], Transformer [25], NET-DNF [129] . . . have been applied to them, as soon as the dataset is actually categorical, i.e. it has mainly nominal attributes [130].

In the literature, one can split the architectures into two categories: the raw deep learning models and the adapted deep learning models. The first rely on some known deep learning models directly applied on tabular data, without any modification of their architecture. One example is the work presented in [131], which attempts to transform the heterogeneous nature of tabular data into a homogeneous numerical representation, in order to apply successful deep learning methods to this type of data. The second one adapts deep learning architectures in order to better fit the tabular data specificity [118, 132, 133].

All these attempts did not outperform the standards models such as XGBoost from [116] or CatBoost from [117] which are still the state-of-the-art in this domain [121]. The evaluation is performed on the Adult Census Income (ACI) dataset [134], which is frequently utilized to showcase the handling of categorical data. In deep learning, each observation is represented as a fixed-size feature vector, where each feature is assumed to provide information about the observation. When using one-hot encoding for categorical data, a vector representation is introduced that contains undefined components, which is inherently challenging for deep learning approaches. Indeed in deep learning architectures, all parameters are typically updated at every iteration except through the use of methods such as Dropout [135] or LayerOut [136] that are general learning tricks non specific to any kind of data. This assumption may not hold true for categorical data, leading to potential inaccuracies in the training process.

Hence, there is a requirement for the development of novel approaches that can better account for the inherent characteristics of categorical data and are better equipped to handle the characteristics associated with this type of data. We now shift our focus to the exploration of categorical models and their encoding techniques. These models specifically address the unique challenges posed by categorical attributes in tabular data.

III.1.2. Categorical models and one-hot-encoding

Id	Cat	Discount (%)	Sales
001	pants	20	7
002	shirt	10	3
003	shirt	15	2
...	...	...	...
00n	shirt	20	8

Table III.1.: Categorical data.

Deep learning methods are designed to work well with numerical data, such as arrays of continuous values, because they are built on matrix multiplication or convolution that are well-defined for numerical data. Categorical data, on the other hand, refers to data that can take on a limited number of discrete values, and there is no existing ordering of these values. Let us denote *categorical models* the set of models that accept categorical attributes by design and are numerical, i.e. their parameters can be updated through gradient descent. By categorical attributes, we denote an attribute whose possible values belongs to an alphabet of n_s symbols $\{s_1, \cdots, s_{n_s}\}$. In Table III.1, *Cat* is the only categorical attribute, while pants and shirt are the symbols, and forms the *Cat* alphabet. We stress that the *categorical* aspect applies to the nature of the input attributes: a categorical model can be used for regression and predict a numerical variable. In that sense regular neural networks are not categorical models as they need numerical inputs. Some specific deep models correspond to this definition as wide models described in [137]. Wide model from [137] consists of two main components: a wide component and a deep component. The wide component captures the memorization aspect by using a linear model that incorporates explicit feature interactions. It allows the model to learn and remember specific feature combinations that have been observed in the training data. This component is beneficial for capturing low-frequency, highly specific patterns. On the other hand, the deep component focuses on generalization by using a deep neural network. It learns complex and abstract representations of the input features, enabling the model to capture high-level interactions and dependencies in the data. The deep component is particularly effective at capturing latent features and non-linear relationships between the features.

The relational linear regression introduced in Section I.5.3 is a categorical model. Let us apply this categorical model to the inputs presented in Table III.1, with the goal of predicting sales based on the discount and the category of the item. This application is visualized in the graphical representation shown in Figures III.1 and III.2.

$$\hat{y}(cat, x) = a_{cat} \times x + b. \tag{III.1}$$

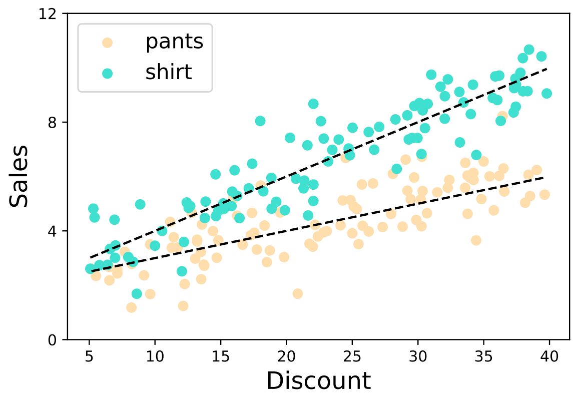


Figure III.1.: Relational linear regression.

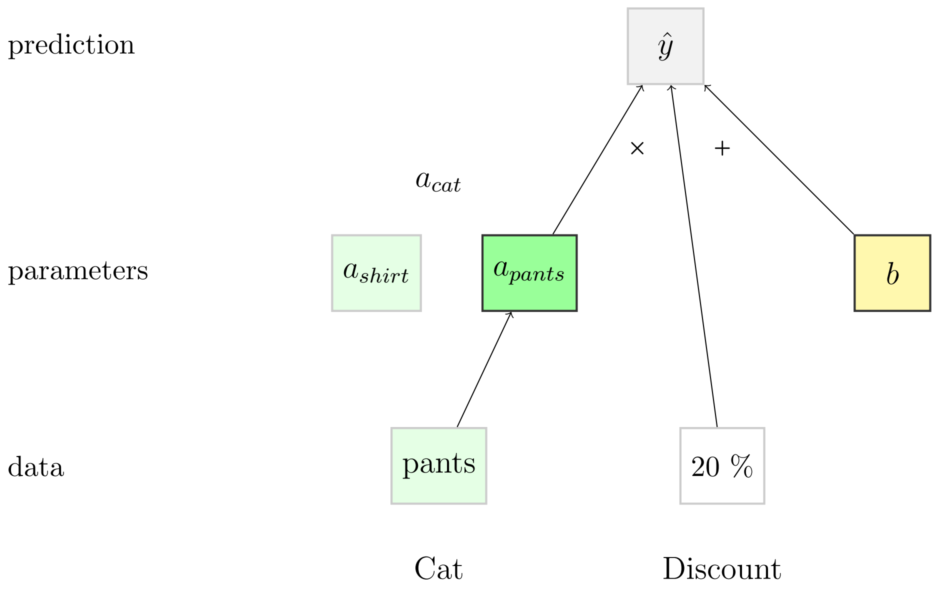


Figure III.2.: Relational linear regression accessing parameters. The slope parameters are not concerned by every observation: parameter a_{pants} is used only for the pants data.

In this application, the parameter a_{cat} has a value for each possible category of *Cat*, and we aim to find the best ones, with the appropriate intercept, in order to build a good predictive model. One of the primary methods to accomplish that is gradient descent. Partly due to the very large amount of data often encountered in practice, *stochastic* gradient descent is used. To apply SGD on categorical models, the categorical data has to be encoded into numerical features. No universally good method of encoding exists and encoding choice should always rely on data (alphabet cardinality, relationships between them . . .). In the following we will focus on one-hot-encoding like [7] because it is precisely what categorical models do. One-hot-encoding a categorical variable with cardinality n is performed by creating n binary vectors for each occurrence of the symbol. If there are few symbols, there are only a few newly created columns. For example, on data stored in Table III.1, one-hot-encoding the attribute *Cat* creates the features is_{pants} and is_{shirt}.

In high-stakes contexts such as disease diagnostics, interpretability of the model is fundamental [12, 138]. In such scenarios, the human expert is expected to make the final decision based on the explainability of the model's results. Tree-based models are known for their interpretability and have been used to interpret deep learning models [139]. In this direction, using one-hot-encoding is crucial. Having parameters directly related to the application semantic by giving access to their relation with the input symbols is a requirement for the design of white-box models. In the illustrated example, the variable a_{pants} holds significant meaning, namely the degree of sensitivity of sales in the pants category to the proposed discount. Parameter values not only serve model prediction quality, they are also *interpretable*. On Model III.1, $a_{pants} > a_{shirt}$ means that the pants sales better react to the discount than the shirt ones. Not only is the prediction of the model explainable, but the trained model itself conveys meaning because the parameters have their own semantics. It also maintains the structure of the data: it does not impose any arbitrary ordering on the nominal categories (no intrinsic order).

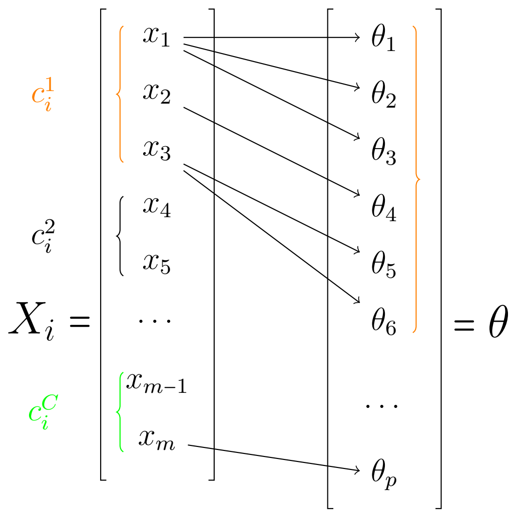
When dealing with low cardinality attributes, one-hot-encoding is a suitable approach to turn them into numerical values. However, if this approach is used for high cardinality attributes, the curse of dimensionality may arise, as explained in [140]. In such cases, alternative encoding methods should be considered. The leave-one-out encoding method transforms a categorical attribute into a numerical feature, offering several benefits such as avoiding the curse of dimensionality. However, this approach does not result in interpretable parameters. For the purposes of this study, we will exclude such high-cardinality categorical attributes, which are not common in domains such as health or supply chain. We also exclude any attribute that has a native ordinal encoding, like size attribute with possible values { small ; medium ; large}.

Applying SGD on categorical models raises an issue as common gradient update techniques are not designed for one-hot encoded categorical features: not every symbol of a categorical attribute is present in every observation of a dataset while regular numerical models assume that every feature is present on every observation. Thus we propose an updated version of gradient estimation used to update categorical parameters. An important characteristic of the approach is its consideration of the categorical nature of the model features.

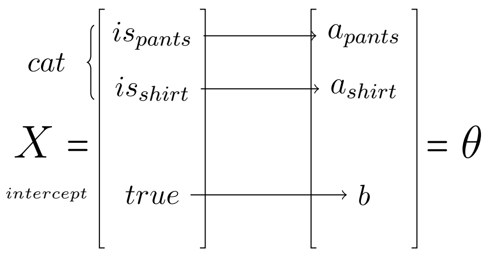
III.1.3. One-hot-encoding notations

We consider the supervised learning set up with a given set of training labeled data $\mathcal{Z} = \{z_i = (\mathbf{C}_i; y_i); i = 1 \dots n\}$, with the attribute vectors $\mathbf{C}_i \in \underset{c=1}{\overset{C}{\times}} A_c$ (cartesian product) where each A_c is an alphabet, i.e. a finite set of $\mid A_c \mid$ symbols. Thanks to one-hot encoding, one can turn the attribute vectors $\mathbf{C}_i$ into boolean features $X_i \in \{false; true\}^m$ where $m = \sum_{c=1}^{C} \mid A_c \mid$. Let's define an arbitrary order on the (disjoint) union of all the alphabets: $\{s_k\}_{k \le m} = \sqcup_c A_c$. Then the boolean vectors X_i are defined as:

$$\forall i \le n \quad \forall k \le m \quad X_i^k = true \Leftrightarrow \mathbf{C}_i^c = s_k. \tag{III.2}$$



(a) Generic notations.



(b) Notations applied to relational linear regression

Figure III.3.: One-hot encoding for categorical models notations. The parameter θ is not entirely dependent on each observation. On the left figure, if x_3 is *false* on an observation, then θ_5 and θ_6 are not concerned by this observation. On the right figure, the intercept is shared among all the observations, resulting in the relative coordinate in the boolean vector being assigned a value of *true*. The arrows serve to summarize the interpretability of the model.

This one-hot encoding is utilized to establish a correspondence with the model parameters. Each parameter of the categorical model is uniquely associated with a specific coordinate of the boolean vector.We aim to find the best parameter $\theta^\star \in \mathbb{R}^p$ ($p \geq m$) to minimize the loss F_θ on the whole dataset:

$$f: \quad \begin{aligned} \mathbb{R}^p \times \mathcal{Z} &\longrightarrow \mathbb{R} \\ \theta, (\mathbf{C}, y) &\longrightarrow f_\theta(\mathbf{C}, y) \end{aligned} \qquad \theta^\star = \operatorname*{arg\,min}_{\theta \in \mathbb{R}^p} \quad F_\theta = \operatorname*{arg\,min}_{\theta \in \mathbb{R}^p} \sum_{\mathbf{C}, y \in \mathcal{Z}} f_\theta(\mathbf{C}, y) = \operatorname*{arg\,min}_{\theta \in \mathbb{R}^p} \sum_{i=1\dots n} f_\theta(\mathbf{C}_i, y_i).$$

Figures III.3 illustrate this formal definition. In Figure III.3a, $\{\theta_j\}_{j=1..6}$ are related to the first attribute c^1. On relational linear regression III.1, such notations give Figure III.3b where the slope is shared among the category while the intercept is shared by all the observations.

The variables X_i^k can be mistakenly abusively as numerical inputs, such as assigning them values of 0 or 1, when feeding them into a machine learning algorithm. This constitutes a misuse when the algorithm specifically expects continuous numerical values as input.

III.1.4. Gradient descent issues with categorical features

In classical SGD, instead of computing the complete gradient on all observations, the observations are divided into *batches* and the gradient is estimated batch wise on them:

$$\widehat{\nabla_\theta F} = g_\theta = \sum_{obs \in batch} \nabla_\theta f_{obs} \tag{III.3}$$

Regarding categorical models and one-hot-encoding, we stress that categorical parameters are not equally concerned by the batch, especially when the batch is made of a single observation. For example in Model III.1 a_{pants} is only used on observations that concern a pants product. By construction via one-hot-encoding, each observation concerns one and only one symbol for each categorical attribute. It is rational to solely update the parameters of the concerned symbol, whereas Equation III.3 computes all the gradient components. In this example, the gradient estimation reduces to:

$$g_{a_{pants}} = \sum_{\substack{obs \in batch \\ cat(obs)=pants}} \nabla_{a_{pants}} f_{obs}. \tag{III.4}$$

What would be the parameter's gradient of a symbol that is not present at all in the dataset? What would be the gradient of a_{hat} in Model III.1 with no hat products in the batch? The set $\{obs \in batch | cat(obs) = pants\}$ from Equation III.4 might be empty. In this case, the parameters related to the *pants* symbol are not concerned by the batch and an undefined gradient is not equivalent to a zero-gradient. Thanks to one-hot-encoding, we have prior information about the gradient: if we encounter an observation that does not involve the symbol s_k, we know with certainty that the gradient of its related parameters does not exist whereas it is numerically zero in standard implementations of SGD. This numerically zero gradient does not convey any information and it should not be used for parameter updates. This atomic property of categorical attributes is completely ignored when using standard SGD approaches. Notice that this problem occurs when an optimizer with momentum is used, where a zero gradient differs in its implications from a non-existent gradient. In this case the structural zeros of categorical data are broadcast among successive batches, which may even amplify the bias of the estimation of the gradient.

$$\begin{gathered} a_{cat} = a_{pants} \times is_{pants} + a_{shirt} \times is_{shirt} \\ \frac{\partial a_{cat}}{\partial a_{pants}} \Big|_{cat=shirt} = \varnothing \quad ; \quad \frac{\partial a_{cat}}{\partial a_{shirt}} \Big|_{cat=pants} = \varnothing. \end{gathered} \tag{III.5}$$

This issue especially concerns under-represented symbols and small batches. The smaller the cardinality of the symbols and the batch size, the higher the likelihood of the symbol not being included in the batch. When a symbol is not present in the batch, we state that its related parameters should not be updated. As a conclusion, the encoding of categorical data should not be part of the gradient-exposed portion of the model and should not infer on the model's parameters updates, as highlighted in Equation III.5.

III.2. Gradient estimator for categorical data

III.2.1. GCE definition

The problem with SGD on one-hot-encoded categorical data arises from the updating of all parameters at each iteration. To address this issue, we propose a new approach that combines a modification of the training loss with a novel gradient estimator. This gradient estimator has been specifically designed to handle categorical data. By combining these two elements, the solution provides a more effective and efficient way of training models on categorical data. The experimentation results show the benefits of this new approach, which has the potential to significantly improve the performance of gradient-based machine learning models on categorical data. Let's consider a given *batch*. All the following is based on the observation that if $\{symbol(obs) = s_k / obs \in batch\}$ is empty, parameters related to the s_k symbol should **not** impact the parameters update in any way. Indeed an undefined gradient is not a zero-gradient. The proposed gradient estimator thus makes the difference between a zero gradient and an undefined gradient. Then one needs to count each symbol occurrence and to apply the unbiased gradient estimator. Therefore, one needs to divide the accumulated gradient by the cardinality of $S_k = \{obs \in batch / symbol(obs) = s_k\}$. If this set is empty, parameters related to the s_k symbol should not be updated. This is presented in Algorithm 1. Note that this quantity varies at each iteration for every symbol of every categorical parameter.

To support this method, we modify the loss function itself to mirror what we truly aim to minimize while working on categorical data. On the given batch, it results to the following loss:

$$\tilde{F}_\theta = \frac{1}{m}\sum_{k=1}^{m}\sum_{\mathbf{C},y \in S_k}\frac{1}{\mid S_k \mid} f_\theta(\mathbf{C}, y). \tag{III.6}$$

We recall that $\{s_k\}_{k \le p}$ is the union of all the alphabet attributes. With this modified loss function, simply summing the gradients of the parameters from Equation III.3 no longer results in an unbiased gradient estimator. It is necessary to calculate the number of terms contributing to the gradient estimator for each symbol of each categorical parameter.

The solution is the gradient estimator for categorical features (GCE) $\tilde{g}_\theta$ presented in Equation III.7 and used by Algorithm 1.

$$\tilde{g}_\theta = \frac{1}{m}\sum_{k=1}^{m}\sum_{\mathbf{C},y \in S_k}\frac{1}{\mid S_k \mid} \nabla f_\theta(\mathbf{C}, y), \tag{III.7}$$

Algorithm 1 **GCE.**

```
Require: $\mathcal{Z}$: data
Require: $update(\cdot,\cdot)$: chosen optimizer
Require: $\theta_0$: Initial parameter vector
    $t \leftarrow 0$
    while $\theta_t$ not converged do $t \leftarrow t + 1$
        Divide $Z$ in $Batches$
        for batch $\in$ Batches do
5:          for symbol $\in$ Alphabet do
                $c_{symbol} \leftarrow 0$
            end for
            $\mathbf{g} \leftarrow \vec{0}$
            for X, y $\in$ batch do
10:             $c_{symbol(X)} \leftarrow c_{symbol(X)} + 1$
                compute $\nabla_{\theta_{t-1}} f_{\theta_{t-1}}(X)$ thanks to $y$
                $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\theta_{t-1}} f_{\theta_{t-1}}(X)$        ▷ accumulate gradient
            end for
            $\theta_t \leftarrow \theta_{t-1}$
15:         for $symbol \in Alphabet$ do
                if $c_{symbol} > 0$ then        ▷ a non-present gradient is not a zero-gradient
                    $\theta_{t,symbol} \leftarrow update(\theta_{t-1,symbol}, \frac{1}{c_{symbol}} \mathbf{g}_{symbol})$        ▷ scaled gradient
                end if
            end for
20:     end for
    end while
```

Equation III.8 states that this estimator is unbiased, proof can be found in the following Section III.2.3:

$$\mathbb{E}[\tilde{g}_\theta] = \nabla \sum_{batch} \tilde{F}_\theta. \tag{III.8}$$

This is a sufficient condition for convergence in the previously presented setting Section II.3 as soon as the target loss satisfies regularity conditions. The loss function depicted in Equation III.6 seems similar to the loss used for classification with unbalanced *output* categories. Let's recall that what we propose here is different as we consider unbalanced *input* symbols. In the case where symbol groups have the same size C then the objective function F_θ resumes to $\tilde{F}_\theta$ on a single batch containing all the observations:

$$
\begin{aligned}
\tilde{F}_\theta &= \frac{1}{m}\sum_{k=1}^{m}\sum_{\mathbf{C},y\in S_k}\frac{1}{\mid S_k\mid}f_\theta(\mathbf{C},y)\\
&= \frac{1}{m}\sum_{k=1}^{m}\frac{1}{C}\sum_{\mathbf{C},y\in S_k}f_\theta(\mathbf{C},y)\\
&= \frac{1}{m\times C}\sum_{\mathbf{C},y\in\mathcal{Z}}f_\theta(\mathbf{C},y)\\
&= \frac{1}{\mid\mathcal{Z}\mid}\sum_{\mathbf{C},y\in\mathcal{Z}}f_\theta(\mathbf{C},y)\\
&= F_\theta,
\end{aligned}
$$

as the m symbol groups form a partition of $\mathcal{Z}$. In this case, our proposed gradient estimator is proportional to the classic one. If one uses the vanilla optimizer, GCE is equivalent to the classic one with a bigger learning rate:

$$\theta_t = \theta_{t-1} - \alpha g_{\theta,t}.$$

In this scenario, the gradient's scale is directly related to the learning rate. However, this relationship does not hold true for adaptive optimizers which are highly dependent on the learning rate. In the case of Adam, the update parameter is approximately bounded by the learning rate, making the scale transfer irrelevant.

Thus, even in a balanced scenario, all the conducted experiments show that it is more effective to have a small learning rate and a large gradient using GCE rather than a large learning rate and a small gradient with adaptive optimizers. Results are presented in Section III.3.

