Normandie Université

THÈSE

Pour obtenir le diplôme de doctorat

Spécialité INFORMATIQUE

Préparée au sein de l'université Rouen Normandie

Programmation différentiable à grande échelle sur données relationnelles

Présentée et soutenue par

Paul PESEUX

Thèse dirigée par THIERRY PAQUET
et co-dirigée par MAXIME BERAR

ED 590 MIIS
Mathématiques, Information, Ingénierie des Systèmes

UNIVERSITÉ DE ROUEN NORMANDIE

litis

Contents

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 finance, healthcare, and supply chain, where data often comes in the form of structured tables or databases.

Lokad is a french company that specializes in supply chain optimization for businesses. This PhD research project is initiated and funded by Lokad with the support of ANRT through a CIFRE contract. Lokad is engaged in a diverse range of businesses, each with their unique set of supply chain challenges and constraints. They help businesses manage their inventory levels, forecast their demand, 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 [Deletoille, 2022]. To achieve this goal, Lokad heavily relies on machine learning models [Durut and Rossi, 2011]. The data used in supply chain optimization is often relational, meaning that it consists of tables of data that are interconnected in various ways. For example, a business's inventory data might be connected to their sales data, which is in turn connected to their order data, etc.

Complex and interrelated datasets present a significant challenge when applying traditional machine learning techniques [Kadra et al., 2021, Gorishniy et al., 2021], such as deep learning, to optimization problems on relational data. Conventional machine learning approaches often treat data as a collection of independent features, neglecting to consider the intricate relationships that exist among them. 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, complicating their application to such databases.

In contrast, 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. Developing white-box machine learning models for relational data is a demanding task, necessitating a deep comprehension of both the domain and underlying data structures [Deng et al., 2020]. 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 issue. By treating queries on relational databases as differentiable programs, it becomes feasible to build and optimize models capable of directly reasoning about relational data. 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.

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

Differentiable Programming

Differentiable programming is a programming paradigm that is becoming increasingly popular in the field of machine learning [Baydin et al., 2018, Li et al., 2018, Abadi and Plotkin, 2019]. 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 such as backpropagation. While programming deals with program that are not directly differentiable, they implement functions that are. Differentiating 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. 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 [Baydin et al., 2020, Roussel et al., 2022]. 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. Differentiable programming, on the other hand, allows domain experts to directly encode their knowledge into the optimization process.

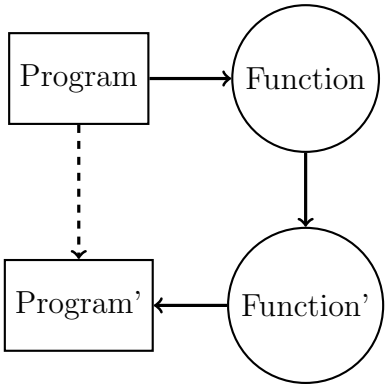


Figure 1.: Differentiable Programming. A program implements a function, the derivative of which is mathematically defined. Consequently, the derivative of the program represents the program that corresponds to the function's derivative.

The key advantage of differentiable programming is that it allows us to build complex models by composing simple building blocks. 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. In addition to being highly flexible, differentiable programming also allows domain experts to highlight their knowledge and expertise in a specific field. 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. By expressing their knowledge as a differentiable program, domain experts can ensure that their knowledge is used in a way that is both interpretable and integrated into the optimization process. This can lead to significant improvements in model performance and can also help to identify potential problems with the model.

Furthermore, differentiable programming allows for a more seamless integration be-

tween 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.

Relational data

Relational data, which is data organized in tables, is a common way to represent complex and structured data in many industries, including finance, 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, as illustrated in Figure 2.

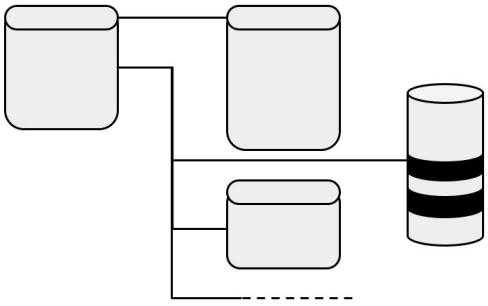


Figure 2.: Relational data illustration. Data entries are stored into tables with structural relationships between them.

However, in the modern machine learning research community, the focus has been primarily on unstructured data, such as text, image, and audio data. This is because 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 modern machine learning to relational data is 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 provides 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 [Schleich et al., 2019].

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, when dealing with large amounts of data, querying a single large table can become very slow and inefficient. This is because a large table requires more processing power and memory to handle than smaller, more normalized tables. As a result, queries may take longer to execute, reducing performance and limiting scalability. Thirdly, working with a single large table can make it more difficult to maintain data integrity and consistency. In a normalized database, we can use foreign key constraints and other database features 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.