III.2.2. GCE on relational linear regression

Let's consider the data from Table III.1 and compare the value of the gradient after the first iteration with the classical gradient estimator and GCE. Let's consider a batchsize of 3, so the first iteration concerns the 3 first lines of the table, noted $\{(x_1,y_1),(x_2,y_2),(x_3,y_3)\}$. With $\tilde{g}_\theta$ the estimated gradient of $\tilde{F}_\theta$ with the GCE method and g_θ the classical one of F_θ, it gives:

$$
\begin{aligned}
g_b = \tilde{g}_b &= \frac{1}{3}(\nabla_b f(x1,y1) + \nabla_b f(x2,y2) + \nabla_b f(x3,y3))\\
g_{a_{pants}} &= \frac{1}{3}(\nabla_{a_{pants}} f(x1,y1) + 0 + 0)\\
\tilde{g}_{a_{pants}} &= \frac{1}{1}(\nabla_{a_{pants}} f(x1,y1))\\
g_{a_{shirt}} &= \frac{1}{3}(0 + \nabla_{a_{shirt}} f(x2,y2) + \nabla_{a_{shirt}} f(x3,y3))\\
\tilde{g}_{a_{shirt}} &= \frac{1}{2}(\nabla_{a_{shirt}} f(x2,y2) + \nabla_{a_{shirt}} f(x3,y3))\\
g_{a_{hat}} &= 0 \quad \text{but} \quad \tilde{g}_{a_{hat}} = \varnothing.
\end{aligned}
$$

This very simple example with only one categorical attribute with a two element alphabet highlights the specificity of our proposed gradient estimator. As spotted by the equality $g_b = \tilde{g}_b$, if the parameter is not considered as categorical, this does not change anything to its gradient estimation. This example is illustrative and the difference between g_θ and $\tilde{g}_\theta$ has a bigger impact on the parameter updates when there are multiple categorical parameters, as demonstrated in the results presented in Section III.3.

III.2.3. GCE unbiasedness proof

We have defined GCE from the modified loss we aim to minimize, which makes it unbiased by design. Nevertheless, to properly prove that GCE is unbiased, we first need to prove that it is well defined. We initially establish the requirement for a finite amount of time to elapse before drawing a batch with an observation associated with a specific category.

Uniform draw

Let Z be a non-empty finite set and $T \subset Z$ also non-empty. We uniformly draw $m > 0$ elements in Z with replacement, which forms the batch. We focus on the first drawing where at least one of the m drawn elements belongs to T. We note $\tilde{K}$ this drawing. Thus:

$$\begin{aligned} \mathbb{P}(\tilde{K} = 1) &= 1 - \left(\frac{|Z| - |T|}{|Z|}\right)^m = P_1 \\ \mathbb{P}(\tilde{K} = n) &= (1 - P_1)^{n-1} P_1. \end{aligned} \tag{III.9}$$

Thanks to Equation III.9, the expectancy of the stopping time can be computed.

Theorem 5 (Stopping time). $\mathbb{E}[\tilde{K}] = \frac{1}{P_1}$.

Proof.

$$\begin{aligned} \mathbb{E}[\tilde{K}] = \sum_{n=1}^{\infty} n\mathbb{P}(\tilde{K} = n) &= \sum_{n=1}^{\infty} n(1 - P_1)^{n-1} P_1 \\ &= \frac{P_1}{1 - P_1} \sum_{n=1}^{\infty} n(1 - P_1)^n. \end{aligned}$$

For $0 < x < 1$ we get:

$$\begin{aligned} \sum_{n=1}^{\infty} n x^n &= \sum_{n=1}^{\infty} x \frac{\partial x^n}{\partial x} \\ &= x \frac{\partial}{\partial x} \sum_{n=1}^{\infty} x^n \\ &= x \frac{\partial}{\partial x} \sum_{n=0}^{\infty} x^n \\ &= x \frac{\partial}{\partial x} \frac{1}{1 - x} \\ &= \frac{x}{(1 - x)^2}. \end{aligned}$$

Then

$$\begin{aligned}\sum_{n=1}^{\infty} n\mathbb{P}(\tilde{K}=n) &= \frac{P_1}{1-P_1}\frac{1-P_1}{P_1^2}\\ &= \frac{1}{P_1}.\end{aligned}$$

□

Remark 6. *It is the same result if the drawings are done without replacement. The only difference is a higher P_1.*

Now we can prove the unbiasedness of our estimator, which comes naturally as it was designed with the loss itself.

Estimator

Let Z be a non-empty finite set and $T \subset Z$ also non-empty.

We have a score function s on T:

$$\begin{aligned} s \quad : \quad & T \longrightarrow \mathbb{R}\\ & t \longrightarrow s(t)\end{aligned}$$

We aim to estimate

$$s_T = \frac{1}{|T|}\sum_{x\in T} s(x).$$

Let $(M_k)_{k\le K}$ a series of K draws uniform with replacement of m elements of Z.

Remark 7. *Thanks to Theorem 5 we can ignore the first draws M_0 such as $M_0 \cap T = \varnothing$*

One notes

$$M_k = \underbrace{(M_k \cap T)}_{M_k^T} \sqcup (M_k \cap (Z\backslash T)),$$

and

$$\underset{M_k^T}{\text{avg}}\ s = \begin{cases} 0 \quad \text{if} \quad M_k^T = \varnothing \\ \frac{1}{|M_k^T|}\sum_{x\in M_k^T} s(x) \quad \text{otherwise,}\end{cases}$$

$$\bar{K} = |\{k \le K | M_k^T \ne \varnothing\}|$$

Thanks to Remark 7 we have $\bar{K} \ge 1$. Then the proposed estimator is $\hat{a}$:

$$\hat{a} = \frac{1}{\bar{K}}\sum_{k=1}^{K} \underset{M_k^T}{\text{avg}}\ s.$$

Theorem 6 (Unbiased estimator)**.** *$\hat{a}$ is an unbiased estimator of s_T*

Proof.

$$\begin{aligned}\mathbb{E}[\tilde{a}] &= \frac{1}{\hat{K}} \sum_{\substack{M_k^T \neq \varnothing \\ k=1}}^{K} \frac{1}{\mid M_k^T \mid} \sum_{x \in M_k^T} \mathbb{E}[s(x)] \\ &= \frac{\bar{K} \mid M_k^T \mid}{\bar{K} \mid M_k^T \mid} \mathbb{E}[s_T] \\ &= s_T.\end{aligned}$$

□

III.3. Experimentations

We have implemented Algorithm 1 in two different scenarios and programming languages: deep learning models and categorical models both using one-hot-encoded categorical data. In both cases, we aim to assess the impact of GCE. To evaluate its effectiveness, we compare its performance with the current treatment of categorical parameters in batch gradient descent. We use the public datasets listed in Table III.2 as well as a private dataset from the supply chain domain for our evaluations.

Regarding public datasets, Adult Census Income (ACI) dataset [134] aims to predict the wealth status of individuals, Compas dataset predicts the likelihood of re-offending among criminal defendants, Forest Cover dataset [141] predicts the forest cover type based on categorical characteristics of $30m^2$ forest cells, KDD99 dataset [142] aims at predicting cyber-attacks, Don't Get Kicked (DGK) dataset [143] predicts whether a car purchased at auction is a good or a bad buy. Used Cars dataset from Belarus contains vehicle attributes and aims to predict the selling price of the car.

Dataset	Chicago	ACI	Compas	DGK	Forest Cover	KDD99	UsedCars
observations	194m	48k	7.2k	72k	15k	494k	38k
inputs number	2	7	7	9	2	4	11
task	REG	CLASSIF	CLASSIF	CLASSIF	CLASSIF	CLASSIF	REG
max cardinality	7.9k	42	341	1k	40	66	1.1k

Table III.2.: Dataset characteristics. We provide information on the size of the datasets and the maximum cardinality of their categorical attributes. For instance, in the Forest Cover dataset, no categorical input has more than 40 possible values.

The chosen metrics for evaluating the performance are the mean squared error (MSE) for regression tasks (REG) and the error rate (i.e., $1 - \mathit{Accuracy}$) for classification tasks (CLASSIF). It should be noted that during the training of the models using GCE, the corrected loss $\tilde{F}_\theta$ is utilized, while the standard loss F_θ is employed to evaluate the performance on the test dataset.

III.3.1. Application to deep learning

In our study, we utilized PyTorch [40] for implementing our proposed solution for deep learning models. The framework provides ease in updating the gradient of every parameter using Algorithm 1. The code and the corresponding experiments can be accessed through the GitHub repository[1]. To evaluate the effectiveness of our solution, we conducted experiments on six different datasets with categorical data: ACI, Compas, DGK, Forest Cover, KDD99 and UsedCars.

In order to only measure the impact of GCE, we only use the categorical variables in our experiments. Those dataset tasks are quite easy. As a consequence we use small networks to highlight our approach. The MLP network is made up of 3 dense layers of sizes $[4, 8, 4]$. We also perform experiments on a ResNet-like network very similar to [7]. We have tested three different optimizers with their default settings: SGD (vanilla), AdaGrad and Adam. Tests have been run on several batch sizes: $2^{5 \dots 10}$. To record the results, each experiment has been run 10 times. Results are reproducible in the repository and are recorded in Tables A.1 A.2 A.3 A.4 A.5 A.6 A.7 A.8 A.9 A.10 A.11 A.12 in the appendices A.2.2. In our experiments, we found that the use of GCE resulted in improvement in loss on the test dataset. Figure III.4 presents the performance of GCE on the Adult Census Income (ACI) dataset. The bigger the batch, the less GCE outperforms the classical estimator. It is logical as in big batches, more symbols are concerned. This proves the need to specifically handle stochastic gradients on categorical data. Results in different settings demonstrate the advantage to use GCE whatever the optimizer. For instance, while AdaGrad has been designed to handle gradients on sparse data (including one-hot encoded data), the use of GCE still resulted in a clear improvement in performance. It is noteworthy that our experiments utilized compact network architectures and solely concentrated on the categorical characteristics of the dataset. This was done to isolate the impact of GCE, thereby excluding input variables such as "age" or "income" on the ACI dataset. Despite these stringent limitations, our approach achieved an accuracy of 83% (as shown in Table A.4) on this dataset when employing GCE. This result is comparable to the state-of-the-art of deep learning, as reported in [121]. Only boosting methods have exceeded 87% accuracy, and they have employed all the features, including the non-categorical ones.

III.3.2. Categorical model on public datasets

The experiments for categorical models were conducted using the Envision Domain Specific Language for Supply Chain, a Python-like implementation of SQL designed for supply chain problems. This language includes a differentiable programming layer as described in [77] that provides access to the gradients of categorical models. Stochastic optimization using Adam and a relational linear regression were compared on two publicly available datasets: the Chicago Taxi ride dataset [91] and the Belarus used car dataset [144].

For each ride of the Chicago Taxi dataset, we use the taxi identifier, distance, payment type and the tip amount. We use an extended version of the relational linear regression to predict the tip based on the trip distance and the payment type. The slope depends on the taxi and the payment method, the intercept remains shared among all the trips,

[1]https://github.com/ppmdatix/GCE

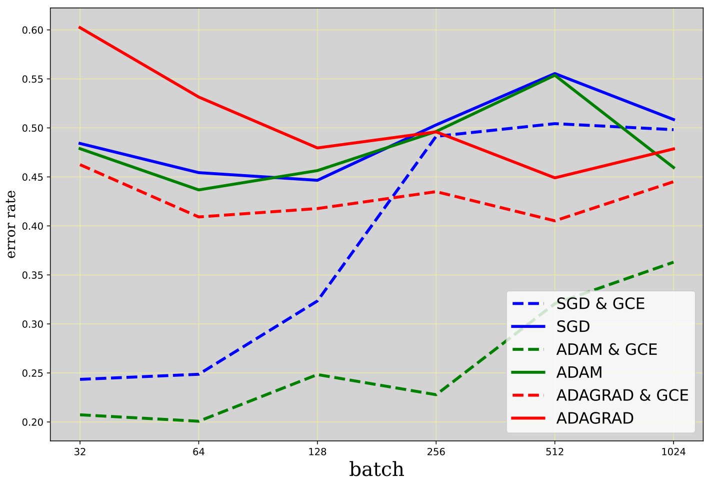


Figure III.4.: Results (error rate) on the ACI dataset with the ResNet-like network. The dashed curves represent experiments with **GCE** and show an improvement on the test loss for every optimizer used, regardless of the batchsize.

as presented in Equation III.10:

$$\hat{tips} = (\gamma_{\text{taxi}} \times \mu_{\text{payment}}) \times distance + b. \tag{III.10}$$

There is one γ per taxi and also one μ per payment method, that would fit the presented setting with the $Taxi \times Payment$ cross vector construction. As the intercept is shared among all taxis, the dataset is unsplittable while a model based on Equation III.11

$$\hat{tips} = \gamma_{\text{taxi}} \times distance + b_{taxi}, \tag{III.11}$$

could be split into different datasets (one per taxi) and thus we would be in the classical setting of a linear regression.

We also worked on the Belarus used cars dataset. We take into account the car manufacturer, the production year, the origin region of the car to predict the selling price of the car as presented in Equation III.12.

$$\hat{price} = (\gamma_{\text{manufacturer}} \times \mu_{\text{region}}) \times year + b. \tag{III.12}$$

As reported in Table III.3, the relational batch performed better with our proposition based on Algorithm 1 with the following setting: 10 epochs ; optimizer Adam with default setting ; batch size of 1. Experiment was reproduced 20 times. The experiments conducted on the Used Cars dataset can be executed at `https://try.lokad.com/s/Peseux-PhD-BelarusCars`.

III.3.3. Categorical models in production

We applied GCE on the categorical model presented in Section I.5.4. It is a multiplicative model used daily on retail data. We employed Adam optimizer with its default values along with GCE and SGD for updating the parameters. The use of GCE results in a significant improvement in the performance of the categorical model as compared to the

Dataset	Adam	Adam & GCE
Chicago Ride	35.58 ± 1.11	$\mathbf{9.45 \pm 1.63}$
Used Cars	$(1.30 \pm 0.03).10^{-2}$	$\mathbf{(1.1 \pm 0.07)}.10^{-2}$

Table III.3.: Results (RMSE) of relational linear regression applied to categorical datasets. GCE strictly outperforms the traditional gradient estimator on these two examples.

classical gradient estimator. The testing dataset's final loss, measured in terms of decayed MSE, is about an order of magnitude better with GCE. However, it is worth noting that while GCE works well in practice, multiplicative models do not meet the assumptions outlined in Section II.3. Hence, there are no convergence guarantees, and the third assumption remains unsatisfied. To give a glitch of why such a multiplicative model gradient is not L-Lipschitz-continuous, let's consider $h : \mathbb{R}^3 \to \mathbb{R}$ such as $h(x, y, z) = xyz$. Then its gradient is easily computed:

$$\nabla h(x, y, z) = \begin{pmatrix} yz \\ xz \\ xy \end{pmatrix}.$$

Then the difference of the gradient can not be bounded above by the difference of the parameters. To prove this, lets consider $a, b \in \mathbb{R}$:

$$\begin{aligned} \|\nabla h(a, a, a) - \nabla h(b, b, b)\|_2^2 &= 3(a^2 - b^2) \\ &= 3(a - b)^2(a + b)^2 \\ &= (a + b)^2 \times \left\| \begin{pmatrix} a \\ a \\ a \end{pmatrix} - \begin{pmatrix} b \\ b \\ b \end{pmatrix} \right\|_2^2 . \end{aligned}$$

This is valid for any reels a and b, which proves that this difference cannot be properly controlled. However this is not harmful because II.3 setting is a sufficient in theory one but not necessary to observe convergence in practice. For many neural networks, it is not clear if parameters are supposed to converge, but with proper learning parameters it often does.

We have discussed optimization of categorical models through SGD using GCE but we have iinly considered random initialization of our parameters. However appropriate initialization of a model can reduce the learning time and prevent the model from getting trapped in local minima. While it may be infeasible for deep learning models agnostic from the data structure, initialization is a tractable task for categorical models. We present our findings in the following.

III.3.4. Categorical models initialisation

Two features example

Consider the multiplicative model presented in Section III.3.3 without the size vector for simplicity. The categorical parameter for size is excluded as it has an ordinal order. In this model, the parameters θ_{store} and θ_{color} are associated with categories while the parameter vector Θ aims to capture the seasonality. Now, we focus on the initialization of θ_{store} and θ_{color}, which are meant to be independent of the seasonality by construction of the model. Thus we would like

$$\forall s \in Stores, \forall c \in Colors; \quad \theta_s \times \theta_c \sim \underset{i \in A^{s \times c}}{avg} \quad \theta_{store(i)} \times \theta_{color(i)}, \tag{III.13}$$

with $A^{s\times c} = \{item \mid store(item) = s \wedge color(item) = c\}$.

With traditional vector notations and with n_c the number of colors and n_s the number of stores, we are looking $\vec{c}, \vec{s} \in \mathbb{R}^{n_c} \times \mathbb{R}^{n_s}$ such that

$$\vec{c} \otimes \vec{s} = A,$$

with $A \in \mathbb{R}^{n_c \times n_s}$

Of course, not every matrix $A = [a_{i,j}]$ can be factorized this way, simply because if $n_c, n_s > 2$ then $rank(\vec{c} \otimes \vec{s}) \leq n_c + n_s < n_c \times n_s$. Consequently, as soon as the rank of the matrix A is too large, this factorization is unfeasible.

But let assume that such factorization exists and that A is not the zero matrix. Without loss of generality, we can also assume that $\|\vec{c}\| = 1$. We also assume that $A\vec{1} \neq \vec{0}$. Then

$$\vec{c} \otimes \vec{s}.\vec{1} = A.\vec{1}.$$

Which means that $(\sum_{j=1}^{n_s} s_j)\vec{c} = A.\vec{1}$. Then one can obtain $\vec{c}$:

$$\vec{c} = \frac{1}{\|A.\vec{1}\|} A.\vec{1}. \tag{III.14}$$

Then $\vec{s}$ can also be constructed because we know that there exists $i_1 \leq n_c$ such as $c_{i_1} \neq 0$. Then

$$\forall j \leq n_s \quad s_j = \frac{a_{i_1 j}}{c_{i_1}}. \tag{III.15}$$

We have demonstrated with Equations III.14 and III.15 that if such a factorization of matrix A is possible, there is a straightforward method to obtain it. This factorization serves as an excellent starting point for the initialization of our multiplicative model parameters. However, since an exact factorization is not always feasible, our goal is to find an approximation. In the following section, we introduce and discuss this approximation method, providing an example of its application on the Celio dataset.

Generalization to Singular Value Decomposition

The issue of initializing parameters properly can be resolved by employing matrix decomposition. Specifically, any matrix A can be represented through its Singular Value Decomposition (SVD), given by

$$A = U\Sigma V^\star, \tag{III.16}$$

where U and V are unitary matrices, and Σ is a diagonal matrix with non-negative and non-increasing values on its diagonal. This decomposition is not unique, and its existence can be derived using the spectral theorem (see Appendices A.2.1).

By definition, Σ can be written:

$$\Sigma = \sum_i \sigma_i P_i,$$

with $P_i = e_i \otimes e_i$ the outer product of standard basis vectors. Applying this notation to the Equation III.16 gives:

$$A = \sum_i \sigma_i U_i \otimes V_i. \tag{III.17}$$

Consequently, the best approximation of the matrix A through an outer product is given by the term with the largest singular value, that is, $\sigma_1 U_1 \otimes V_1$. Such practical decomposition is proposed in Example 4 If the other σ_i are equal to zero, it reduces to the previous section with the matrix being factorized with only two vectors.

Using the first term of this matrix decomposition as an approximation provides a good starting point for initializing the parameters of any multiplicative model, such as the one presented in Section I.5.4.

Example 4. *Considering a $\mathbb{R}^{3\times 3}$ matrix, its SVD results in the following:*

$$\begin{bmatrix} 9 & 3 & 6 \\ 8 & 8 & 6 \\ 0 & 3 & 7 \end{bmatrix} \sim \underbrace{17.5 \begin{bmatrix} 0.62 \\ 0.72 \\ 0.32 \end{bmatrix} \otimes \begin{bmatrix} 0.65 \\ 0.49 \\ 0.59 \end{bmatrix}}_{\textit{best approximation}} + 5.57 \begin{bmatrix} 0.37 \\ 0.09 \\ -0.92 \end{bmatrix} \otimes \begin{bmatrix} 0.73 \\ -0.17 \\ -0.66 \end{bmatrix} + 3.26 \begin{bmatrix} 0.69 \\ -0.69 \\ 0.21 \end{bmatrix} \otimes \begin{bmatrix} 0.23 \\ -0.86 \\ 0.47 \end{bmatrix}.$$

(The equality is not exact due to numerical imprecision). The Python code can be found in the Appendix Listing A.3

The key idea behind SVD is to capture the underlying structure of the data by decomposing it into a set of simpler components. By doing this, it becomes possible to represent the data in a more compact form, which can be useful for compression, visualization, and other applications.

The issue of initialization for multiplicative categorical models is of critical importance at Lokad to save daily computing resources. To the best of our knowledge, the proposed initialization technique for multiplicative categorical models is a novel contribution. We have employed this technique on the anonymized Celio dataset to determine the optimal initialization for our categorical parameters. Specifically, we focused on initializing two categorical parameters while leaving the others at random initialization. The experimental results can be found in the repository, specifically in this notebook [2]. Figure III.5 illustrates the effectiveness of using SVD for initializing parameters in a multiplicative model. Note that the similarity between the errors obtained from the zero vectors and the second term of SVD is coincidental. The first term obtained through SVD (represented by the green dashed line) yields the most accurate result, while subsequent SVD

[2] `https://github.com/ppmdatix/GCE/blob/main/SVD-Initialisation/svd.ipynb`

terms and the 0 matrix (generated by taking the outer product of two zero vectors) fail to outperform random initializations.

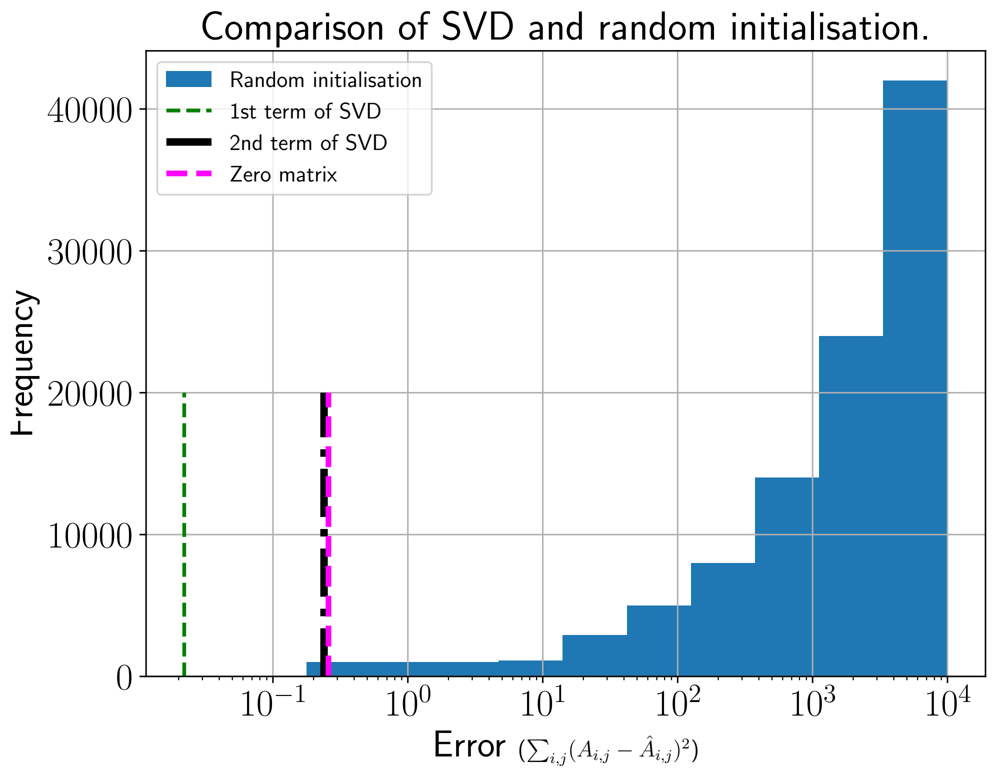


Figure III.5.: The histogram depicts the error distribution in estimating a matrix that represents the average sales based on two attributes of the Celio dataset (in its anonymized form). In order to estimate this matrix, we utilize two vectors, each from a corresponding attribute table, and employ their outer product. For the purpose of random initialization, we tested 100,000 combinations.

Conclusions

This chapter focuses on the challenge of using SGD for machine learning on categorical data. One-hot-encoding is proposed as a solution for creating interpretable models from categorical data, however, this encoding method can result in incorrect gradients and incorrect training results. The novel gradient estimator presented overcomes this problem by recognizing that a non-present gradient should not be considered as a zero-gradient. This new estimator allows for the correct treatment of categorical data in gradient-based models, including deep learning. The results of the study, including code and details, are open-sourced and demonstrate the utility of the proposed solution on various datasets, including an in-production supply chain model. This model was also used as an example to emphasize the significance of appropriate initialization of a categorical model. It was demonstrated how singular value decomposition can resolve this issue for multiplicative models.

This chapter is motivated by the lack of attention given to categorical data in both public datasets and the field of machine learning as a whole. The primary goal of introducing the GCE is to highlight the significance of categorical data and encourage the development of new techniques that effectively address its unique characteristics. Moving forward, the focus of this work is to enhance the implementation of GCE in deep learning, thereby facilitating its adoption within the research community. Furthermore, the broader adoption of differentiation on relational queries, as discussed in the first chapter, will also benefit GCE, as it aligns with the framework's targeted models.

In the preceding chapter, we introduced two distinct sources of stochasticity for gradient estimation: one based on the observations and the other based on the function itself. GCE is derived from a reorganization of the gradient contributions within each batch. As a result, the subsequent and final chapter focuses on harnessing the stochasticity achievable through the decomposition of the LCG . This decomposition has the potential to mitigate memory consumption and enhance the optimization process.

IV. Gradient code stochasticity, overfitting and memory consumption

The work presented in this chapter is currently undergoing review for publication in a journal.

The upcoming section serves as a link between Chapters I and III, bringing together the disparate fields of compilation and optimization. Communication between these two communities is limited, and this work aims to bridge this gap.

Introduction

In the field of gradient-based models for solving classification and regression problems, the use of automatic differentiation facilitates the training of such models as the expert does not need to hand-code the gradient. However, this process can be resource-intensive, particularly with respect to memory consumption during the reverse mode of AD, which is necessary for these types of problems. To reduce memory usage, checkpointing can be employed as a trade-off between execution speed and memory consumption [24].

In addition to the resource consumption issue, overfitting the training data is a common problem that can reduce the generalization power of the model. To mitigate this issue, the dropout method was introduced in [135]. Dropout involves temporarily turning off certain nodes in the execution graph during training.

To address the issue of memory consumption without relying on checkpointing, Randomized Automatic Differentiation (RAD), a novel gradient estimator was proposed in [114]. RAD, which is unbiased, is built by drawing random paths through the backpropagation execution graph, effectively acting as backpropagation dropout. The unbiasedness of the estimator is crucial as it ensures that the convergence properties of the gradient estimator are preserved. The use of a uniform distribution proposed by [114] for drawing random paths is just one possibility, but other distributions may lead to better learning results. Furthermore, the "best" distribution may depend on the training dynamic, and it should be selected with respect to the learning stage. Ideally, the best distribution should be optimized through heuristics that may emphasize the most important paths for the gradient for the current iteration of the training process.

Our approach to constructing the gradient estimator is to keep it unbiased regardless of the path distribution, and we have tested various distributions and found one that outperforms the uniform one in various settings without increasing memory consumption.

IV.1. Memory consumption and gradient based methods

IV.1.1. Checkpointing

The process of reverse mode automatic differentiation results in programs with a specific structure. The adjoint program $\overline{P}$ consists of a forward pass $\vec{P}$ followed by a backward pass $\overleftarrow{P}$. When executing $\overleftarrow{P}$, certain values computed during $\overline{P}$ are required. There are two options to access these values: store them during the execution of $\overline{P}$ or recompute them. This choice requires a trade-off between time and memory usage. In checkpointing strategies, only the checkpoint values are stored, whereas the others are recomputed from the most recent checkpoint when needed in the forward pass computation graph.

```
$x, y \longleftarrow \dots$        // Checkpoint?
   $z \longleftarrow x \times y$
        $\dots$
   $\overline{z} \longleftarrow \dots$
   $\overline{x} \longleftarrow \overline{z} \times y$
   $\overline{y} \longleftarrow \overline{z} \times x$
```

Listing IV.1.: Reverse mode automatic differentiation and checkpointing. Variables x and y are computed in the forward pass and reused in the backward one. Checkpointing techniques arbitrate between storing or recomputing them.

As presented in Listing IV.1, to calculate the adjoint of x, the value of y computed in the first pass is required. Two options exist to access this value: either store y or recompute it using the checkpointed values of x and y.

There exists two checkpointing tactics without any trade-off: STORE-ALL and RECOMPUTE-ALL.

STORE-ALL

STORE-ALL strategy involves storing all intermediate values during the forward pass of the computation graph, so that during the backward pass, all necessary values are readily available for efficient gradient computation. This requires significant memory usage to store all intermediate values, but reduces the computational cost of the backward pass as all values can be accessed without recomputation. It is presented in Figure IV.1.

RECOMPUTE-ALL

RECOMPUTE-ALL strategy involves recomputing all intermediate values during the backward pass. This reduces the memory requirements, as intermediate values are not stored, but increases the computational cost of the backward pass as recomputation is required. It is presented in Figure IV.2.

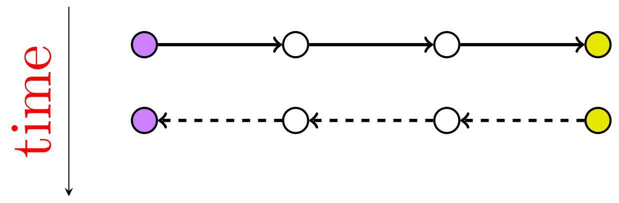


Figure IV.1.: STORE-ALL. The forward paths are represented as the full lines while the dashed ones represent the backpropagation. The input node is represented by the purple nodes on the left, whereas the output node of the model is denoted by the yellow node on the right. All intermediate variables are stored.

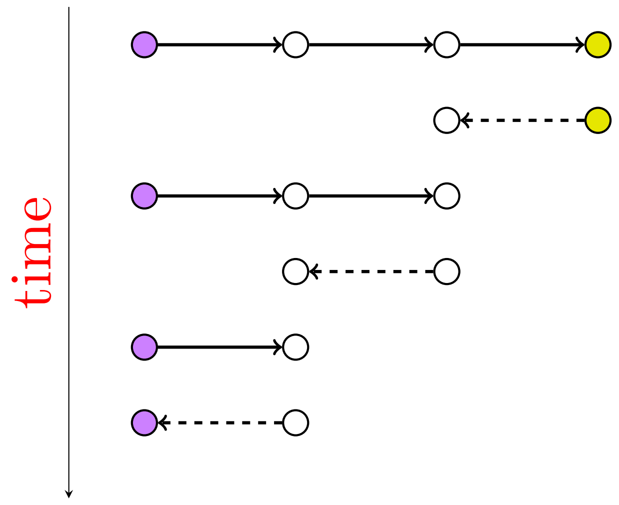


Figure IV.2.: RECOMPUTE-ALL. No intermediate variables are saved, one needs to recompute each one of them from the start (in purple) to determine the adjoint. A function execution using this strategy is presented in Appendix A.3.1

Hybrid strategies

Hybrid strategy combines STORE-ALL and RECOMPUTE-ALL approaches. It performs periodic checkpointing during the forward pass, storing the necessary activations and recomputing the remaining activations during the backward pass. This strategy aims to strike a balance between the memory requirements of STORE-ALL and the computational overhead of RECOMPUTE-ALL. The frequency of checkpointing can be tuned based on the available memory and computational resources. If the available memory is limited, more frequent checkpointing may be necessary to avoid running out of memory. On the other hand, if computational resources are limited, less frequent checkpointing may be more appropriate to reduce the overhead of recomputing activations. Hybrid strategies can be more efficient than either STORE-ALL or RECOMPUTE-ALL approaches alone, but they require careful tuning to achieve optimal performance.

While optimal strategies have been identified in some specific cases [145], the general case is still an active area of research.

IV.1.2. Adsl take on checkpointing

Adsl is designed to make differentiation as easy as possible, so its practical implementation has to tackle the checkpointing issue.

First, it should be noted that the SA property of Adsl does not completely eliminate all checkpointing issues. If an intermediate variable is required for both forward and backward passes, both the STORE-ALL and RECOMPUTE-ALL solutions may satisfy the SA property. If the STORE-ALL solution is chosen, it is necessary to store a variable created by the tupling of the intermediate variable. On the other hand, recomputing the intermediate variable is also SA since its original version is used only once in the forward pass.

Then Let us recall that in a categorical model, the stochastic loss only employs the parameters that are associated with the observation table line, as shown in Figure III.2. Adsl has been especially crafted for such categorical models so only the concerned fraction of the parameters are loaded for each observation. Furthermore, due to the utilization of PolyStar, each observation pertains to only a small fraction of the model parameters. Therefore, memory consumption is not a significant concern during the computation of stochastic loss. Consequently, in our implementation of automatic differentiation in Adsl, we have chosen to store almost every variable and rely on the STORE-ALL tactic.

The only exception is about the states of a loop that are recomputed for the computation of the adjoint. Consider that a loop is fundamentally $w_i = L(r_i)$ (L loops on a read vector r_i and writes the vector w_i. Differentiating it becomes $\overline{r_i} = \overline{L}(r_i, w_i, \overline{w_i})$. We reduce this to $\overline{r_i} = \overline{L}(r_i, \overline{w_i})$ because the $\overline{w_i}$ themselves are either sums (the value of which cannot have an effect on the gradient) or output arrays, which can be recomputed from the ri instead. Since a copy of the body of this loop already exists in the list of statements, we cannot let it keep its variables, so we create a remapper that will be invoked on all the variables inside the loop. We emit a copy of the loop to export the initial state as an array as it is needed for the reverse pass.

IV.1.3. Embedded artificial intelligence

The range of applications for machine learning methods based on gradient descent is vast, encompassing fields such as healthcare and supply chain management, as discussed in previous chapters. However, there are numerous problems where these models could be highly beneficial, but their implementation is constrained by limited computing resources. This area is referred to as embedded artificial intelligence, which involves deploying machine learning algorithms and techniques within resource-constrained hardware devices. This subject is extensively studied [146, 147]. While it is impossible to list every such device, several notable examples can be mentioned. First, smart vehicles employ embedded artificial intelligence for tasks such as lane detection, object recognition, and path planning [148]. These vehicles must strike a balance between computational power and accurate driving performance. They cannot rely on a network connection for computations, as the system must function in all environments, including tunnels. Second, wearable devices like fitness trackers, smartwatches, and health monitoring devices depend on embedded artificial intelligence for various tasks, including activity recognition, heart rate monitoring, and sleep tracking. These devices typically possess limited processing power due to their small size and lightweight design. Similarly, drones and autonomous robots utilize machine learning for navigation, object detection, and obstacle avoidance, all while operating within the constraints of their available processing power [149]. In each of these applications, the machine learning models must function effectively despite the limitations of the hardware they are deployed on.

As discussed in Section IV.1.1 above, the most resource-intensive aspect of a gradient-based model is the backpropagation of the gradient. Consequently, one of the primary challenges in developing embedded artificial intelligence is to reduce the memory consumption associated with this part of the model optimization process. Checkpointing is an interesting technique, as an appropriately tailored hybrid strategy can limit the memory requirements for training a model. However, this approach comes at the expense of longer training times, which is often an unacceptable trade-off for many applications in embedded artificial intelligence.

IV.2. Overfitting the data and dropout technique

In addition to memory consumption, under or overfitting is a common problem in machine learning, where the model learns to fit the training data too closely or not enough, resulting in poor performance on new, unseen data. If the model is too simplistic or under-parameterized, it may struggle to effectively represent the underlying data. This leads to bias in the model, commonly referred to as model bias, resulting in an underfitting situation. On the other hand, if the model is excessively complex or over-parameterized relative to the size of the data, it has the capacity to learn the noise or variance present in the data. This phenomenon is known as over-learning, and it often leads to poor generalization performance on new, unseen data. A graphical illustration of overfitting is given in Figure IV.3.

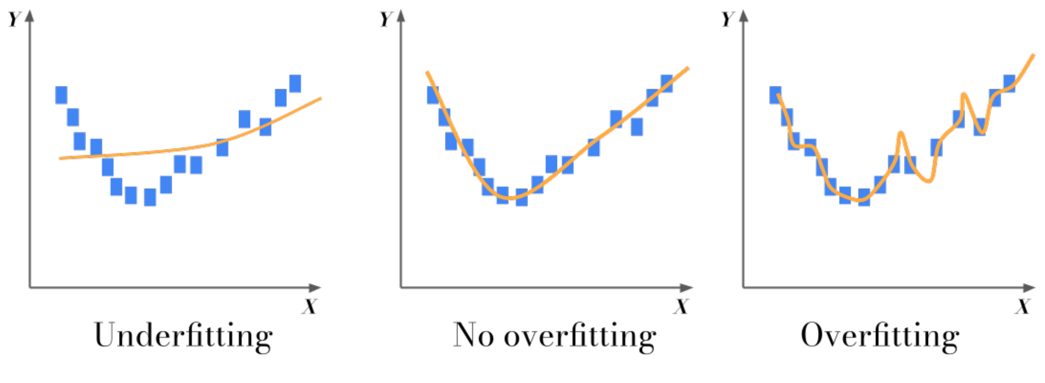


Figure IV.3.: Illustration of overfitting with a model that aims to predict Y in function of X. On the left, the model represented with the orange line does not fit data enough. On the center, the model captures the pattern of the data without being exact on every observation. On the right, the model is exact on every training observation but loses its generalization capabilities.

One common approach to prevent overfitting is dropout [135] which can help prevent the model from memorizing the training data too closely. It is a regularization trick that randomly deactivates some neurons in the architecture as presented in Figure IV.4.

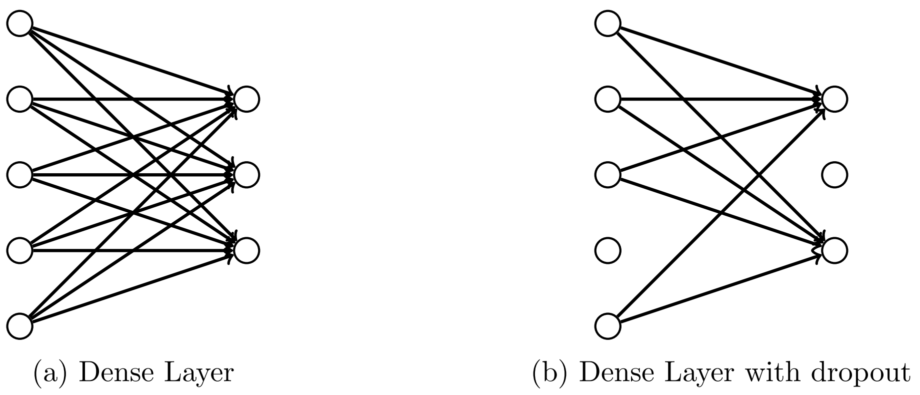

(a) Dense Layer (b) Dense Layer with dropout

Figure IV.4.: Dense layer and Dropout. Turning off a neuron involves turning off all the parameters related to this one. In (b), the fourth neuron of the first layer and second neuron of the second layer are turned off.

Dropout can be seen as a form of model averaging [150]. This encourages the network to learn more robust features, as it is forced to rely on a subset of neurons rather than relying on a single, highly correlated group of neurons. There is extensive research to show that applying dropout does not remove the convergence properties, the key point is that the averaging of the network gives an unbiased estimate of the gradient, which is sufficient in certain conditions.

During the training process with dropout, the number of active weights in the network is limited to a fraction of the total weights of the network, which highlights the fact that smaller networks could theoretically achieve the task, thus reducing the size of the network. However, current implementations of dropout do not facilitate memory reduction, as they primarily focus on reducing overfitting and do not take into account such size reduction.

Dropout techniques are applicable to deep learning models where parameters do not have a specific meaning and can be randomly deactivated. However, they are not suitable for white box models like categorical models. In such models, if a parameter is associated with a category through one-hot encoding, deactivating it is equivalent to deactivating the input vector. Consider the application of dropout on relational linear regression as illustrated in Figure III.2. This would entail estimating the target value using either the slope or the intercept alone. Such an approach is not logical. Therefore, dropout cannot be applied to these models, even though the resulting regularization and memory consumption reduction are desirable features. An intermediate solution is to apply dropout not during the loss calculation, but during the backpropagation of the gradient.

This technique would preserve the output of the model, while introducing stochasticity to the gradient computation from the code of the loss itself (derived from the architecture of the model), and not only from the dataset split into batches. The process is illustrated below.

IV.3. Sample random paths

In Section II.2, we have presented how the estimation of the gradient can be on the observations or on the code itself. In the following section we will focus on the stochasticity obtained from an appropriate randomization of the code.

IV.3.1. Gradient code stochasticity

Thanks to the introduction of LCG in Section II.2.1, and thanks to Equation II.33, the gradient is decomposed as a sum of all the path contributions. This decomposition can be generalized as Equation IV.1, regardless of the source of each term of the sum.

$$\nabla_\theta f = \sum_{i=1..N} g_{\theta,i}. \tag{IV.1}$$

We will stick to the formulation where each $g_{\theta,i}$ is related to a specific backpropagation path, even though all the following applies to optimization problems where the objective function is expressed as a sum, since the derivative operator is linear.

The RAD approach [114] employs a uniform distribution across all possible paths. However, we suggest that not all gradient paths are equally important at any given time of the optimization process. Therefore, we aim to go further by utilizing a non-uniform distribution with varying probabilities to draw a $g_{\theta,i}$ to use it during gradient descent. Let us define $I_t \sim (p^t_{\theta,1}, \dots, p^t_{\theta,N})$ the probability to draw $g_{\theta,i}$ to compute gradient descent, defined over the $T \in \mathbb{N}$ epochs. For notational simplicity, we omit θ, which gives: $I_t \sim (p^t_1, \dots, p^t_N)$. We have

$$\forall t \leq T \quad \sum_{i=1}^{N} p^t_i = 1.$$

The intuition tells us that locally, there is an optimal probability distribution that would decrease faster the objective function f_θ. There is no reason that this distribution is uniform. To support this intuition, we argue that certain $g_{\theta^t,i}$ may be negligible compared to others at a specific stage of the optimization process, i.e. at a specific iteration t. Drawing such $g_{\theta^t,i}$ would have an almost negligible impact on minimizing the objective function. As a result, resources would be better utilized by computing the $g_{\theta^t,i}$ that significantly reduces the target function. However, the magnitude of the $g_{\theta^t,i}$ depends on the position of the parameter $theta^t$ in the search space; therefore, the probability distribution should be updated alongside the iterations.

One of the consequences of such non-uniform distribution over the $g_{\theta,i}$ is the construction of a gradient estimator that may be biased. This is problematic as many convergence guarantees [98] rely on the unbiasedness of the gradient estimator. To address this issue, we present two key points. First, the probability distribution I_t varies during the iterations of the learning process. The similarity between a $g_{\theta,i}$ and the exact gradient is not constant over the search space of θ. Therefore, our objective is to continuously update the probability associated with the terms of the gradient sum. Using the uniform distribution gives an unbiased estimator which gives convergence guarantees, so a proper update rule will smooth the probability associated to a backpropagation path $g_{\theta,i}$ over the iterations, i.e $\frac{1}{T}\sum_{t=1}^{T} p^t_i$ will tend toward $\frac{1}{N}$. In that case, the estimator becomes unbiased over the iterations. Secondly, and more importantly, we propose a modification to the computed gradient to ensure the unbiasedness of our novel estimator, regardless of the evolution of I_t:

Definition 10 (Normalization trick). *Let's define $g_\mathcal{I}$ the stochastic gradient estimator relative to $I_t \sim (p^t_1, \dots, p^t_N)$:*

$$g_{\mathcal{I}_t} = \begin{cases} \frac{1}{p_I^t} g_{\theta,I_t}, & \textit{if } p_I^t > 0 \\ 0, & \textit{otherwise.} \end{cases} \tag{IV.2}$$

The corrective term $\frac{1}{p_i^t}$ is introduced in order to preserve the unbiasedness of the gradient estimator, which is necessary to rely on convergence guarantees [98]. $g_{\mathcal{I}_t}$ is unbiased as long as none of the p_i^t values are equal to zero:

$$\forall t \leq T \quad \mathbb{E}[g_{\mathcal{I}_t}] = \sum_{i=1..N} p_i^t \times \frac{1}{p_i^t} \mathbb{E}[g_{\theta,i}] = \mathbb{E}[\nabla_\theta f].$$

This normalization trick evacuates all the possible issues about unbiasedness of a non uniform distribution over the backpropagation paths. Let's remember that our objective is to constrain memory usage and prevent overfitting without excessively lengthen training time. Consequently, seeking the optimal term $g_{\theta,i}$ of the gradient at every iteration is infeasible. Inspired by multi-armed bandits [151], we introduce a heuristic that balances the exploration of the best probability distribution with the utilization of the one established during exploration (a.k.a. exploitation).