The title of this Phd thesis is: *Large-scale Differentiable Programming for relational data.*

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 transformation of the 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 domain-specific language Envision has been augmented with differentiable programming capabilities through the design, differentiation, and implementation of Adsl. Multiple models leveraging the relational aspect of data have been developed and demonstrated in a native relational programming language environment.

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 overrepresented in relational data. 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. 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 understanding 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), an innovative technique that can be regarded as a combination of dropout and layer freezing within neural networks. The implementation of this gradient estimation approach is facilitated by the design decisions made during the differentiation of Adsl.

Plan

This Phd is separated in four chapters. **TODO** à faire en dernier dernier

Chapter I

Chapter I presents ...

Chapter II

Chapter I presents ...

Chapter III

Chapter III presents ...

Chapter IV

Chapter IV presents ...

Conclusion

We summarize these results in the conclusion of this Phd research project ...

I. Differentiating Relational Queries

The work presented in this chapter is based on [Peseux, 2021].

Introduction

Nowadays, data recording and management are essential in nearly every field, although this data is highly heterogeneous, including image, sound, text, and physical measurements, among others. Relational data is a key type of data in domains such as healthcare or supply chain management, where patients may have multiple health problems and take incompatible drugs or customers may purchase items from multiple providers. In these cases, the data structure itself is highly valuable: both theoretical and practical frameworks have been developed and widely used to handle these structures. Database systems are often employed to query these structures, and non-mathematical professionals frequently use these systems to retrieve information. With the emergence of machine learning, experts in various fields are eager to utilize this technology to perform tasks such as regression or classification, which are essential in healthcare or supply chain management. However, there is currently no machine learning tool in database systems. This absence 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. 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 [Vaswani et al., 2017] 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 model can be created by a domain expert 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, 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.

This chapter is organized as follows. First, we present the relational algebra theory to properly define how we aim to handle relational data. Second, we 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. Then 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 implementation 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. Relational Query

The primary objective 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 we precisely define using the principles of relational algebra theory.

I.1.1. Relational algebra

Relational algebra is a theoretical framework for handling and querying structured data, which was first introduced by Codd [Codd, 1970]. 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 later 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* $t_{i\leq n}$*.*

Definition 2 (primary key)**.** *The primary key of a table is a subset of the table's attribute such that the tuples values are unique throughout the table. 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:

$$\Pi_{A \subset \{a_1 \dots a_m\}}(T)$$

The projection can be extend into generalized projections that applies 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:

$$\Pi_f(T)$$

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:

$$\sigma_\phi(T)$$

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 to different tables. This is defined in Formula I.1:

$$R \times S = \{(r_1, \dots, r_n, s_1, \dots, s_m) | (r_1, \dots, r_n) \in R, (s_1, \dots, s_m) \in S\}. \tag{I.1}$$

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 given predicate θ rather than the simple matching of the values on the shared attributes.

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

AGGREGATION

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

$$G_{agg(a)}(T)$$

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

NULL

In order to handle missing or impossible values, relational algebra introduces the NULL values. It is a way to indicate that certain data does not exist in the database. It is distinct from a default value and a NULL value in a float column should not be replaced by $0.0f$. 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. Moreover the TOTAL JOIN operator later introduced in Section I.3.4 cannot create NULL values by design. Without NULL values as input and with operators that do not create ones, this assumption is realistic.

Remark 1. *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.1.2. Query

A query is the result of applying a combination of operators on input tables. As described in Section I.1.1, 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.

Having presented the theoretical framework of relational algebra, we contend that performing machine learning within it is crucial to achieving optimal performance on regression or classification tasks using such data.

I.2. Motivation and existing solutions

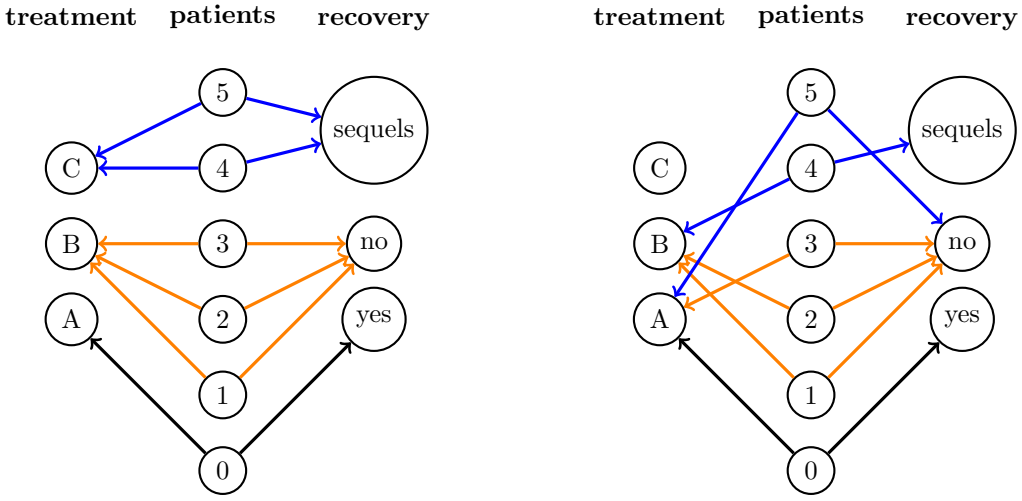
I.2.1. Relational data is specific

Modern data is extremely heterogeneous. We can think of image, sound, text, meteorologic measurements ... Some are strongly structured. We can think of supply chain data that store information on stores, warehouses, clients ... This data is strongly structured in the sense where a data singleton does not bring any information while its connections with the other do. This contrasts with an image that makes sense by itself. Relational algebra is the adapted framework for this strongly structured data. As presented in [Borisov et al., 2021] or [Gorishniy et al., 2021], tabular data, as opposed to images, are heterogeneous, i.e. they present high variability of data types and formats, in their underlying structure. They often contain categorical input features and are strongly structured. This structure is thus specific to each tabular dataset and a modification of an input categorical feature might completely change the meaning of the concerned data while a pixel modification does not fundamentally change an image. To highlight this statement,

we present two different data and their modification with 40% of noise. On one hand, Figure I.1 represents two instances of the same tabular data storing medical treatment information on different patients. (a) is the clean data while (b) 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

On the other hand, Figure I.2 represents two instances of the image. (a) is the clean data while (b) 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.1.: Relational data and noise. Figure I.1a represents clean toy relational data and Figure I.1a depicts its polluted version. Such kind of relational data do not support this level of noise.

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

Figure I.2.: Image data and noise. Figure I.2a represents a clean image of a logo and Figure I.2b depicts its polluted version. Such kind of image data do not supports this level of noise, as the main information are still present.

This aims to convince that relational data deserve a specific treatment, especially to perform machine learning on it.

I.2.2. Machine Learning

Machine learning is a collection of techniques that use examples to learn and generalize to perform specific tasks. It 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. The success of these models is often measured by their ability to minimize a specific function known as a loss function. Optimization techniques, such as gradient descent, are used to minimize the loss function by adjusting the model's parameters. Gradient descent works by following the direction of the gradient, which provides the best linear approximation of the function at a particular point, 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 provided in Chapter II.

Remark 2. *In the following we assume that all the considered functions are differentiable, even though differentiation is still possible in wider area [Lee et al., 2020]*

I.2.3. Differentiation and Differentiable Programming

The derivative of a function f from $\mathbb{R} \longrightarrow \mathbb{R}$ is:

$$f'(x) = \lim_{h \longrightarrow 0} \frac{f(x+h) - f(x)}{h}. \tag{I.2}$$

In generalizes to higher dimension and is thus called gradient:

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

$$\nabla f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_p} \\ \cdots & \cdots & \cdots \\ \frac{\partial f_q}{\partial x_1} & \cdots & \frac{\partial f_q}{\partial x_p} \end{bmatrix} \tag{I.3}$$

The matrix from Equation I.3 is called the Jacobian. Many optimization techniques are gradient based. These iterative optimizations are performed step by step by following the best local update, i.e. the gradient, to get closer from the optimal solution.

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.3. 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. 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 another one. An example is given in Section I.4.4

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

Figure I.3.: 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.4. The most common methods for numerical differentiation are the finite difference method, which involves subtracting the function values at two points and dividing the result by the difference in their independent variable. This approximations become more accurate as the interval becomes smaller. Formula I.4 and I.5 respectively present the forward difference and the central difference methods.

$$[\nabla f(a)]_{i,j} \approx \frac{f_i(a + h_j e_j) - f_i(a)}{h_j} \tag{I.4}$$

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

with e_j is the unit vector in the j^{th} direction and h_j represents a step size. The forward differences need $p(q+1)$ evaluation 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 roundoff and truncation. Second, it is sensitive to the choice of step size, and the results can be highly dependent on the choice of method and the behavior of the function being differentiated. Third, numerical differentiation can be computationally expensive, especially for high-dimensional functions or for functions with multiple local extrema.