In addition to reducing memory consumption, it might help the optimization process by avoiding local minima. In gradient descent a local minimum gives a zero gradient that might slow or even stuck the minimization of the objective function. However the gradient being equal to zero does not mean that all the $g_{\theta,i}$ are zero. Using one of them might help to avoid this unwanted scenario. An example is given on a toy function:

Example 5. *Let's consider the function $f_3 : \mathbb{R} \to \mathbb{R}$: $f_3(x) = x^2(2 + \cos(4x))$.*

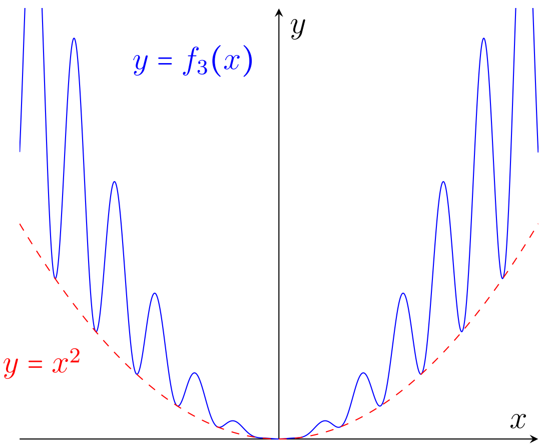


Figure IV.5.: Representation of $f_3(x) = x^2(2 + \cos(4x))$. This function has an infinite number of local minima, but its general trend follows x^2.

This function is chosen because it presents multiple local minima. The decomposition of the derivative of f_3 following the backpropagation paths of its LCG is given in Equation IV.3.

$$\frac{\partial f_3}{\partial x} = \underbrace{2x(2 + \cos(4x))}_{1^{st}\ component} \underbrace{-4x^2 \sin(4x)}_{2^{nd}\ component}. \tag{IV.3}$$

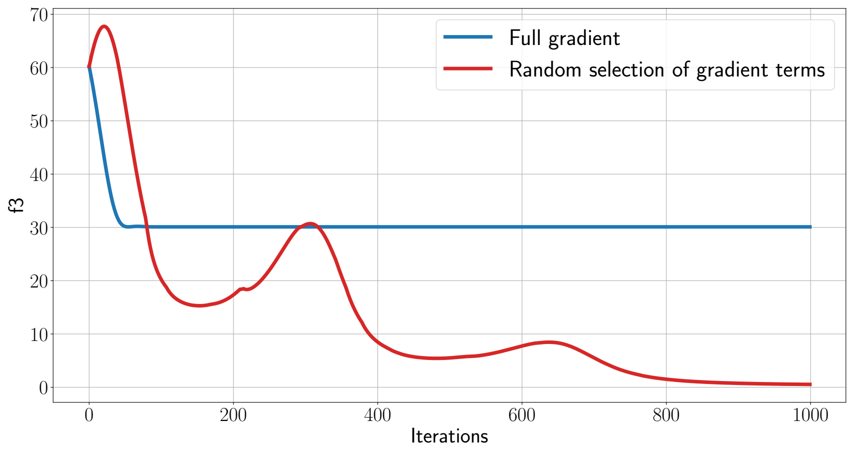


Figure IV.6.: Minimization of f_3 through SGD with Adam and its default values as optimizer. The starting point is $x = 5$. The blue curve represents the use of the full gradient from equation IV.3, which gets trapped in a local minimum. In contrast, the red curve represents the random selection of gradient terms during iterations, which allows for the avoidance of local minima and leads to a decrease in the target function.

If one employs the true gradient of f_3 and applies gradient descent with standard optimizers, it will undoubtedly become trapped in a local minimum. However, if one opts to utilize the first component of the gradient, it will reach the global minimum of f_3 at $x = 0$. This claim is supported by Figure IV.6

This example highlights the usefulness of approaches based on code stochasticity. From a practical standpoint, there are various techniques to access such decomposition and draw random paths in the backpropagation graph. We present a specific approach for neural networks and a generic one that is applicable to any type of gradient-based model.

IV.3.2. Projection matrices on Neural Networks

RAD [114] introduced a method to draw random paths on the backpropagation graph of neural networks. In order to generate random pathways in the backpropagation graph, RAD selects random projection matrices to sample some paths.

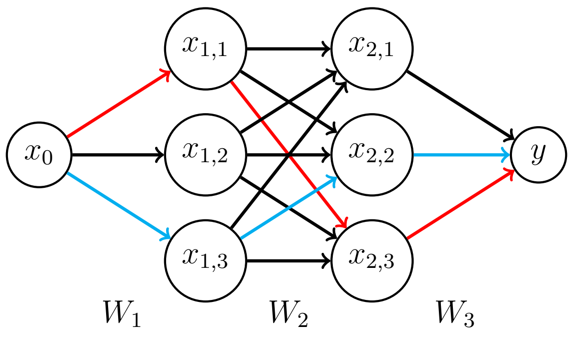


Figure IV.7.: Dense Layers with two selected paths. The graph being directed, selecting a forward path is equivalent to selecting a backward one.

Equation II.33 gives that the full gradient is the sum of the contributions of all the paths in the dense architecture. It is important to note that parameters are the edges of the Figure IV.7 while parameters are the parent nodes in an execution graph. We get $y = W_3 W_2 W_1 x_0$.

To get a path in a neural layer, one can multiply each layer by a $P_i = e_i \otimes e_i$ which is the outer product of standard basis vectors. Equation IV.4 (IV.5, respectively) denotes the contribution of the red (blue, respectively) path depicted in Figure IV.7 to the overall gradient of y.

$$\frac{\partial y}{\partial \theta} = \frac{\partial W_3 W_2 W_1 x_0}{\partial \theta} = \frac{\partial y}{\partial W_3} P3 \frac{\partial W_3}{\partial W_2} P3 \frac{\partial W_2}{\partial W_1} P1, \tag{IV.4}$$

$$\frac{\partial y}{\partial \theta} = \frac{\partial W_3 W_2 W_1 x_0}{\partial \theta} = \frac{\partial y}{\partial W_3} P2 \frac{\partial W_3}{\partial W_2} P2 \frac{\partial W_2}{\partial W_1} P3. \tag{IV.5}$$

Although P_i matrices are square, they can be represented as e_i vectors to save memory. The storing cost decreases quadratically thanks to this factorization.

However, for deep neural networks, a single path is insufficient to properly estimate gradients for efficient network updates due to their large size. To address this issue, a fraction $\frac{k}{d}$ of possible paths can be sampled by multiplying gradient layers with a random mask P_k^d:

$$P_k^d = \frac{d}{k} \sum_{s=1}^{k} P_s.$$

Each P_s is sampled from $\{P_i\}_{i \leq d}$. If the sampling is uniform then the estimator is unbiased thanks to Equation IV.6

$$\mathbb{E}[P_k^d] = \frac{d}{k} \mathbb{E}[\sum_{s=1}^{k} P_s] = \frac{d}{k} \frac{k}{d} I_d = I_d. \tag{IV.6}$$

Such masks can be stored in order to save memory with the decomposition $P = RR^T$. R is not a square matrix anymore but a $d \times k$ one, which reduces the memory impact as $k < d$. In order to produce such a mask R, one can compute a random matrix of independent Rademacher random variables. Such random variables sample uniformly random paths. In Section IV.4 we develop how to draw non uniformly paths in the execution graph.

The presented method, introduced by RAD, applies on neural networks but lacks generalization. In the following section we present how to access paths in the execution graph of an adjoint program thanks to a compilation trick.

IV.4. Beyond uniform distribution on backpropagation paths

IV.4.1. Selective Path Automatic Differentiation

The search for a good path is computationally demanding, as finding the exact best path implies evaluating all possible paths. An approximation is then to draw and evaluate a subset of path and choose the best path of this subset. But even in this case, repeating the procedure at every iteration will be costly. Remark that if a particular $g_{\theta,i}$ has the highest contribution to the gradient magnitude at a specific point θ^t, then it will also have the highest contribution in the surrounding parameter space as SGD is an iterative method. Hence, using this $g_{\theta,i}$ for a few iterations seems like a reasonable approximation. This approximation is even more reasonable when we assume that the difference between the $g_{\theta,i}$ values is independent of the batch being used. In other words, the more the observation batch is representative of the dataset, the better the approximation.

We introduce Selective Path Automatic Differentiation (SPAD), a new gradient estimator which deals with the trade-off of choosing the best component and keeping it for the next few iterations. We denote $\mathcal{P}$ the set of the LCG paths. We sample m random paths in the backpropagation graph, denoted as $P_m \subset \mathcal{P}$, and calculate the induced gradients $g_{\theta,i}$ (with $i \in [1..m]$ without loss of generality) restricted to these paths. Among these m paths and for the next $k_{\max}$ iterations, the one yielding the largest gradient $i_{\max}$ is associated to an almost one probability with keeping an $\epsilon > 0$ fraction of exploration for all the other paths (not restricted to the m ones).

To represent SPAD, we conveniently introduce the Almost-Dirac notation $D_i^{\epsilon}(j)$ in IV.7 below:

$$\forall \epsilon > 0; \forall i, j \leq N; \quad D_i^{\epsilon}(j) = (1-\epsilon)\delta_{j=i} + \frac{\epsilon}{N-1}\delta_{j\neq i}, \tag{IV.7}$$

for a given $i \leq N$, D_i^{ϵ} can be used as probability distribution over $[1..N]$ as $\sum_j D_i^{\epsilon}(j) = 1$. The probability distribution of SPAD described above is formalized by I_t from Equation IV.8.

$$\forall t \leq T \quad I_t \sim D^{\epsilon}_{i_{\max}^{qk_{\max}}}, \tag{IV.8}$$

with $i_{\max}^t = \underset{i \in P_m}{\arg\max} \|g_{\theta,i}\|$ and $t = qk_{\max} + r$ (euclidean division).

Algorithm 2 SPAD.

```
Require: $\mathcal{Z}$ (data)
Require: $\theta \in$ model (model parameters)
Require: epochs, iterations, $k_{\max}$, $m$, $\epsilon$ (hyper parameters)
    $epoch \leftarrow 0$
    while $epoch < epochs$ do
        $k \leftarrow 0$
        for i $\in$ iterations do
5:          batch = $\mathcal{Z}$[i]
            do forward on batch
            for $\theta \in$ rev(model) do
                if $k \equiv 0 \pmod{k_{\max}}$ then
                    draw m random paths
10:                 $i_{\max} = \arg\max_{j \leq m} \|g_{\theta,j}(batch)\|$
                    for $j \leq m$ do
                        $p_{\theta,j} = (1-\epsilon)\delta_{j=i_{\max}} + \frac{\epsilon}{N-1}\delta_{j\neq i_{\max}}$
                    end for
                end if
15:             draw $I$ according to $(p_{\theta,1}, \dots p_{\theta,m})$
                update $\theta$ with $\frac{1}{p_{\theta,1}} g_{\theta,I}(batch)$
            end for
            $k \leftarrow k + 1$
        end for
20:     $epoch \leftarrow epoch + 1$
    end while
    Return: $\theta$
```

SPAD is presented in Algorithm 2 and is particularly appealing as it avoids the need for a complete evaluation of the gradient, which is a resource-intensive process. Additionally, it does not require additional memory beyond storing the m random paths and their associated gradients. Notice that, as an implementation technique, checkpointing does not influence the gradient estimation itself, but rather the manner in which it is obtained. Consequently, all variations of checkpointing are compatible with SPAD. By choosing the largest gradient among the sampled paths, this approach has the potential to enhance the learning process, as the target loss is expected to decrease more rapidly compared to a random selection of the path. This heuristic introduces two new parameters, namely m and $k_{\max}$. However, there is a tradeoff to be made as increasing m may lead to a better gradient estimation but also slows down the learning process. With m and $k_{\max}$ both set to 1, SPAD reduces to the RAD method exclusively.

The parameter m represents the number of gradient paths that we select to determine the one with the largest contribution. It is desirable to have a large value of m in order to ensure that the strongest contribution is identified. The estimation of a maximum is always underestimated but it will not have a strong impact on our experiments thanks to quite large values of m. The parameter $k_{\max}$ determines the number of consecutive iterations that the chosen gradient path is used for. A larger value of $k_{\max}$ can be used if the chosen path is more effective. During the $k_{\max}$ iterations, the parameters corresponding to the unchosen paths are frozen. If $k_{\max}$ is set to a large value, it makes the method similar to freezing layers presented in [136]. If $k_{\max}$ is large, a large value for m is preferable to ensure that the chosen random path is carefully selected for multiple iterations. However, if $k_{\max}$ is small, we can tolerate a smaller m since the path selection has an impact on a limited number of iterations.

SPAD is an intermediate solution between RAD that does not try to determine the optimal choice of distribution and optimizations schemes that would need to duplicate the memory for the parameters, which would eliminate the benefits of our method. In the following part we show how to implement SPAD thanks to code stochasticity based on automatic differentiation, whatever the shape of the model to optimize, whereas RAD implementation based on matrix injections was only compatible with neural networks.

Compatibility with GCE

This chapter is presented as a bridge between Chapters I and III. It has clearly been explained how the compilation approach led to the SPAD gradient estimator, without any assumption on the form of the model. For the relationship with Chapter III, one has to note that SPAD and GCE are compatible.

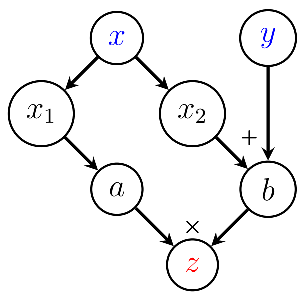
Both approaches are particularly suited for categorical models. While GCE has been designed to handle gradient updates of categorical parameters, SPAD enables a form of dropout for such models. The limited number of parameters involved in categorical models makes it impossible to use raw dropout, whereas SPAD only deactivates back-propagation nodes. Consequently, this regularization technique is now available for this set of white box models.

IV.4.2. From compilation to random paths: implementation generalization

To implement SPAD, we need a computational way to obtain the terms of the gradient from Equation II.33 written as a sum. It reduces to finding the backpropagation paths among the graph. The LCG being oriented, finding forward paths or backpropagation paths is the same problem. In a general context, and without any assumptions on the form of the LCG, we propose an alternative method for performing this task on any language suited for automatic differentiation satisfying the Static Single Assignment (SSA) and the Single Access (SA) properties.

Due to the SA property, the LCG of a program will contain tupling nodes, as highlighted in Example 6 below. They make possible the construction of programs using a variable multiple times by duplicating it. With the exception of these tupling nodes, there is only one edge that exits a node, which is a strict translation of the SA property on the LCG. Consequently, choosing a contribution to the gradient from Equation II.33 involves following the path from a parameter node to the output node and selecting one of the edges emanating from the encountered tupling nodes.

Example 6. *Let's consider the function $f_1(x, y) = e^x \times (x + y)$. To satisfy the SA property, since x is utilized twice in the program, its node is tupled, resulting in the LCG and the corresponding program as depicted in Figure IV.8:*



```
$x \leftarrow Param_0$
$y \leftarrow Param_1$
$x_1, x_2 \leftarrow x$
$a \leftarrow e^{x_1}$
$b \leftarrow x_2 + y$
$z \leftarrow a \times b$
$Return \quad z$
```

Figure IV.8.: (Left) SA-LCG of $f_1(x, y) = e^x(x + y)$. The node x is a tupling node. (Right) SSSA-SA version of the program relative to f_1.

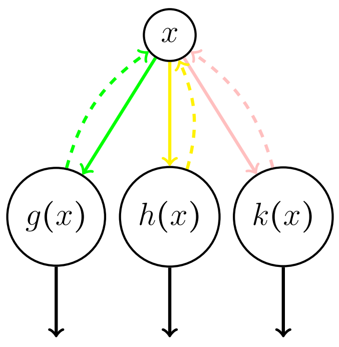
The tupling of variables in order to fulfill the SA property results in the gradient being expressed as a sum as proved in Equations IV.9. This is a key aspect of reverse mode automatic differentiation, also known as backpropagation. It is given by letting x be a parameter of f tupled in N variables $\{x_i\}_{i=1..N}$, Equation II.33 turns into:

$$\begin{aligned}\frac{\partial f}{\partial x} &= \sum_{x \longrightarrow z} \prod_{z_k \longrightarrow z_l} \frac{\partial z_l}{\partial z_k} \\ &= \sum_{i=1}^{N} \frac{\partial x_i}{\partial x} \sum_{x_i \longrightarrow z} \prod_{z'_k \longrightarrow z'_l} \frac{\partial z_{l'}}{\partial z_{k'}} \\ &= \sum_{i=1}^{N} 1.\frac{\partial f}{\partial x_i} = \sum_{i=1}^{N} \frac{\partial f}{\partial x_i}.\end{aligned} \tag{IV.9}$$

The chain rule of differentiation in reverse mode yields $\bar{x} = \frac{\partial f}{\partial x}$ called the adjoint of x, which is our objective. Figure IV.9 highlights how the SSA-SA property directly gives the gradient as a sum.

As previously framed, SPAD can be conceptualized as a form of dropout during back-propagation. By implementing SPAD independently of any specific model architecture, a generalized form of dropout can be introduced to a wider range of machine learning models. While dropout is a viable technique for deep learning models comprising numerous parameters without distinct significance for each individual one, it may not be suitable for smaller models.

The two approaches to obtain the gradient expressed as a sum, matrix injection or differentiation of SSA-SA languages, both rely on the multiple use of the parameters in the model implementation. If there is one and only one path from the parameter to the output node of the LCG, SPAD is pointless. Hopefully, this does not happen in many cases, as presented in the experiments Section IV.5.



$$\begin{aligned}
u &\longleftarrow g(x)\\
v &\longleftarrow h(x)\\
w &\longleftarrow k(x)\\
&\dots\\
\bar{x} &\longleftarrow 0\\
\bar{x} &\mathrel{+}= \bar{w}k'(x)\\
\bar{x} &\mathrel{+}= \bar{v}h'(x)\\
\bar{x} &\mathrel{+}= \bar{u}g'(x)
\end{aligned}$$

$$\begin{aligned}
x_1, x_2, x_3 &\longleftarrow x\\
u &\longleftarrow g(x_1)\\
v &\longleftarrow h(x_2)\\
w &\longleftarrow k(x_3)\\
&\dots\\
\bar{x}_3 &\mathrel{+}= \bar{w}k'(x_3)\\
\bar{x}_2 &\mathrel{+}= \bar{v}h'(x_2)\\
\bar{x}_1 &\mathrel{+}= \bar{u}g'(x_1)\\
\bar{x} &\longleftarrow \bar{x}_1 + \bar{x}_2 + \bar{x}_3
\end{aligned}$$

Figure IV.9.: (Left) Generic non SA LCG, with x being used three different times. The dashed lines represent backpropagation. g, h and k are arbitrary differentiable functions. (Center) SSA version of the derivative program. (Right) SSA-SA version of the derivative program.

IV.4.3. Adsl take on SPAD

To introduce this novel gradient estimator into our programming language Adsl, we extend its grammar by adding a new statement:

$$\langle v \longleftarrow \oplus_{SPAD} \quad tup \rangle$$

This innovative statement embodies the SPAD algorithm and incorporates the probability distribution over the elements of the tuple. Instead of calculating the exact gradient using the adjoints presented in Section I.4.5, one can use SPAD and substitute the differentiation of a tupling with this new statement:

$$\langle\overline{tup \leftarrow v}\rangle^{SPAD} = \langle\overline{v_1 \dots v_t \leftarrow v}\rangle^{SPAD} = \langle\overline{v}^{SPAD} \leftarrow \oplus_{SPAD} \quad \overline{v_1}\dots\overline{v_t}\rangle$$

Adsl has been presented a a language close by differentiation. As a result, it is expected that the adjoint of this new statement should be defined to align with this characteristic. However it is not immediately apparent what the statement representing $\overline{\langle v \longleftarrow \oplus_{SPAD} \quad tup\rangle}$ should be. We will not attempt to create such an adjoint, as it introduces more problems than it resolves. Consequently, when using SPAD in Adsl, we forget higher-order derivatives. However, we maintain that Adsl remains differentiable, as the $\oplus_{SPAD}$ cannot originate from a raw Adsl program but only from the differentiation of one.

IV.5. Experiments

We conducted experiments on two different types of tasks. Firstly, we applied our novel gradient estimator to a set of functions that are commonly used for evaluating optimization algorithms. These functions are not particularly suited for gradient descent as they present many local minima, but SPAD might solve this issue by following estimations of the gradient rather than the exact one. Evaluating SPAD on these functions further validates the usefulness of the implementation beyond the domain of neural networks. Secondly, we tested the estimator on the MNIST and the CIFAR10 datasets using standard deep architectures in order to compare our method to RAD. These experiments vary significantly in several aspects. Firstly, the data varies greatly, as the first search space is 2-dimensional while our dense architecture for MNIST presented in A.3.3 has over 410k parameters. Additionally, the minimum of the optimization functions is known, which is obviously not the case for neural networks. The diversity of tasks provides us with a deeper understanding of the implications of the proposed method.
Remember that the theoretical probability distribution given by SPAD is an Almost-Dirac on the largest gradient contribution. In practice we do not use this exact estimator $g_{\mathcal{I}}$ but simply $g_{\theta,I}$ by selecting the largest gradient contribution for $k_{\max}$ iterations. It removes the necessity of a random draw at each iteration for choosing the backpropagation path. Consequently a proper implementation of this version requires the storage of only two random paths as our goal is not to sort the gradient norms, but rather to find the arg max. Therefore, the memory usage is independent of the value of m. This alternative version of SPAD is depicted in Algorithm 3.

IV.5.1. Optimization functions

We evaluate the performance of the methods on four optimization functions by considering the ϵ-success rate from Definition 11, which measures the ratio of optimizations with different initializations that end less than ϵ away from the known global minimum for the given function.

Definition 11 (ϵ-success). $\boldsymbol{X}_T \in \mathcal{Z}$ *is an ϵ-success for the minimization of f if and only if* $f(\boldsymbol{X}_T) - \underset{X\in\mathcal{Z}}{\arg\min} f(X) < \epsilon$.

We conduct experiments using three different setups. The baseline method is SGD with the classical full gradient estimator, and we compare it against RAD and SPAD. The

Algorithm 3 SPAD in practice.

```
Require: $\mathcal{Z}$ (data)
Require: $\theta \in$ model (model parameters)
Require: epochs, iterations, $k_{\max}$, $m$, $\epsilon$ (hyper parameters)
    $epoch \leftarrow 0$
    while $epoch < epochs$ do
        $k \leftarrow 0$
        for i $\in$ iterations do
            batch = $\mathcal{Z}$[i]
            do forward on batch
            for $\theta \in$ rev(model) do
                if $k \equiv 0 \pmod{k_{\max}}$ then
                    draw m random paths
                    $i_{\max} = \arg\max_{j \leq m} \|g_{\theta,j}(batch)\|$
                end if
                update $\theta$ with $\frac{1}{p_{\theta,1}} g_{\theta,i_{\max}}(batch)$
            end for
            $k \leftarrow k + 1$
        end for
        $epoch \leftarrow epoch + 1$
    end while
    Return: $\theta$
```

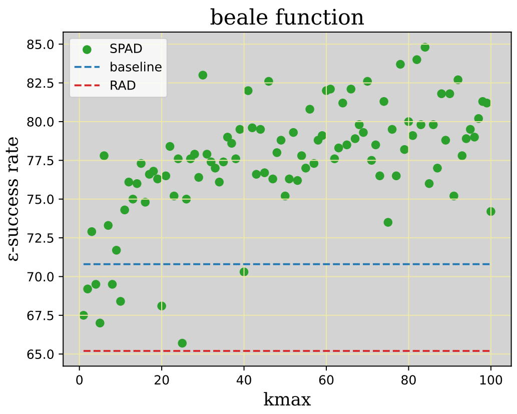
functions used for evaluation are described in A.3.2 and have a proven minimum thus the definition of the ϵ-success is possible. These functions cannot be written as neural networks, so we run our experiments on Envision, the domain specific language of Lokad, where the random paths can be drawn from the differentiation of the SSA-SA form of the language. Dropping out one of the few parameters of these functions is meaningless. We run 1000 experiments for each configuration with Adam [110] as optimizer on $T = 2000$ epochs. We tested $k_{\max} = 5$ and $k_{\max} = 50$, and report the ϵ-success rate in Table IV.1 (respectively IV.2) for $\epsilon = 0.05$ ($\epsilon = 0.01$ respectively). We have not tested multiple values of m because this parameter is strictly dependent on the function being used. We have chosen to select one branch from each tupling node in the backpropagation graph execution. For example, in function f_1 from Example 6, when the input x is used twice in the function, m is set to 2.

Function	baseline	RAD	SPAD$_{k_{\max}=5}$	SPAD$_{k_{\max}=50}$
Ackley	**12.2** %	0.1 %	2.1 %	1.6 %
Beale	70.8 %	65.2 %	67.0 %	**75.2** %
Levi	0.0 %	0.7 %	**2.2** %	1.4 %
Schaffer$_2$	**14.2** %	8.8 %	10.1 %	11.4 %

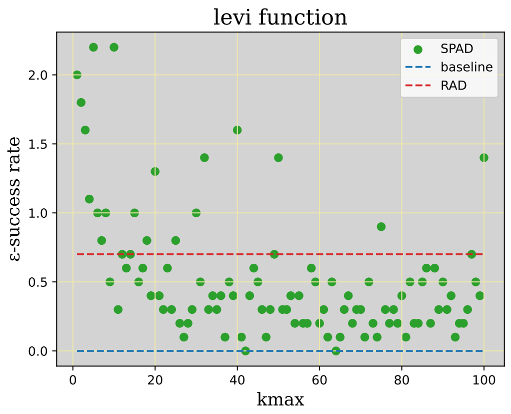
Table IV.1.: ϵ-success table with $\epsilon = 0.05$. In bold, the method with the higher ϵ-success rate for the corresponding function. On the beale and the levi functions, the best results are obtained with SPAD. The function definitions can be found in Appendix A.3.2.

Function	baseline	RAD	SPAD$_{k_{\max}=5}$	SPAD$_{k_{\max}=50}$
Ackley	**12.2** %	0.1 %	2.1 %	1.6 %
Beale	65.4 %	58.2 %	62.7 %	**70.5** %
Levi	0.0 %	0.2 %	**1.7** %	1.1 %
Schaffer$_2$	**13.9** %	8.8 %	10.0 %	11.3 %

Table IV.2.: ϵ-success table with $\epsilon = 0.01$. In bold, the method with the higher ϵ-success rate for the corresponding function. All the ϵ-success rate are lower than in Table IV.1 as ϵ is smaller.



(a) $k_{\max}$ impact on the ϵ-success of beale function minimization, with $\epsilon = 0.05$.



(b) $k_{\max}$ impact on the ϵ-success of levi function minimization, with $\epsilon = 0.05$.

Figure IV.10.: ϵ-success as a function of $k_{\max}$. On these graphs, the higher the better. Both experiments show an impact of $k_{\max}$ on the ϵ-success of the gradient descent. On Figure IV.10a on the beale function, a bigger $k_{\max}$ upgrades the optimization while it is the opposite on Figure IV.10b and the levi function. In both cases, the better results are obtained with a version of SPAD that outperforms the baseline and RAD.

As gradient methods are not well-suited on these functions, we did not expect good results. However, we can observe that when the gradient expression in the form of a sum is particularly adapted, as in the Beale function, SPAD yields better results. In more challenging cases, such as the Levi function, the baseline never manages to find a minimum, whereas using SPAD allows, albeit in a limited number of cases, to find the minimum.

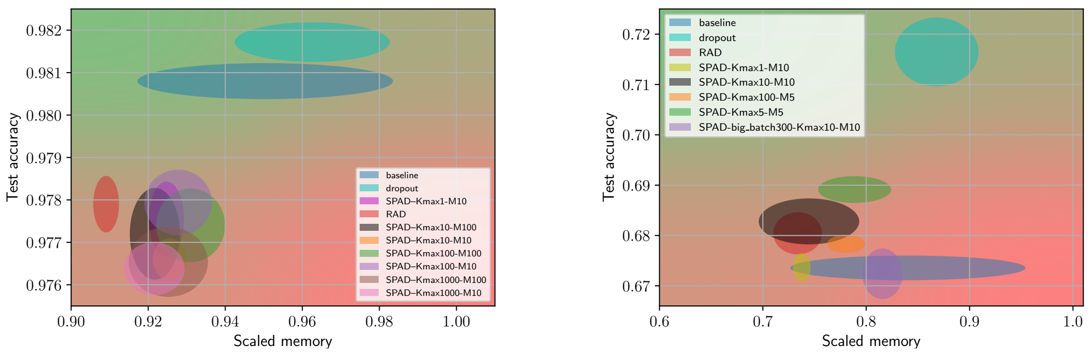
In Figure IV.10, we present the ϵ-success rate for varying values of $k_{\max}$ on the beale and the levi functions. In these examples the proposed method SPAD (with the appropriate $k_{\max}$) outperforms the baseline and RAD, which is very promising.

It also demonstrates that there is no universal optimal value of $k_{\max}$, as the performance seems to increase with $k_{\max}$ on the *beale* function but decreases on the *levi* function.

The choices we made to conduct these experiments are motivated by two observations. Firstly, The choice of the functions in this section is motivated by the fact that they employ several times their input variables. As highlighted in Section IV.4.2, it is necessary to use SPAD. Secondly, although SPAD is promoted as a way to reduce overfitting, this concept is not relevant in optimization problems where the goal is to find the optimal parameters that maximize the objective function, without considering factors such as the model's generalization capabilities.

IV.5.2. Deep learning

We conducted experiments on the MNIST and CIFAR10 datasets and compared SPAD with RAD, the standard stochastic gradient estimator and the dropout technique. We use the same experimental framework described in [114], which does not include any early stopping. Doing so would increase the memory requirements that we want to avoid. However such a framework may lead to overfitting, which we aim to mitigate. The objective of our approach is to maintain the learning quality while reducing the memory peak compared to traditional SGD.
Because of the large number of parameters in the networks used, drawing a single path in the backpropagation graph would result in negligible updates. Instead, we think in terms of the fraction of the path to be drawn and, as a result, we conducted experiments in which 10% of the network is updated at each iteration. To rephrase it, we draw m sets of random paths, with each set covering 10% of the network. In contrast, the theoretical version of SPAD generates m random paths, with each path covering $\frac{1}{N}$% of the model. We applied the same proportion (i.e. 10%) when executing dropout runs.



(a) SmallFCNet on MNIST. The network is made of 4 linear layers with Rectified Linear Unit activation.

(b) SmallConvNet on CIFAR10. The network is made of 4 convolutional layers with Rectified Linear Unit activation.

Figure IV.11.: Accuracy on test versus memory peak tradeoff. The displayed memory is a fraction of the biggest memory peak of the baseline, the same is used for every run. The superior results are located in the upper left quadrant of the graph, indicated by the green color.

Figure IV.11 displays the two metrics we aim to optimize, i.e. the final accuracy on the testing dataset and the memory peak in % required by the training. The objective is to get higher accuracy on testing with the lowest memory consumption, i.e. ending in the green zone. A run is considered as strictly better than another if it reaches higher accuracy with less memory. Otherwise one cannot rank two runs. On these examples, the many variants of SPAD are competitive with the baseline and RAD, and it achieves strictly superior results on CIFAR10. With regards to the MNIST dataset, as shown in Figure IV.11a, none of the methods outperform the baseline, although the differences are minimal, as every model achieves over 97% accuracy. The least accurate results occur when $k_{\max} = 1000$. This outcome is reasonable since the selected paths may

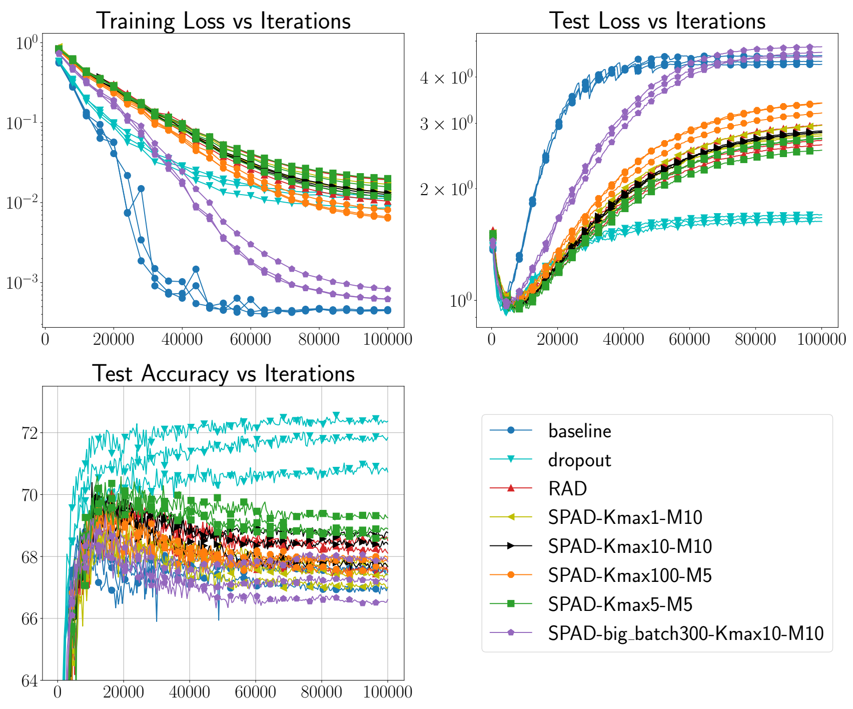


Figure IV.12.: Learning curves of the SmallConvNet on CIFAR10. The baseline model exhibits rapid performance improvement on the training dataset, while its performance on the testing dataset deteriorates just as quickly. This behavior is characteristic of overfitting, whereas the various versions of SPAD effectively mitigate this undesired decrease in generalization.

be utilized for an excessive number of iterations and might lose relevance at a specific stage. On IV.11b which concerns the CIFAR10 dataset, some versions of SPAD like ($k_{\max} = m = 10$) are strictly better than the baseline. Note that all the runs share the same neural architectures, which is a Fully connected network on MNIST and a convolutional one on CIFAR10. More details are given in A.3.3.
Concerning overfitting, detailed results on the CIFAR10 dataset are presented in Figure IV.12, while more details on the MNIST dataset are given in the appendices. They tend to confirm that our method effectively reduces overfitting compared to the baseline. While the training loss of the baseline quickly decreases during the first iterations, its test loss quickly increases. On the contrary, SPAD slowly decreases its loss on the training dataset and its testing loss increases slowly compared to the baseline. This observation highlights the similarities between the process of randomly drawing paths during backpropagation and the dropout technique. Turning off a fraction of the network, on the forward pass for dropout and on the backpropagation for SPAD, tends to reduce overfitting. The dropout runs reach the highest test accuracy, but with a significant memory consumption.
This approach effectively mitigates overfitting, as the testing loss increases at a much slower rate compared to other heuristics in Figures A.6 and A.7 from the appendices. Incorporating random matrix injections could prove highly beneficial for such learning techniques.
The primary objective of SPAD is to minimize memory consumption. From the perspective of a fixed memory budget, employing SPAD liberates resources that can be reallocated to increase the batch size, for instance. We evaluated this hypothesis by employing the SPAD method, utilizing a batch size twice as large as that in the other

experiments denoted by *big batch* in the legend of Figure IV.11b. While this approach led to increased memory consumption, it also resulted in increasing overfitting, exhibiting behavior akin to the baseline in both training and testing data sets. This observation is consistent with the findings of [152], which assert that large batch training methods are more prone to overfitting compared to the same network trained using smaller batch sizes.

Other heuristics We have tested other probability distribution construction over the $g_{\theta,i}$ like

$$I_t \sim D^{\epsilon}_{s_{qk_{\max}}} \quad \text{with } s_t = \underset{p \in P_m}{\arg\min} \|\nabla f_\theta - g_{\theta,p}\| \tag{IV.10}$$

Nevertheless, none of the other tested methods yielded superior results compared to SPAD. Furthermore, SPAD exhibits the lowest memory consumption, as it eliminates the need to compute the full gradient even once, in contrast to the heuristic presented in Equation IV.10.

Implementation trick

The following paragraph is very Pytorch-specific.

The deep learning experiments were performed using Pytorch. The main challenge was persisting tensors from the backward pass to the forward pass. The random paths P_k^d to be selected for multiple iterations were chosen during the gradient calculation in the backward pass. Although intermediate tensors can be saved using the $save\_for\_backward$[1] function, there is no similar function for saving tensors from the forward pass to the backward pass. To address this issue, we passed the factorized version of P_k^d as a ghost input to the forward pass, manually updated its version in each of the k_{max} iterations into the corresponding gradient, and finally replaced P_k^d with the value artificially stored in its gradient.

Conclusion and perspectives

From the perspective of deep learning, SPAD can be regarded as a combination of dropout and layer freezing within a neural network. By drawing backpropagation paths, our method proposes a similar technique to dropout for any gradient based model. It is based on reverse mode automatic differentiation of Static Single Assignment - Single Access languages.

Moreover, our main idea is to draw more frequent examples that have a bigger impact on the loss minimization. Concerning this code's stochasticity, our result shows the advantages of a non uniform probability distribution. This is aligned with multiple works [153, 154] that use non-uniform distributions on the observations and outperforms the uniform one.

Table IV.3 summarizes the construction of gradient stochasticity based on the chosen stochasticity. The sampling process can be conducted at the observation or code level, with uniform or non-uniform distribution.

An interesting future work would be about non-uniform distributions on the observations and on the code, which could hopefully get better learning results without increasing

[1] $torch.autograd.function.FunctionCtx.save\_for\_backward$

Granularity	GD	SGD	RAD	SPAD	[153]
Observations	**NO**	Uniform	Uniform	Uniform	Non-Uniform
Code	**NO**	**NO**	Uniform	Non-Uniform	**NO**

Table IV.3.: Small review of the stochasticity origin of gradient estimators.

the training memory needs. Such direction would help parameters updates on embedded artificial intelligence, which would open many industrial applications, like embedded machine learning on devices with constrained computational resources.
All of these advancements promote the implementation of embedded machine learning on devices with constrained computational resources, thereby enhancing their utility and efficiency.

Conclusion

Summary

In the introduction we presented the context of this PhD: this PhD was funded and initiated by the french company Lokad in order to unlock differentiable programming on its own domain specific language Envision.
In Chapter I we have explored the topic of differentiating relational queries. We started by discussing differentiation techniques and automatic differentiation, highlighting their significance in the context of relational data. We also examined existing automatic differentiation systems and their applicability to relational queries. Next, we delved into the specifics of relational queries, considering their industrial context and the unique characteristics of relational data. We explored the concepts of relational algebra and queries, emphasizing the need for automatic differentiation in this domain. We surveyed existing approaches to automatic differentiation on relational queries. We then introduced the concept of differentiable programming on relational queries. We defined relevant notations and discussed how a loss query can be expressed both relationally and mathematically. We introduced the TOTAL JOIN operator and PolyStar as key components in differentiable programming on relational queries. To facilitate the implementation and application of differentiable programming on relational queries, we introduced a dedicated programming language called Adsl. We presented the language and its features, including Static Single Assignment form and Single Access. We also discussed the equivalence of Adsl and Wengert lists and the induced automatic differentiation capabilities of Adsl. Furthermore, we showcased the application of differentiable programming on relational queries through Envision, a domain-specific language. We highlighted the prominence of differentiable programming as a first-class citizen in Envision and demonstrated its utility through the concept of relational linear regression. Additionally, we presented a real-world example involving retail forecasting and discussed the mathematical insights gained from the application.
In Chapter II we have presented SGD and its applications in machine learning optimization. We discussed the convergence properties of SGD and its benefits in terms of computational efficiency. We also examined the use of adaptive optimizers, such as Adam, in conjunction with SGD for improved convergence and generalization. Furthermore, we discussed the application of stochasticity and gradient descent techniques to relational data, specifically through PolyStar. These findings provide a foundation for future research in optimizing and exploring complex relational models.
In Chapter III we addressed the challenges of incorporating categorical features in machine learning models. We discussed the limitations of traditional one-hot-encoding methods and their impact on gradient descent optimization. To overcome these issues, we proposed a novel solution called the Gradient Estimator for Categorical Features. We provided a comprehensive definition of GCE and proved its unbiasedness, demonstrating its effectiveness in accurately estimating gradients for categorical features. We applied GCE to relational linear regression and conducted experiments using both deep learn-

ing models and public datasets. The experimental results showed the superiority of our approach in handling categorical features, both in terms of model performance and convergence speed. We also validated the effectiveness of GCE in real-world scenarios, showcasing its applicability in production environments. Furthermore, we explored the initialization of categorical models, presenting a two-feature example and generalizing it to singular value decomposition.
In Chapter IV we investigated the impact of gradient code stochasticity on memory consumption, overfitting, and optimization performance in gradient-based models. We first addressed the issue of memory consumption and presented checkpointing techniques to alleviate the burden of storing intermediate activations during reverse mode automatic differentiation. We presented the integration of checkpointing in Adsl and discussed its benefits in terms of memory efficiency. Next, we focused on combating overfitting, a common challenge in deep learning. We explored the dropout technique as a regularization method to prevent overfitting and improve generalization performance. We then delved into the concept of sample random paths and their implications for gradient-based optimization. We discussed gradient code stochasticity and its role in diversifying the training process by sampling different paths in the backpropagation algorithm. Furthermore, we introduced the Selective Path Automatic Differentiation technique, which allows us to move beyond the uniform distribution of backpropagation paths. We presented the implementation details of SPAD and highlighted its potential for enhancing optimization performance and generalization. The experimental results showcased the effectiveness of the proposed techniques. We evaluated various optimization functions and conducted experiments using deep learning models. The results demonstrated the benefits of memory-efficient strategies, dropout regularization, and sample random paths in improving training efficiency, reducing overfitting, and achieving better generalization.

Perspectives

I strongly think that the work presented in this PhD holds great promise for future developments in several areas.
Firstly, beyond the strong theoretical set up founded by the introduction of Adsl, the introduction of differentiable programming to Envision has demonstrated its practical feasibility and usefulness on relational programming languages. This should encourage future work in similar directions, such as extending this approach to SQL, which is the most widely used relational language. Undoubtedly, addressing this issue may require overcoming additional challenges, as the PolyStar could be more difficult to define in this language due to the numerous intermediate tables present in its queries. Nonetheless, the benefits provided by our implementation demonstrate that the effort is well worth it. The extension of differentiable programming to widely used relational programming languages will enable domain experts to create bespoke machine learning models more effectively. By facilitating their direct involvement in the development of forecasting engines, they will transition from passive observers to active contributors in the process.
Secondly, categorical data has been relatively neglected by the machine learning community, and we hope that the proposed GCE approach will encourage greater use of such data and improve the effectiveness of machine learning tasks involving it. Additionally, we anticipate that the adoption of more categorical models, such as the introduced relational linear regression, will continue to increase due to their high level of interpretability.

We believe that interpretability will become a critical aspect for future machine learning developments in domains that have significant impact on our society and its economy like health or supply chain.
Thirdly, SPAD investigates an intriguing aspect of gradient stochasticity rooted in the code itself, which, to our knowledge, has been largely underestimated. Exploring non-uniform probability distributions on backpropagation paths raises questions about the potential for coupling it with non-uniform distributions on observations, which has already been addressed independently.
These future research directions can be regarded as independent, but they all pertain to differentiable programming, which, in my opinion, should be considered holistically. The introduction of SPAD, based on compilation design choices made during the construction of Adsl, supports this perspective, rather than addressing each problem individually without considering the big picture.

Acknowledgements

I would like to express my gratitude to my thesis advisors, Thierry Paquet and Maxime Berar, for their guidance throughout my PhD journey. Their insightful feedback has been instrumental in shaping the direction and the quality of this work.
I am also grateful to Victor Nicollet, CTO of Lokad, who provided invaluable feedback and guidance from the industry side. I wish to every CIFRE PhD student to have such an advisor in his company.
I also thank Joannes Vermorel, CEO of Lokad, whose constant stimulation and vision on differentiable programming played a crucial role in this work.
I would like to extend my thanks to Vincent Berthoux for his contribution to the implementation of differentiable programming in Envision, and to Gaëtan Delétoille and Antonio Cifonelli for their theoretical insights and discussions on various topics.
I also want to express my gratitude to Kevin Baumann and Baptiste Miceli for the common work on the Celio dataset, as well as to Estelle Dewost and the administrative team for their support in managing the administrative aspects of this PhD.
I am grateful to Dr. Boldrini for the invaluable insights and guidance he provided me. He has been a great example for me.
I would like to acknowledge all the people who have contributed to this work in one way or another, and to my family and friends for their unwavering support and encouragement. A special thanks to Auriane Charlot that supported (in English but also in French) me throughout this journey.

Finally, I would like to express my gratitude to all the Lokad employees who have contributed to making it a great place to work, it really mattered in the successful completion of this three-year PhD.

Bibliography

[1] A. Baydin, B. Pearlmutter, A. Radul, and J. Siskind, "Automatic differentiation in machine learning: A survey," *Journal of Machine Learning Research*, vol. 18, pp. 1–43, 04 2018.

[2] G. Deletoille, *Gestion de stock sous contrainte de quantité minimale de commande multi-références*. PhD thesis, Litis, 2022. Thèse de doctorat dirigée par Adam, Sébastien Informatique Normandie 2022.

[3] M. Durut and F. Rossi, "Communication challenges in cloud k-means," in *Proceedings of XIXth European Symposium on Artificial Neural Networks (ESANN 2011)*, (Bruges (Belgium)), pp. 387–392, 4 2011.

[4] A. Aziz, E. Kosasih, R.-R. Griffiths, and A. Brintrup, "Data considerations in graph representation learning for supply chain networks," *International Conference for Machine Learning (ICML) workshop on ML4Data*, 07 2021.

[5] M. Durut, B. Patra, and F. Rossi, "A discussion on parallelization schemes for stochastic vector quantization algorithms," in *Proceedings of the XXth European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2012)*, (Bruges, Belgique), pp. 477–482, 4 2012.

[6] A. Kadra, M. T. Lindauer, F. Hutter, and J. Grabocka, "Well-tuned simple nets excel on tabular datasets," in *NeurIPS*, 2021.

[7] Y. Gorishniy, I. Rubachev, V. Khrulkov, and A. Babenko, "Revisiting deep learning models for tabular data," in *Advances in Neural Information Processing Systems* (A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, eds.), 2021.

[8] W. Fu and T. Menzies, "Easy over hard: a case study on deep learning," *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[9] B. Eck, D. Kabakci-Zorlu, Y. Chen, F. Savard, and X. Bao, "A monitoring framework for deployed machine learning models with supply chain examples," in *IBM at Big Data 2022*, pp. 2231–2238, 12 2022.

[10] C. Deng, X. Ji, C. Rainey, J. Zhang, and W. Lu, "Integrating machine learning with human knowledge," *iScience*, vol. 23, no. 11, p. 101656, 2020.

[11] M. E. Schüle, F. Simonis, T. Heyenbrock, A. Kemper, S. Günnemann, and T. Neumann, "In-database machine learning: Gradient descent and tensor algebra for main memory database systems," in *BTW*, 2019.

[12] C. Rudin, "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead," *Nature Machine Intelligence*, vol. 1, pp. 206–215, 05 2019.

[13] M. Schleich, D. Olteanu, M. Abo-Khamis, H. Q. Ngo, and X. Nguyen, "Learning models over relational data: A brief tutorial," in *Scalable Uncertainty Management* (N. Ben Amor, B. Quost, and M. Theobald, eds.), pp. 423–432, Springer International Publishing, 2019.

[14] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley, "Differentiable programming for image processing and deep learning in halide," *ACM Transactions on Graphics (TOG)*, vol. 37, pp. 1 – 13, 2018.

[15] M. Abadi and G. Plotkin, "A simple differentiable programming language," *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1–28, 12 2019.

[16] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[17] I. Newton, "De quadrutura curvarum," *Opticks*, 1704.

[18] Leibnitz, "A new method for maxima and minima as well as tangents, which is impeded neither by fractional nor irrational quantitites, and a remarkable type of calculus for this," *Acta Eruditorum*, 1664.

[19] R. Wengert, "A simple automatic derivative evaluation program," *Commun. ACM*, vol. 7, pp. 463–464, 08 1964.

[20] A. G. Baydin, K. Cranmer, M. Feickert, L. Gray, L. Heinrich, A. Held, A. Melo, M. Neubauer, J. Pearkes, N. Simpson, N. Smith, G. Stark, S. Thais, V. Vassilev, and G. Watts, "Differentiable programming in high-energy physics," in *Snowmass 2021 Letters of Interest (LOI), Division of Particles and Fields (DPF), American Physical Society*, 2020.

[21] R. Roussel, A. Edelen, D. Ratner, K. Dubey, J. P. Gonzalez-Aguilera, Y. K. Kim, and N. Kuklev, "Differentiable preisach modeling for characterization and optimization of particle accelerator systems with hysteresis," *Phys. Rev. Lett.*, vol. 128, p. 204801, May 2022.

[22] F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and J. Z. Kolter, "End-to-end differentiable physics for learning and control," in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, 2018.

[23] F. Kamiran and T. Calders, "Data pre-processing techniques for classification without discrimination," *Knowledge and Information Systems*, vol. 33, 10 2011.

[24] Y. Volin and G. Ostrovskii, "Automatic computation of derivatives with the use of the multilevel differentiating technique—1. algorithmic basis," *Computers and Mathematics with Applications*, vol. 11, no. 11, pp. 1099–1114, 1985.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

[26] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms.," *CoRR*, vol. abs/1707.06347, 2017.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[28] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in *Advances in Neural Information Processing Systems* (Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, eds.), vol. 27, Curran Associates, Inc., 2014.

[29] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NeurIPS’14, (Cambridge, MA, USA), p. 3104–3112, MIT Press, 2014.

[30] D. Elizondo, B. Cappelaere, and C. Faure, “Automatic versus manual model differentiation to compute sensitivities and solve non-linear inverse problems,” *Computers and Geosciences*, vol. 28, pp. 309–326, 04 2002.

[31] L. Feigenbaum, “Brook taylor and the method of increments,” *Archive for History of Exact Sciences*, vol. 34, no. 1/2, pp. 1–140, 1985.

[32] R. Chartrand, “Numerical differentiation of noisy, nonsmooth data,” *ISRN Appl. Math.*, 01 2011.

[33] A. Griewank, “On automatic differentiation,” *Mathematical Programming: Recent Developments and Applications*, pp. 83–108, 01 1989.

[34] S. Laue, “On the equivalence of forward mode automatic differentiation and symbolic differentiation,” *CoRR*, vol. abs/1904.02990, 2019.

[35] A. Griewank and A. Walther, “Evaluating derivatives - principles and techniques of algorithmic differentiation, second edition,” in *Frontiers in applied mathematics*, 2000.

[36] W. Lee, H. Yu, X. Rival, and H. Yang, “On correctness of automatic differentiation for non-differentiable functions,” in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), vol. 33, pp. 6719–6730, Curran Associates, Inc., 2020.

[37] L. Hascoët and V. Pascual, “The tapenade automatic differentiation tool: Principles, model, and specification,” *ACM Trans. Math. Softw.*, vol. 39, pp. 20:1–20:43, 2013.

[38] M. Innes, “Don’t unroll adjoint : Differentiating ssa-form programs,” *ArXiv*, vol. abs/1810.07951, 2018.

[39] B. van Merrienboer, O. Breuleux, A. Bergeron, and P. Lamblin, “Automatic differentiation in ml: Where we are and where we should be going,” in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, Curran Associates, Inc., 2018.

[40] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library.," in *NeurIPS*, pp. 8024–8035, 2019.

[41] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[42] R. Fourer, D. Gay, and B. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, vol. 36. Duxbury Press, 01 2002.

[43] C. H. Bischof, L. Roh, and A. J. Mauer-Oats, "Adic: An extensible automatic differentiation tool for ansi-c," *Softw. Pract. Exper.*, vol. 27, p. 1427–1456, dec 1997.

[44] G. Walther, "Getting started with adol-c," *Combinatorial Scientific Computing*, pp. 181–202, 01 2009.

[45] B. M. Bell and J. V. Burke, "Algorithmic differentiation of implicit functions and optimal values," in *Advances in Automatic Differentiation* (C. H. Bischof, H. M. Bücker, P. Hovland, U. Naumann, and J. Utke, eds.), (Berlin, Heidelberg), pp. 67–77, Springer Berlin Heidelberg, 2008.

[46] C. Bendtsen, "Fadbad, a flexible c++ package for automatic differentiation - using the forward and backward method." http://www.imm.dtu.dk/fadbad.html, 1996.

[47] J.-F. Ostiguy and L. Michelotti, "Mxyzptlk: An efficient, native c++ differentiation engine," in *2007 IEEE Particle Accelerator Conference (PAC)*, pp. 3489–3491, 2007.

[48] A. Shtof, A. Agathos, Y. Gingold, A. Shamir, and D. Cohen-Or, "Geosemantic snapping for sketch-based modeling," *Computer Graphics Forum*, vol. 32, 05 2013.

[49] A. G. Baydin, B. A. Pearlmutter, and J. Siskind, "Diffsharp: An ad library for .net languages," *ArXiv*, vol. abs/1611.03423, 2016.

[50] C. Bischof, P. Khademi, A. Carle, and A. Mauer, "Adifor 2.0: Automatic differentiation of fortran 77 programs," *Computing in Science and Engineering*, vol. iii, pp. 18–32, sep 1996.

[51] U. Naumann and J. Riehme, "Computing adjoints with the nagware fortran 95 compiler," in *Automatic Differentiation: Applications, Theory, and Implementations* (M. Bücker, G. Corliss, U. Naumann, P. Hovland, and B. Norris, eds.), (Berlin, Heidelberg), pp. 159–169, Springer Berlin Heidelberg, 2006.

[52] R. Giering and T. Kaminski, "Recipes for adjoint code construction," *ACM Trans. Math. Softw.*, vol. 24, p. 437–474, dec 1998.

[53] C. Bischof, G. Corliss, A. Griewank, M. Berz, K. Makino, K. Shamseddine, G. Hoffstatter, and W. Wan, *COSY INFINITY and Its Applications in Nonlinear Dynamics.* Computational Differentiation: Techniques, Applications, and Tools, 12 1997.

[54] E. I. Sluşanschi and V. Dumitrel, "Adijac – automatic differentiation of java classfiles," *ACM Trans. Math. Softw.*, vol. 43, sep 2016.

[55] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in Julia," *arXiv:1607.07892 [cs.MS]*, 2016.

[56] W. Moses and V. Churavy, "Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients," in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), vol. 33, pp. 12472–12485, 2020.

[57] J. Willkomm and A. Vehreschild, "The adimat handbook," *URL http://adimat. sc. informatik. tu-darmstadt. de/doc*, 2013.

[58] S. M. Rump, *INTLAB—interval laboratory.* Springer, 1999.

[59] S. A. Forth, "An efficient overloaded implementation of forward mode automatic differentiation in matlab," *ACM Trans. Math. Softw.*, vol. 32, p. 195–222, June 2006.

[60] D. Maclaurin, *Modeling, inference and optimization with composable differentiable procedures.* Harvard University, 2016.

[61] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," in *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NeurIPS)*, vol. 5, pp. 1–6, 2015.

[62] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs." http://github.com/google/jax, 2018.

[63] B. van Merriënboer, A. B. Wiltschko, and D. Moldovan, "Tangent: Automatic differentiation using source code transformation in python," *arXiv preprint arXiv:1711.02712*, 2017.

[64] S. Tremaine, "Structure and interpretation of classical mechanics," *Physics Today*, vol. 55, no. 1, pp. 54–56, 2002.

[65] B. A. Pearlmutter and J. M. Siskind, "Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator," *ACM Transactions on Programming Languages and Systems*, vol. 30, no. 2, pp. 7.1–7.36, 2008.

[66] Y. Hu, L. Anderson, T.-M. Li, Q. Sun, N. Carr, J. Ragan-Kelley, and F. Durand, "Difftaichi: Differentiable programming for physical simulation," *ICLR*, 2020.

[67] R. Wei, D. Zheng, M. Rasi, and B. Chrzaszcz, "Differentiable programming manifesto." https://github.com/apple/swift/blob/main/docs/DifferentiableProgramming.md, 2020.

[68] C. Mak and C. Ong, "A differential-form pullback programming language for higher-order reverse-mode automatic differentiation," *LAFI*, 2021.

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

[70] E. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, pp. 377–387, 01 1970.

[71] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, pp. 56 – 61, 2010.

[72] M. E. Schüle, M. Bungeroth, D. Vorona, A. Kemper, S. Günnemann, and T. Neumann, "ML2SQL - compiling a declarative machine learning language to SQL and python," in *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pp. 562–565, 2019.

[73] S. Kläbe, S. Hagedorn, and K.-U. Sattler, "Exploration of approaches for in-database ml," in *International Conference on Extending Database Technology*, 2023.

[74] K. Karanasos, M. Interlandi, F. Psallidas, R. Sen, K. Park, I. Popivanov, D. Xin, S. Nakandala, S. Krishnan, M. Weimer, Y. Yu, R. Ramakrishnan, and C. Curino, "Extending relational query processing with ML inference," in *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*, www.cidrdb.org, 2020.

[75] M. Schüle, H. Lang, M. Springer, A. Kemper, T. Neumann, and S. Gunnemann, "In-database machine learning with sql on gpus," in *33rd International Conference on Scientific and Statistical Database Management*, SSDBM 2021, (New York, NY, USA), p. 25–36, Association for Computing Machinery, 2021.

[76] W. Wang, M. Zhang, G. Chen, H. V. Jagadish, B. C. Ooi, and K.-L. Tan, "Database meets deep learning: Challenges and opportunities," *SIGMOD Rec.*, vol. 45, p. 17–22, sep 2016.

[77] P. Peseux, "Differentiating relational queries," in *PhD@VLDB*, 2021.

[78] S. Dasgupta, "Learning polytrees," *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, 01 2013.

[79] R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser, "Optimal broadcast and summation in the logp model," in *ACM Symposium on Parallelism in Algorithms and Architectures*, 1993.

[80] L. Libkin and L. Wong, "On the power of aggregation in relational query languages," in *Database Programming Languages* (S. Cluet and R. Hull, eds.), pp. 260–280, Springer Berlin Heidelberg, 1998.

[81] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[82] E. S. Gardner, "Exponential smoothing: The state of the art," *Journal of Forecasting*, vol. 4, pp. 1–28, 1985.

[83] A. Baydin, B. Pearlmutter, D. Syme, F. Wood, and P. Torr, "Gradients without backpropagation," *arXiv preprint*, vol. arXiv, 02 2022.

[84] G. Belouze, "Optimization without backpropagation," *arXiv preprint*, vol. arXiv, 09 2022.

[85] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, 2004.

[86] R. Sutton, D. Mcallester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Adv. Neural Inf. Process. Syst*, vol. 12, 02 2000.

[87] A. Mari, T. R. Bromley, and N. Killoran, "Estimating the gradient and higher-order derivatives on quantum hardware," *Phys. Rev. A*, vol. 103, p. 012405, Jan 2021.

[88] M. Cerezo and P. J. Coles, "Higher order derivatives of quantum neural networks with barren plateaus," *Quantum Science and Technology*, vol. 6, p. 035006, jun 2021.

[89] I. Trummer and C. Koch, "Multi-objective parametric query optimization," *ACM SIGMOD Record*, vol. 45, pp. 24–31, 06 2016.

[90] F. Parsana and K. Atkotiya, "Study and analyzing an effective query optimization, strategies and algorithms in database systems.," in *Conference: Emerging trends in computer science and information technology*, 08 2019.

[91] C. of Chicago, "Taxi trips of chicago." https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew, 2021.

[92] S. Hosseini and D. Ivanov, "Bayesian networks for supply chain risk, resilience and ripple effect analysis: A literature review," *Expert Systems with Applications*, vol. 161, p. 113649, 2020.

[93] C. Di Domenico, Y. Ouzrout, M. M. Savinno, and A. Bouras, "Supply chain management analysis: A simulation approach of the value chain operations reference model (vcor)," in *Advances in Production Management Systems* (J. Olhager and F. Persson, eds.), (Boston, MA), pp. 257–264, Springer US, 2007.

[94] D. Ivanov, S. Hosseini, and A. Dolgui, "Review of quantitative methods for supply chain resilience analysis," *Transportation Research Part E Logistics and Transportation Review*, vol. 125, pp. 285–307, 03 2019.

[95] Z. Benomar, "Reproducible parallel stochastic gradient descent for very high dimensional data." https://blog.lokad.com/pdf/reproducible-parallel-sgd-ziyad-benomar-2022-09.pdf, 2022.

[96] S. Taylor and B. Letham, "Forecasting at scale," *The American Statistician*, vol. 72, 09 2017.

[97] H. Robbins and S. Monro, "A Stochastic Approximation Method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400 – 407, 1951.

[98] A. Défossez, L. Bottou, F. Bach, and N. Usunier, "A simple convergence proof of adam and adagrad," *Transactions on Machine Learning Research*, 2022.

[99] A. Amini, A. Soleimany, S. Karaman, and D. Rus, "Spatial uncertainty sampling for end-to-end control," *CoRR*, vol. abs/1805.04829, 2018.

[100] G. Garrigos and R. M. Gower, "Handbook of convergence theorems for (stochastic) gradient methods," 2023.

[101] *Formulations.* John Wiley and Sons, Ltd, 2020.

[102] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming.* Cambridge, MA, USA: MIT Press, 1989.

[103] M. Mitchell, *An Introduction to Genetic Algorithms.* Cambridge, MA, USA: MIT Press, 1998.

[104] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *International Conference on Computational Statistics*, 2010.

[105] A. Nemirovski, "Prox-method with rate of convergence o (1/ t) for variational inequalities with lipschitz continuous monotone operators and smooth convex-concave saddle point problems," *SIAM Journal on Optimization*, vol. 15, pp. 229–251, 01 2004.

[106] S. Shalev-Shwartz, Y. Singer, and N. Srebro, "Pegasos: Primal estimated subgradient solver for svm," in *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, (New York, NY, USA), p. 807–814, Association for Computing Machinery, 2007.

[107] G. Turinici, "The convergence of the stochastic gradient descent (sgd) : a self-contained proof," *ArXiv*, vol. abs/2103.14350, 2021.

[108] B. Polyak, "Some methods of speeding up the convergence of iteration methods," *Ussr Computational Mathematics and Mathematical Physics*, vol. 4, pp. 1–17, 12 1964.

[109] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 07 2011.

[110] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014.

[111] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, OpenReview.net, 2018.

[112] A. F. Agarap, "Deep learning using rectified linear units (relu)," *CoRR*, vol. abs/1803.08375, 2018.

[113] F. L. Bauer, "Computational graphs and rounding error," *SIAM Journal on Numerical Analysis*, vol. 11, pp. 87–96, 1974.

[114] D. Oktay, N. McGreivy, J. Aduol, A. Beatson, and R. P. Adams, "Randomized automatic differentiation," in *ICLR*, 2021.

[115] P. Peseux, M. Berar, T. Paquet, and V. Nicollet, "Stochastic gradient descent with gradient estimator for categorical features," *arXiv preprint*, vol. arXiv, 09 2022.

[116] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[117] L. Ostroumova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "Catboost: unbiased boosting with categorical features," in *NeurIPS*, 2018.

[118] S. Popov, S. Morozov, and A. Babenko, "Neural oblivious decision ensembles for deep learning on tabular data," in *International Conference on Learning Representations*, 2020.

[119] N. Frosst and G. E. Hinton, "Distilling a neural network into a soft decision tree," in *Proceedings of the First International Workshop on Comprehensibility and Explanation in AI and ML 2017 co-located with 16th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2017), Bari, Italy, November 16th and 17th, 2017* (T. R. Besold and O. Kutz, eds.), vol. 2071 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017.

[120] H. Luo, F. Cheng, H. Yu, and Y. Yi, "Sdtr: Soft decision tree regressor for tabular data," *IEEE Access*, vol. 9, pp. 55999–56011, 2021.

[121] V. Borisov, T. Leemann, K. Seßler, J. Haug, M. Pawelczyk, and G. Kasneci, "Deep neural networks and tabular data: A survey," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2022.

[122] U. Rehman, "Relation on nlp with machine and language," *Global Sci-Tech*, vol. 13, pp. 39–42, 01 2021.

[123] M. Gupta and P. Agrawal, "Compression of deep learning models for text: A survey," *ACM Trans. Knowl. Discov. Data*, vol. 16, jan 2022.

[124] Y. Mathov, E. Levy, Z. Katzir, A. Shabtai, and Y. Elovici, "Not all datasets are born equal: On heterogeneous tabular data and adversarial examples," *Knowledge-Based Systems*, vol. 242, p. 108377, 2022.

[125] R. Shwartz-Ziv and A. Armon, “Tabular data: Deep learning is not all you need,” *Information Fusion*, vol. 81, 11 2021.

[126] S. A. Fayaz, M. Zaman, S. Kaul, and M. A. Butt, “Is deep learning on tabular data enough? an assessment,” *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 4, 2022.

[127] S. Haykin, *Neural networks: a comprehensive foundation.* Prentice Hall PTR, 1994.

[128] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *arXiv preprint arXiv:1512.03385*, 2015.

[129] A. Abutbul, G. Elidan, L. Katzir, and R. El-Yaniv, “Dnf-net: A neural architecture for tabular data,” 06 2020.

[130] Y. Hayashi, “Does deep learning work well for categorical datasets with mainly nominal attributes?,” *Electronics*, vol. 9, p. 1966, 11 2020.

[131] V. Borisov, K. Broelemann, E. Kasneci, and G. Kasneci, “Deeptlf: robust deep neural networks for heterogeneous tabular data,” *International Journal of Data Science and Analytics*, 08 2022.

[132] S. O. Arik and T. Pfister, “Tabnet: Attentive interpretable tabular learning,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 6679–6687, May 2021.

[133] W. Song, C. Shi, Z. Xiao, Z. Duan, Y. Xu, M. Zhang, and J. Tang, “Autoint: Automatic feature interaction learning via self-attentive neural networks,” *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019.

[134] R. Kohavi, “Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid,” *KDD*, 09 1997.

[135] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” *arXiv preprint*, vol. arXiv, 07 2012.

[136] K. Goutam, B. S, D. Gera, and R. Sarma, “Layerout: Freezing layers in deep neural networks,” *SN Computer Science*, vol. 1, p. 295, 09 2020.

[137] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah, “Wide and deep learning for recommender systems,” in *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, DLRS 2016, (New York, NY, USA), p. 7–10, Association for Computing Machinery, 2016.

[138] M. T. Ribeiro, S. Singh, and C. Guestrin, “”why should i trust you?”: Explaining the predictions of any classifier.,” in *HLT-NAACL Demos*, pp. 97–101, The Association for Computational Linguistics, 2016.

[139] A. Blanco-Justicia, J. Domingo-Ferrer, S. Martínez, and D. Sánchez, "Machine learning explainability via microaggregation and shallow decision trees," *Knowledge-Based Systems*, vol. 194, p. 105532, 2020.

[140] L. Chen, *Curse of Dimensionality*, pp. 545–546. Boston, MA: Springer US, 2009.

[141] J. A. Blackard and D. J. Dean, "Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables," *Computers and Electronics in Agriculture*, vol. 24, pp. 131–151, 1999.

[142] U. K. Archive, "Kdd99 dataset." http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html, 1999.

[143] Kaggle, "Don't get kicked competitions." https://www.kaggle.com/c/DontGetKicked/overview, 2012.

[144] K. Lepchenkov, "Used-cars-catalog." https://www.kaggle.com/lepchenkov/usedcarscatalog, 2019.

[145] A. Griewank, "Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation," *Optimization Methods and Software*, vol. 1, 04 1994.

[146] P. Teikari, R. P. Najjar, L. Schmetterer, and D. Milea, "Embedded deep learning in ophthalmology: making ophthalmic imaging smarter," *Therapeutic Advances in Ophthalmology*, vol. 11, 2018.

[147] M. Cunneen, M. Mullins, and F. Murphy, "Autonomous vehicles and embedded artificial intelligence: The challenges of framing machine driving decisions," *Applied Artificial Intelligence*, vol. 33, no. 8, pp. 706–731, 2019.

[148] C. Badue, R. Guidolini, R. V. Carneiro, P. Azevedo, V. B. Cardoso, A. Forechi, L. Jesus, R. Berriel, T. M. Paixão, F. Mutz, L. de Paula Veronese, T. Oliveira-Santos, and A. F. De Souza, "Self-driving cars: A survey," *Expert Systems with Applications*, vol. 165, p. 113816, 2021.

[149] M.-A. Lahmeri, M. A. Kishk, and M.-S. Alouini, "Artificial intelligence for uav-enabled wireless networks: A survey," *IEEE Open Journal of the Communications Society*, vol. 2, pp. 1015–1040, 2021.

[150] P. Baldi and P. Sadowski, "Understanding dropout," *Advances in Neural Information Processing Systems*, vol. 26, 01 2013.

[151] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, vol. 25, pp. 285–294, 1933.

[152] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[153] R. Liu, T. Wu, and B. Mozafari, “Adam with bandit sampling for deep learning,” in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[154] D. Csiba and P. Richtárik, “Importance sampling for minibatches,” *Journal of Machine Learning Research*, vol. 19, 02 2016.

A. Appendices

A.1. Envision

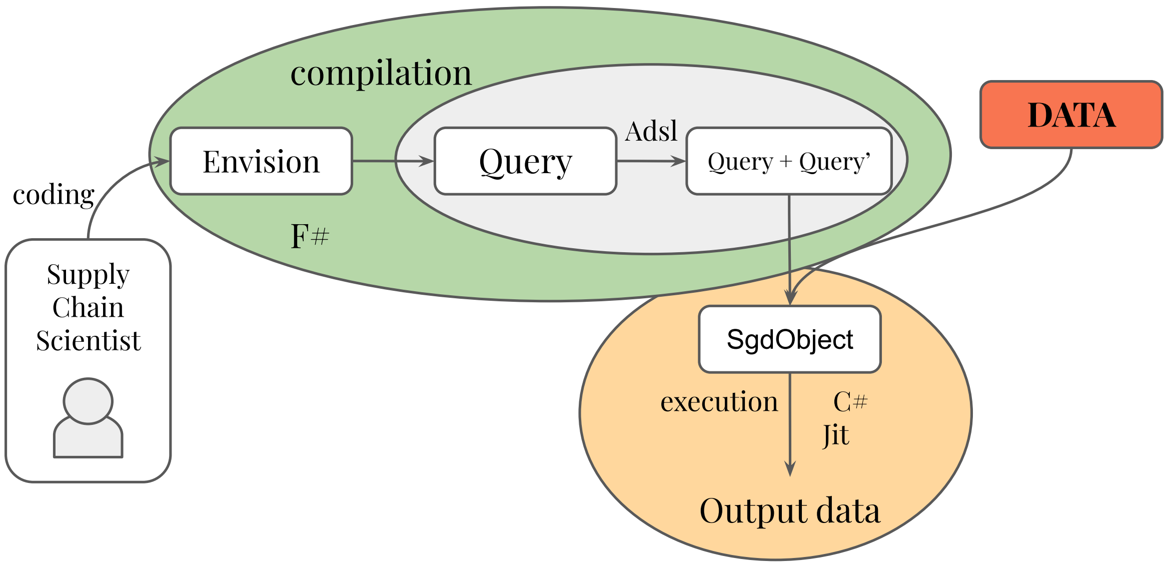
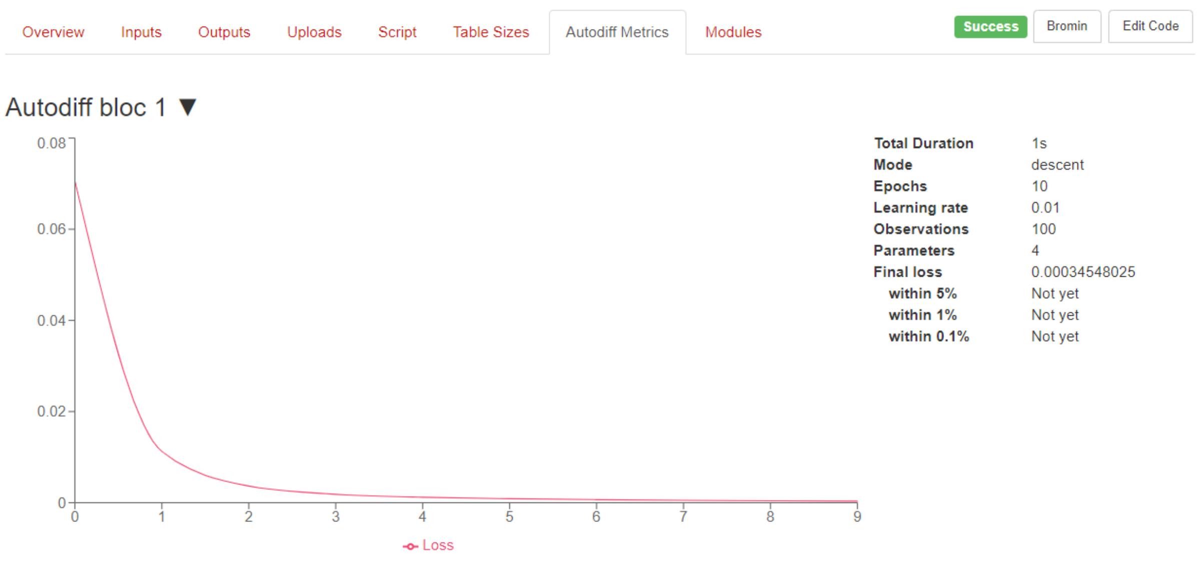


Figure A.1.: Description of pipeline implemented at Lokad. The supply chain scientist (Lokad employee) first encodes their designed model using Envision, a programming language specifically designed for this purpose. The Envision code is then compiled into multiple intermediate languages developed in $F\#$. As part of this compilation process, the model is translated into Adsl, which allows for the creation of its derivative. The inclusion of the derivative within Adsl is a fundamental aspect of its design. Finally, the query and its derivative are made accessible to an object that manages the parameter updates by executing them on available data.

```
/// size of the future Observations table.
n = 100

/// Creation of the Observations table.
/// It contains only one vector: Observations.N,
/// with values from 1 to 100.
table Observations = extend.range(n)

/// Creation of the category vector thanks
/// to the use of a random function.
/// Observations.Category has 3 possible values: "A", "B" or "C".
Observations.Category = match random.integer(3 into Observations) with
                          1 -> "A"
                          2 -> "B"
                          3 -> "C"

/// Creation of the X vector in the Observations table.
/// Observations.X values are between 0.0 and 1.0.
Observations.X = random.uniform(0 into Observations, 1)

/// Creation of scalar variables
b0 = 1.5
aA = 1.8
aB = 0.7
aC = 1.1

/// Creation of the Y vector in the Observations table.
/// The slope used depends on the Observations.Category.
Observations.Y = match Observations.Category with
                   "A" -> aA * Observations.X + b0
                   "B" -> aB * Observations.X + b0
                   "C" -> aC * Observations.X + b0

/// Creation of the Upstream table.
/// Each line of the Upstream table corresponds to one and
/// only one element of the set of values of Observations.Category.
table Upstream[Category] = by Observations.Category
```

```
/// The Stochastic Gradient Descent block.
/// This block
///       - creates the parameters 'Upstream.a' and 'b',
///       - performs sgd on 10 epochs
///         (ie 10 passes on the Observations table)
///         with a batch of 1 and Adam as optimizer,
///       - returns the updated values of the
///         parameters 'Upstream.a' and 'b'.
autodiff Observations epochs:10 with
    /// Parameters initialiaztion.
    params Upstream.a auto
    params b          auto

    /// Relational.
    a = Upstream.a
    X = Observations.X
    Y = Observations.Y

    /// Mathematical.
    prediction = a * X + b
    error = prediction - Y

    /// Construction of the loss AT THE OBSERVATIONS LEVEL.
    return error^2


/// Reconstruction of the estimated Y on the Observations data.
Observations.EstimatedY = Upstream.a * Observations.X + b0

Upstream.aTrue = match Upstream.Category with
                    "A" -> aA
                    "B" -> aB
                    "C" -> aC

/// Display.
show scalar "Estimation of the intercept (b0=1.5)" c1d1 with b

show table "Upstream" a1b3 with
  Upstream.Category
  Upstream.a as "Estimated slope"
  Upstream.aTrue as "Real slope"

show table "Observations" a4d7 with
  Observations.Category as "Category"
  Observations.X as "X"
  Observations.Y as "Y"
  Observations.EstimatedY as "Estimated Y"
```

Listing A.1: Extension of Listing I.13. This code is standalone and can be executed on `https://try.lokad.com/s/Peseux-PhD-RLRegression`. Execution of such code results in run details presented in Figure A.2



Total Duration	1s
Mode	descent
Epochs	10
Learning rate	0.01
Observations	100
Parameters	4
Final loss	0.00034548025
within 5%	Not yet
within 1%	Not yet
within 0.1%	Not yet

Figure A.2.: Envision run details of Listing A.1 execution in Lokad environment. The red curve represent the evolution of the loss function over epochs.

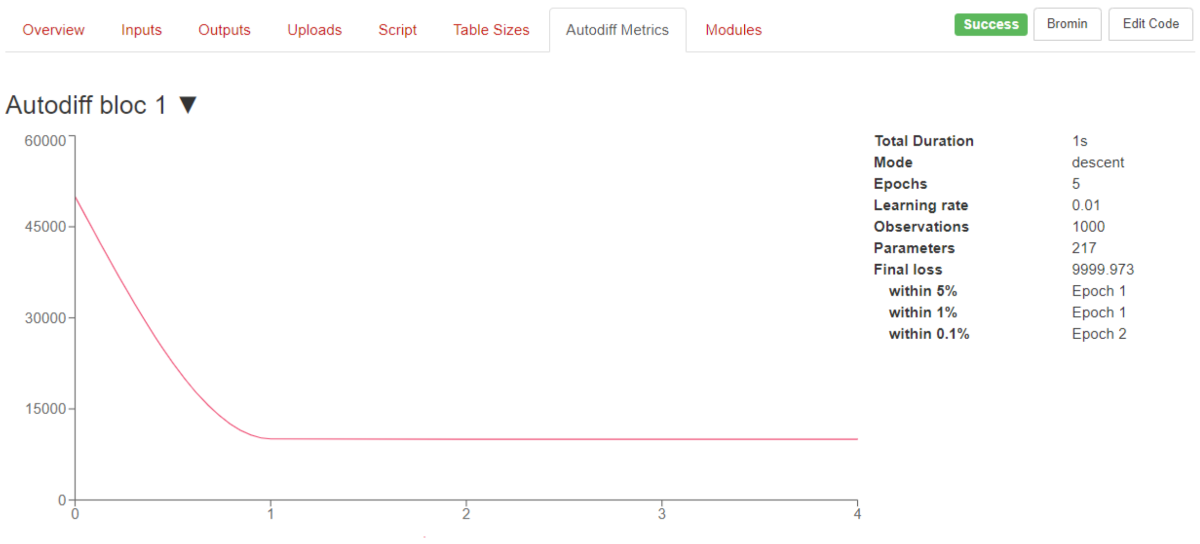
```
/// size of the future Observations table
n = 1000
/// Creation of the Observations table.
/// It contains only one vector: Items.Id, with values from 1 to 1000
table Items[Id] = extend.range(n)

/// Creation of the category vectors thanks the use
/// of a random function.
Items.Store = match random.integer(3 into Items) with
                        1 -> "Paris"
                        2 -> "Rome"
                        3 -> "Berlin"

Items.Color = match random.integer(2 into Items) with
                        1 -> "red"
                        2 -> "blue"

Items.Size = match random.integer(4 into Items) with
                        1 -> "S"
                        2 -> "M"
                        3 -> "L"
                        4 -> "XL"

Items.Group = match random.integer(4 into Items) with
                        1 -> "A"
                        2 -> "B"
                        3 -> "C"
                        4 -> "D"
```

```
/// Creation of the upstream tables.
/// Each line of the Upstream table corresponds to one
/// and only one element of the set of
/// values of Observations.Category.
table Store[Store] = by Items.Store
table Color[Color] = by Items.Color
table Size[Size]   = by Items.Size
table Group[Group] = by Items.Group


/// Creation of the date dimension.
/// This is necessary only because we deal with synthetic data.
startDate = date(2019,1,1)
endDate = date(2023,1,1)
keep span date = [startDate .. endDate]
Week.WeekNumber = (Week.week - week(startDate)) mod 52
table WeekNumbers[WeekNumbers] max 52 = by Week.WeekNumber

/// Creation of the upstream-cross tables
table GroupWN max 52k = cross(Group, WeekNumbers)
table ItemsWeek small 1m = cross(Items, Week)

/// Ensure proper broadcast between tables.
ItemsWeek.Group = Items.Group
ItemsWeek.WeekNumbers = Week.WeekNumbers


/// Synthetic creation of the objective vector ItemsWeek.Target.
/// The model used in the autodiff block below
/// is perfectly suited for the synthetic data.
/// In practice of course, it does not fit that well.
mini = 0.1
maxi = 5.0
Store.Factor   = random.uniform(mini into Store, maxi)
Color.Factor   = random.uniform(mini into Color, maxi)
Size.Factor    = random.uniform(mini into Size, maxi)
GroupWN.Factor = random.uniform(mini into GroupWN, maxi)

ItemsWeek.Factors =
  Store.Factor *
  Color.Factor *
  Size.Factor

/// Noise addition to avoid a perfect fitting.
ItemsWeek.Target =
        ItemsWeek.Factors *
        GroupWN.Factor *
        random.normal(1 into ItemsWeek, 0.2)
```

```
epsilon = 0.01
/// The Stochastic Gradient Descent block.
/// This block
///       - creates the parameters 'Store.theta',
///                                'Color.theta',
///                                'Size.theta',
///                                'GroupWN.theta'.
///       - performs sgd on 5 epochs (ie 10 passes on the Items table)
///         with a batch of 1 and Adam as optimizer.
///       - returns the updated values of the parameters
///                    'Store.theta',
///                    'Color.theta',
///                    'Size.theta',
///                    'GroupWN.theta'.
autodiff Items epochs:5 with
    params Store.theta   in [epsilon ..] auto
    params Color.theta   in [epsilon ..] auto
    params Size.theta    in [epsilon ..] auto
    params GroupWN.theta in [epsilon ..] auto

    /// Relational.
    /// The following broadcasts are possible
    /// as Store, Color, Size, Group are upstream tables.
    thetaSt = Store.theta
    thetaC = Color.theta
    thetaSi = Size.theta
    WeekNumbers.thetaGroup = GroupWN.theta
    /// Mathematical
    Week.Prediction = thetaSt * thetaC *
                      thetaSi * WeekNumbers.thetaGroup

    /// Relational
    Week.Prediction = Week.Prediction
    /// Mathematical
    Week.Error = Week.Prediction - ItemsWeek.Target
    Week.Error2 = Week.Error^2

    /// Construction of the loss AT THE OBSERVATIONS LEVEL
    loss = sum(Week.Error2)
    return loss
```

```
/// Reconstruction of the estimated vector on the Observations data.
ItemsWeek.Fit = each Items
      thetaSt = Store.theta
      thetaC = Color.theta
      thetaSi = Size.theta
      Week.thetaGroup = GroupWN.theta
      /// Mathematical
      Week.Prediction = thetaSt * thetaC *
                        thetaSi * Week.thetaGroup

      /// Relational
      Week.Prediction = Week.Prediction
      return Week.Prediction



/// Display
ItemsWeek.slice = sliceDashboard(text(ItemsWeek.Id)) by ItemsWeek.Id

show linechart "ItemsWeek" a1f4 slices:Slice with
  sum(ItemsWeek.Fit)    as  "Fitting" {color: #5F7DDF}
  sum(ItemsWeek.Target) as  "Target" {color: "#7e7"}

  group by monday(ItemsWeek.Week)
```

Listing A.2: Extension of forecasting model from Equation I.2. This code is standalone and can be executed on https://try.lokad.com/s/Peseux-PhD-ForecastinRetail. Execution of such code results in run details presented in Figure A.3



Total Duration	1s
Mode	descent
Epochs	5
Learning rate	0.01
Observations	1000
Parameters	217
Final loss	9999.973
within 5%	Epoch 1
within 1%	Epoch 1
within 0.1%	Epoch 2

Figure A.3.: Envision run details of Listing A.2 execution in Lokad environment. The red curve represent the evolution of the loss function over epochs. The data being synthetic, optimization is very smooth. The loss does not decrease to 0 due to the noise addition in data creation.

A.2. GCE

A.2.1. SVD

```
import numpy as np
n = 3

def error(M, A):
    return np.sum(np.square(M-A))

A = np.array([ [9., 3., 6.],
               [8., 8., 6.],
               [0., 3., 7.]])
U, d, V = np.linalg.svd(A, full_matrices=True)
U = np.transpose(U)

print("d:")
print(d)
print("\nU:")
print(U)
print("\nV:")
print(V)

fullDecomposition = d[0] * np.outer(U[0] , V[0]) + \
                    d[1] * np.outer(U[1] , V[1]) + \
                    d[2] * np.outer(U[2] , V[2])
approxs = [d[i] * np.outer(U[i] , V[i]) for i in range(n)]
print("error svd : %f" % error(fullDecomposition , A))
for i in range(n):
    print("approximation %i error : %f" % (i, error(approxs[i], A)))
```

Listing A.3: Construction of the singular value decomposition of a square matrix based on the numpy library.

A.2.2. Deep learning

Dataset	SGD	SGD & GCE	Adagrad	Adagrad & GCE	Adam	Adam & GCE
ACI	0.48 ± 0.24	0.24 ± 0.01	0.48 ± 0.23	**0.21 ± 0.01**	0.60 ± 0.24	0.46 ± 0.22
DGK	0.56 ± 0.35	**0.12 ± 0.01**	0.60 ± 0.35	**0.12 ± 0.01**	0.41 ± 0.36	0.35 ± 0.35
Forest Cover	1.98 ± 0.02	1.95 ± 0.01	1.98 ± 0.02	**1.44 ± 0.04**	1.98 ± 0.04	1.95 ± 0.03
KDD99	1.75 ± 0.20	**0.93 ± 0.12**	1.80 ± 0.17	**0.07 ± 0.03**	1.94 ± 0.19	1.63 ± 0.19
Used Cars	1.07 ± 0.06	**0.98 ± 0.01**	1.08 ± 0.07	**0.99 ± 0.01**	1.10 ± 0.08	1.02 ± 0.04

Table A.1.: Results with mlp and batch of 32 (RMSE)

Dataset	SGD	SGD & GCE	Adagrad	Adagrad & GCE	Adam	Adam & GCE
ACI	0.49 ± 0.10	**0.20 ± 0.01**	0.48 ± 0.14	**0.19 ± 0.01**	0.49 ± 0.14	**0.19 ± 0.01**
DGK	0.45 ± 0.19	**0.12 ± 0.01**	0.50 ± 0.20	**0.12 ± 0.01**	0.52 ± 0.24	**0.14 ± 0.01**
Forest Cover	2.01 ± 0.04	**1.18 ± 0.01**	2.01 ± 0.04	**1.03 ± 0.01**	2.00 ± 0.04	**1.05 ± 0.01**
KDD99	1.81 ± 0.10	**0.07 ± 0.01**	1.84 ± 0.08	**0.01 ± 0.01**	1.82 ± 0.12	**0.04 ± 0.01**
Used Cars	1.23 ± 0.10	**1.02 ± 0.01**	1.20 ± 0.09	**1.04 ± 0.01**	1.18 ± 0.05	**1.03 ± 0.01**

Table A.2.: Results with resnet and batch of 32 (RMSE)

Dataset	SGD	SGD & GCE	Adagrad	Adagrad & GCE	Adam	Adam & GCE
ACI	0.45 ± 0.25	0.25 ± 0.02	0.44 ± 0.23	0.20 ± 0.03	0.53 ± 0.23	0.41 ± 0.22
DGK	0.32 ± 0.32	0.11 ± 0.01	0.43 ± 0.38	0.12 ± 0.01	0.43 ± 0.38	0.29 ± 0.30
Forest Cover	2.00 ± 0.03	1.98 ± 0.02	1.99 ± 0.03	**1.54 ± 0.06**	1.99 ± 0.03	1.97 ± 0.02
KDD99	1.84 ± 0.14	**1.32 ± 0.11**	1.85 ± 0.23	**0.13 ± 0.08**	1.95 ± 0.19	1.74 ± 0.20
Used Cars	1.10 ± 0.06	**1.01 ± 0.01**	1.11 ± 0.09	1.01 ± 0.01	1.11 ± 0.10	1.05 ± 0.06

Table A.3.: Results with mlp and batch of 64 (RMSE)

Dataset	SGD	SGD & GCE	Adagrad	Adagrad & GCE	Adam	Adam & GCE
ACI	0.54 ± 0.12	**0.21 ± 0.01**	0.55 ± 0.11	**0.17 ± 0.01**	0.57 ± 0.14	**0.16 ± 0.01**
DGK	0.45 ± 0.12	**0.11 ± 0.01**	0.52 ± 0.17	**0.12 ± 0.01**	0.44 ± 0.16	**0.13 ± 0.01**
Forest Cover	2.02 ± 0.05	**1.37 ± 0.03**	1.99 ± 0.03	**1.02 ± 0.01**	2.02 ± 0.04	**1.06 ± 0.01**
KDD99	1.81 ± 0.15	**0.12 ± 0.01**	1.87 ± 0.07	**0.02 ± 0.01**	1.89 ± 0.09	**0.06 ± 0.01**
Used Cars	1.13 ± 0.06	**0.99 ± 0.01**	1.15 ± 0.07	**1.02 ± 0.01**	1.20 ± 0.17	**1.01 ± 0.01**

Table A.4.: Results with resnet and batch of 64 (RMSE)

Dataset	SGD	SGD & GCE	Adagrad	Adagrad & GCE	Adam	Adam & GCE
ACI	0.45 ± 0.22	0.32 ± 0.16	0.46 ± 0.23	0.25 ± 0.02	0.48 ± 0.23	0.42 ± 0.22
DGK	0.58 ± 0.37	0.30 ± 0.30	0.58 ± 0.35	**0.12 ± 0.01**	0.74 ± 0.29	0.49 ± 0.30
Forest Cover	1.98 ± 0.03	1.97 ± 0.02	1.99 ± 0.03	**1.60 ± 0.07**	1.99 ± 0.02	1.97 ± 0.02
KDD99	1.80 ± 0.19	1.50 ± 0.15	1.89 ± 0.19	**0.45 ± 0.47**	1.80 ± 0.18	1.69 ± 0.18
Used Cars	1.16 ± 0.09	**1.03 ± 0.02**	1.12 ± 0.08	1.02 ± 0.01	1.13 ± 0.11	1.08 ± 0.09

Table A.5.: Results with mlp and batch of 128 (RMSE)

Dataset	SGD	SGD & GCE	Adagrad	Adagrad & GCE	Adam	Adam & GCE
ACI	0.47 ± 0.12	**0.24 ± 0.01**	0.48 ± 0.15	**0.19 ± 0.01**	0.56 ± 0.09	**0.19 ± 0.01**
DGK	0.50 ± 0.19	**0.11 ± 0.01**	0.47 ± 0.18	**0.12 ± 0.01**	0.39 ± 0.13	**0.13 ± 0.01**
Forest Cover	2.00 ± 0.03	**1.59 ± 0.02**	2.00 ± 0.03	**1.01 ± 0.01**	2.00 ± 0.02	**1.04 ± 0.01**
KDD99	1.85 ± 0.15	**0.22 ± 0.01**	1.90 ± 0.11	**0.01 ± 0.01**	1.82 ± 0.13	**0.08 ± 0.01**
Used Cars	1.17 ± 0.05	**1.02 ± 0.01**	1.17 ± 0.04	**1.06 ± 0.02**	1.16 ± 0.07	**1.03 ± 0.01**

Table A.6.: Results with resnet and batch of 128 (RMSE)

Dataset	SGD	SGD & GCE	Adagrad	Adagrad & GCE	Adam	Adam & GCE
ACI	0.50 ± 0.26	0.49 ± 0.25	0.50 ± 0.23	**0.23 ± 0.01**	0.50 ± 0.26	0.43 ± 0.23
DGK	0.34 ± 0.36	0.30 ± 0.31	0.53 ± 0.36	**0.11 ± 0.01**	0.50 ± 0.36	0.28 ± 0.29
Forest Cover	1.98 ± 0.03	1.98 ± 0.02	1.98 ± 0.02	**1.79 ± 0.06**	2.00 ± 0.03	1.97 ± 0.02
KDD99	1.84 ± 0.24	1.70 ± 0.23	1.74 ± 0.10	**0.57 ± 0.33**	1.88 ± 0.23	1.78 ± 0.23
Used Cars	1.08 ± 0.07	1.04 ± 0.02	1.10 ± 0.06	**1.02 ± 0.01**	1.11 ± 0.07	1.07 ± 0.05

Table A.7.: Results with mlp and batch of 256 (RMSE)

Dataset	SGD	SGD & GCE	Adagrad	Adagrad & GCE	Adam	Adam & GCE
ACI	0.54 ± 0.14	**0.24 ± 0.01**	0.53 ± 0.13	**0.19 ± 0.01**	0.52 ± 0.10	**0.19 ± 0.01**
DGK	0.59 ± 0.16	**0.11 ± 0.01**	0.50 ± 0.16	**0.12 ± 0.01**	0.48 ± 0.19	**0.14 ± 0.01**
Forest Cover	1.99 ± 0.04	**1.73 ± 0.03**	2.01 ± 0.02	**1.03 ± 0.01**	2.03 ± 0.02	**1.07 ± 0.01**
KDD99	1.76 ± 0.07	**0.47 ± 0.04**	1.80 ± 0.05	**0.02 ± 0.01**	1.86 ± 0.09	**0.16 ± 0.02**
Used Cars	1.17 ± 0.11	**1.00 ± 0.01**	1.17 ± 0.11	1.06 ± 0.01	1.13 ± 0.06	**1.01 ± 0.01**

Table A.8.: Results with resnet and batch of 256 (RMSE)

Dataset	SGD	SGD & GCE	Adagrad	Adagrad & GCE	Adam	Adam & GCE
ACI	0.56 ± 0.24	0.50 ± 0.24	0.55 ± 0.26	0.32 ± 0.17	0.45 ± 0.26	0.41 ± 0.24
DGK	0.54 ± 0.35	0.47 ± 0.36	0.80 ± 0.23	**0.22 ± 0.11**	0.51 ± 0.38	0.50 ± 0.38
Forest Cover	1.97 ± 0.02	1.97 ± 0.02	2.01 ± 0.03	**1.92 ± 0.02**	2.01 ± 0.03	1.99 ± 0.02
KDD99	1.82 ± 0.17	1.74 ± 0.16	1.89 ± 0.18	**1.41 ± 0.17**	1.85 ± 0.20	1.79 ± 0.20
Used Cars	1.10 ± 0.08	1.05 ± 0.04	1.10 ± 0.07	0.99 ± 0.03	1.11 ± 0.11	1.08 ± 0.08

Table A.9.: Results with mlp and batch of 512 (RMSE)

Dataset	SGD	SGD & GCE	Adagrad	Adagrad & GCE	Adam	Adam & GCE
ACI	0.47 ± 0.15	**0.25 ± 0.02**	0.49 ± 0.11	**0.18 ± 0.01**	0.54 ± 0.15	**0.19 ± 0.01**
DGK	0.52 ± 0.21	**0.13 ± 0.01**	0.60 ± 0.21	**0.12 ± 0.01**	0.46 ± 0.17	**0.13 ± 0.01**
Forest Cover	2.03 ± 0.04	**1.86 ± 0.03**	2.01 ± 0.04	**1.02 ± 0.01**	2.00 ± 0.05	**1.08 ± 0.01**
KDD99	1.79 ± 0.07	**0.88 ± 0.03**	1.84 ± 0.10	**0.02 ± 0.01**	1.84 ± 0.08	**0.22 ± 0.04**
Used Cars	1.18 ± 0.07	**1.03 ± 0.01**	1.15 ± 0.08	**1.04 ± 0.01**	1.14 ± 0.05	**1.02 ± 0.01**

Table A.10.: Results with resnet and batch of 512 (RMSE)

Dataset	SGD	SGD & GCE	Adagrad	Adagrad & GCE	Adam	Adam & GCE
ACI	0.51 ± 0.26	0.50 ± 0.26	0.46 ± 0.26	0.36 ± 0.21	0.48 ± 0.25	0.45 ± 0.25
DGK	0.42 ± 0.37	0.42 ± 0.37	0.49 ± 0.37	0.16 ± 0.06	0.59 ± 0.36	0.58 ± 0.37
Forest Cover	1.99 ± 0.03	1.99 ± 0.03	1.99 ± 0.02	1.95 ± 0.01	1.99 ± 0.03	1.98 ± 0.03
KDD99	1.83 ± 0.27	1.77 ± 0.26	1.91 ± 0.29	1.67 ± 0.30	1.82 ± 0.21	1.78 ± 0.20
Used Cars	1.08 ± 0.08	1.06 ± 0.06	1.19 ± 0.13	1.08 ± 0.08	1.09 ± 0.07	1.07 ± 0.06

Table A.11.: Results with mlp and batch of 1024 (RMSE)

Dataset	SGD	SGD & GCE	Adagrad	Adagrad & GCE	Adam	Adam & GCE
ACI	0.53 ± 0.13	**0.32 ± 0.07**	0.51 ± 0.10	**0.18 ± 0.01**	0.54 ± 0.15	**0.19 ± 0.01**
DGK	0.48 ± 0.20	**0.18 ± 0.04**	0.44 ± 0.14	**0.13 ± 0.01**	0.48 ± 0.19	**0.15 ± 0.01**
Forest Cover	2.00 ± 0.04	**1.88 ± 0.04**	2.01 ± 0.04	**1.04 ± 0.01**	2.03 ± 0.04	**1.13 ± 0.01**
KDD99	1.80 ± 0.10	**1.12 ± 0.09**	1.86 ± 0.10	**0.05 ± 0.01**	1.81 ± 0.06	**0.32 ± 0.07**
Used Cars	1.11 ± 0.02	**1.05 ± 0.01**	1.19 ± 0.09	**1.03 ± 0.01**	1.12 ± 0.03	**1.01 ± 0.01**

Table A.12.: Results with resnet and batch of 1024 (RMSE)

A.3. SPAD

A.3.1. Checkpointing

Let's consider the function f:

$$\begin{aligned} f: \quad & \mathbb{R}^3 \longrightarrow \mathbb{R} \\ & \theta, w_1, w_2 \mapsto w_2 \times w_1 \times \theta \end{aligned}$$

The execution of its gradient based on RECOMPUTE-ALL strategy is presented in Figure A.4.

Operations	**Data available**
	$[\theta, w_1, w_2]$
$h_1 \leftarrow w_1 \times \theta$	
	$[\theta, w_1, w_2, h_1]$
$h_2 \leftarrow w_2 \times h_1$	
	$[\theta, w_1, w_2, \cancel{h_1}, h_2]$
$\overline{h_2} \leftarrow 1.0$	
	$[\theta, w_1, w_2, \cancel{h_2}, \overline{h_2}]$
$\overline{h_1} \leftarrow \overline{h_2} \times w_2$	
	$[\theta, w_1, w_2, \overline{h_2}, \overline{h_1}]$
$h_1 \leftarrow w_1 \times \theta$	(h_1 was not available to compute $\overline{w_1}$)
	$[\theta, w_1, w_2, \overline{h_2}, \overline{h_1}, h_1]$
$\overline{w_2} \leftarrow \overline{h_2} \times h_1$	
	$[\theta, w_1, w_2, \cancel{\overline{h_2}}, \overline{h_1}, \cancel{h_1}, \overline{w_2}]$
$\overline{w_1} \leftarrow \overline{h_1} \times \theta$	
	$[\theta, w_1, w_2, \cancel{\overline{h_1}}, \overline{w_2}, \overline{w_1}]$
$\overline{\theta} \leftarrow \overline{h_1} \times w_1$	
	$[\theta, w_1, w_2, \overline{w_2}, \overline{w_1}, \overline{\theta}]$

Figure A.4.: Execution of reverse mode automatic differentiation of f using the RECOMPUTE-ALL checkpointing strategy. This example deals with scalars but the exact same logic would apply with matrices as inputs of f, in which case the memory consumption may be an issue. Additionally, as the number of inputs (or the depth of the network in case of matrix multiplication) increases, the memory savings achieved by this technique become more significant.

A.3.2. Optimization functions

Ackley

$$ackley(x, y) = -20 \exp\left[-0.2\sqrt{0.5\left(x^2 + y^2\right)}\right] \exp\left[0.5\left(\cos 2\pi x + \cos 2\pi y\right)\right] + e + 20.$$

Beale

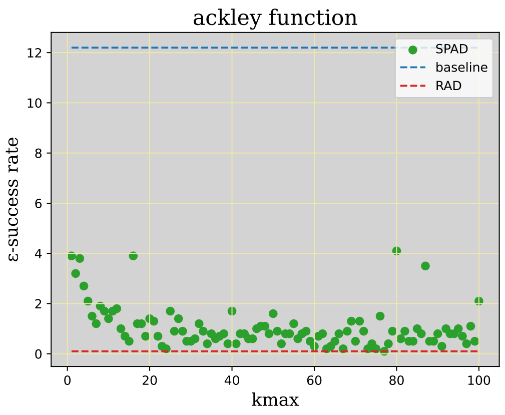
$$beale(x, y) = (1.5 - x + xy)^2 + \left(2.25 - x + xy^2\right)^2 + \left(2.625 - x + xy^3\right)^2.$$

Levi

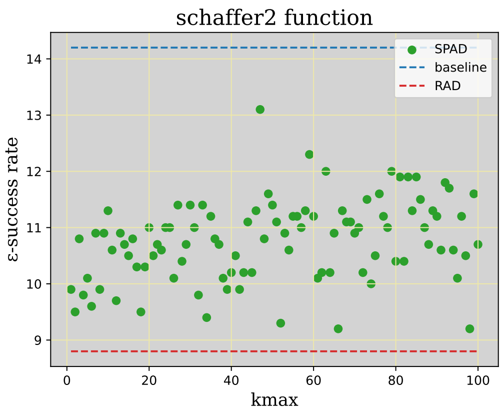
$$levi(x, y) = \sin^2 3\pi x + (x-1)^2\left(1 + \sin^2 3\pi y\right) + (y-1)^2\left(1 + \sin^2 2\pi y\right).$$

Schaffer2

$$schaffer_2(x,y) = 0.5 + \frac{\sin^2(x^2 - y^2) - 0.5}{[1 + 0.001(x^2 + y^2)]^2}.$$



(a) k_{max} on ackley function



(b) k_{max} on schaffer2 function

Figure A.5.: ϵ-success in function of k_{max}. see Figure IV.11 for more context.

A.3.3. Deep learning results

Architecture

The architectures used are directly taken from [114]. The fully-connected architecture on MNIST consists of:

1. Input: 784-dimensional flattened Image
2. Linear layer with 300 neurons (+ bias) (+ ReLU)
3. Linear layer with 300 neurons (+ bias) (+ ReLU)
4. Linear layer with 300 neurons (+ bias) (+ ReLU)
5. Linear layer with 10 neurons (+ bias) (+ softmax)

The convolutional architecture on CIFAR consists of:

1. Input: 3 × 32 × 32-dimensional Image
2. 5×5 convolutional layer with 16 feature maps (+ 2 zero-padding) (+ bias) (+ ReLU)
3. 5×5 convolutional layer with 32 feature maps (+ 2 zero-padding) (+ bias) (+ ReLU)
4. 2 × 2 average pool 2-d
5. 5×5 convolutional layer with 32 feature maps (+ 2 zero-padding) (+ bias) (+ ReLU)
6. 5×5 convolutional layer with 32 feature maps (+ 2 zero-padding) (+ bias) (+ ReLU)
7. 2 × 2 average pool 2-d (+ flatten)
8. Linear layer with 10 neurons (+ bias) (+ softmax)

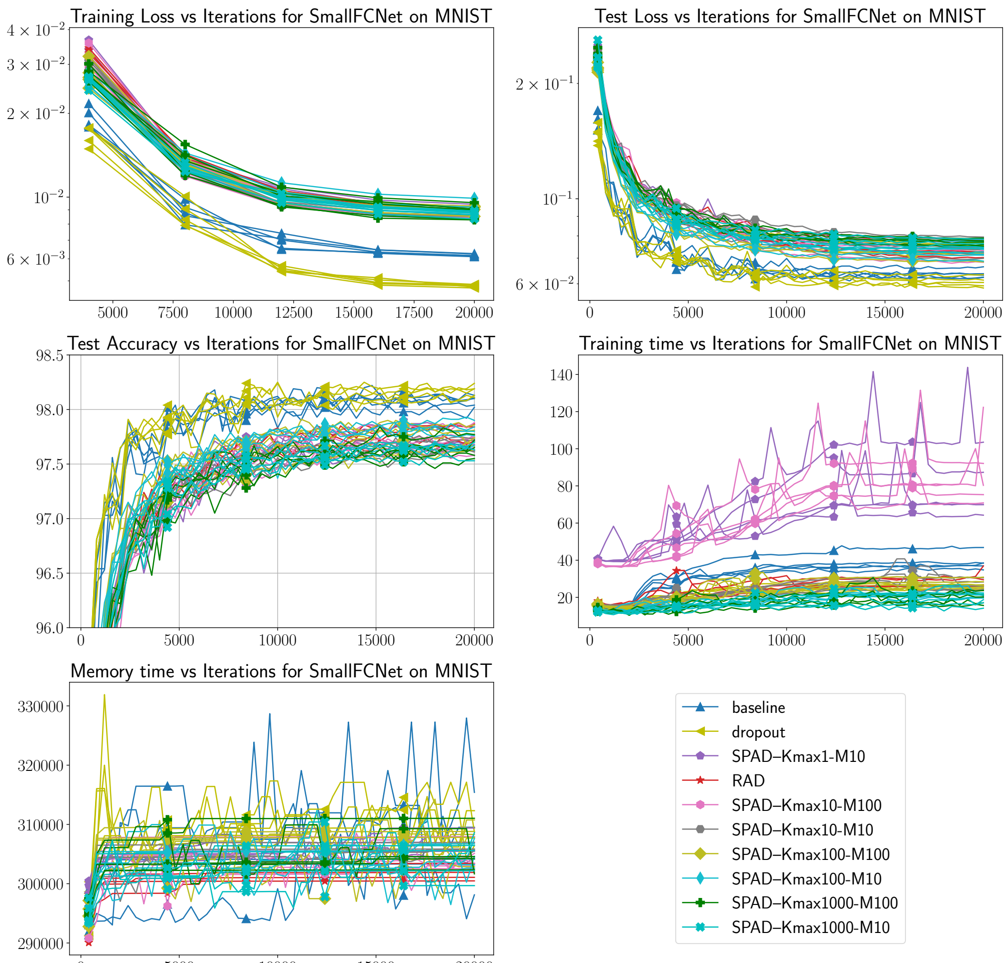


Figure A.6.: Full train/test curves and memory consumption per iteration on MNIST.

On the MNIST dataset, the dropout backpropagation techniques perform slightly worse than the baseline and dropout. We do not observe any overfitting in this task, as shown in Figure A.6, where the test accuracy does not decrease over the iterations even though this data is never used for updating the parameters. In contrast, on the CIFAR10 dataset (Figure A.7), we observe that while training accuracies consistently increase, the test accuracies tend to decrease at some point for many techniques. This is especially true for the baseline, but not for the dropout technique. Dropout backpropagation techniques help mitigate overfitting, as notably highlighted by the evolution of the testing loss. Different versions of SPAD provide an interesting range between the baseline and dropout performance.

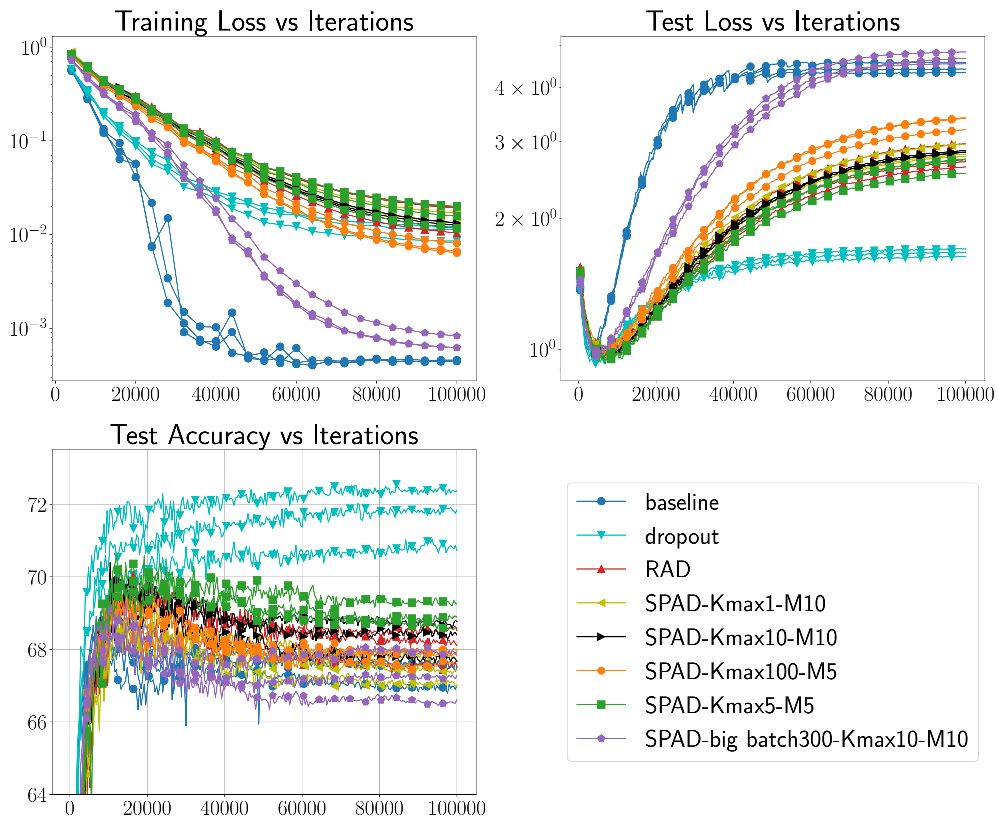


Figure A.7.: Full train/test curves and memory consumption per iteration on CIFAR10.