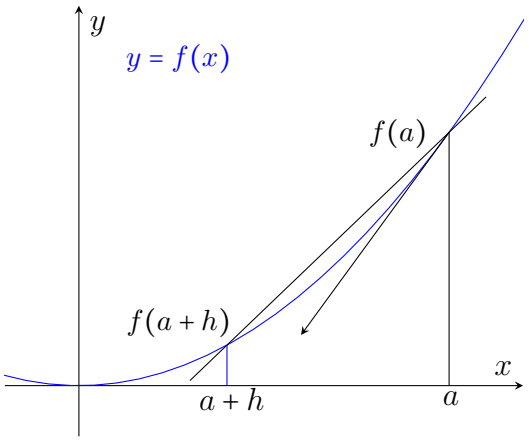


Figure I.4.: Illustration of numerical differentiation of f. The arrow starting from $(a, f(a))$ represents the tangent of the gradient of f at a, oriented in order to decrease f.

Automatic differentiation

Automatic differentiation 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 from calculus.

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.

Automatic differentiation is fully part of the huge rise of machine learning. The scientists do not need to manually implement the derivative of their target function which is error prone and time consuming. To some extent, Differentiable Programming makes every writable model optimizable.

One can also think about Symbolical Differentiation, but this is proven equivalent to automatic differentiation [Laue, 2019]. Symbolic differentiation is a method that involves algebraic manipulation of a function's mathematical expression in order to obtain an exact representation of its derivative.

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. As with the computation of the function to differentiate, there might be numerical imprecision. These are inherent to computer computation. This automatic differentiation approach is fully presented in Section I.4.3.

I.2.4. Existing automatic differentiation systems

To facilitate the implementation of gradient-based algorithms, automatic differentiation (AD) is essential. Many subsets of programming languages can be automatically differentiated, as it is a widely studied subject [Hascoët and Pascual, 2013, Innes, 2018, Baydin et al., 2018, van Merrienboer et al., 2018]. The most well-known libraries that rely on automatic differentiation include PyTorch [Paszke et al., 2019] and TensorFlow [Abadi et al., 2015]. However, 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 [Schüle et al., 2019], have begun to address the subject by differentiating a portion of the SQL language. While this is promising, the authors still describe it as an architectural blueprint. Their aim is to replicate the gradient-based methods used in tensor-based systems, but without fully embracing the relational potential of SQL. Their primary objective is to reduce data transfer costs, which is important, but in our view, the main challenge lies in enabling machine learning to fully leverage the relationship between input and model parameters.

As discussed in Section I.1.1, 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 absence of automatic differentiation systems in database systems. Indeed, as presented in Section I.2.3, modern machine learning relies on gradient methods; without an automatic differentiation system, machine learning is thus not feasible. This lack of suitable tools

for handling relational data may explain why tabular datasets are referred to as "the last unconquered castle for deep learning" by [Kadra et al., 2021].

Furthermore, the database systems community and the machine learning community remain distinctly separate. The absence of 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 are 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.

I.2.5. Tensor based systems are not suited for categorical data

As presented in Section I.2.1, data is heterogeneous, but most machine learning systems rely on tensor types such as Pytorch or Tensorflow. This is perfectly suited for many applications such as image, but this is not the best framework for relational data, while relational algebra is. For example, Pandas library [Wes McKinney, 2010] 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 designed for. As a consequence, it is still painful to do it and is error-prone. Enabling machine learning in database systems decreases the number of libraries used to perform the same task. It thus limits data transfer that is often a significant time-consuming part, as measured in [Schüle et al., 2019] for machine learning tasks. To support this statement we argue that in many production models at Lokad, most of the resources are used to load and prepare the data, while the training 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%.

The aforementioned ratio is not applicable to tensor-based systems running deep neural networks, as the enormous resource consumption is often beyond the reach of small companies. However, since boosting methods are still superior to deep learning when it comes to tabular data, the use of such large models is not yet necessary for these types of data. The use of a unified and appropriate framework for all aspects of data processing, including loading, processing, cleaning, training, and inference, can lead to faster task completion and reduced implementation errors.

We have highlighted the significance of differentiating relational queries in performing important tasks on relational data. Existing solutions are not adequate as they are tailored specifically to relational data. Therefore, our aim is to build a framework where differentiable programming is a first-class citizen in relational programming languages. We will first present our approach to constructing this framework, which involves separating the query into its relational and mathematical components.

I.3. Divide the Query to conquer

In the following section, we develop a theoretical set up for query differentiation. As motivated in Section I.2, this will allow us to build and machine learning models and optimize them through gradient descent. The main idea is to split the operations of the query $\mathcal{R}$ into two parts. First the relational operations on the tables T_s and their relations that just propagate gradient. Second, the mathematical operations f that create complex gradients following automatic differentiation rules. This separation is presented in Figure I.5.

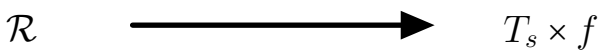


Figure I.5.: Query splitting into its relational and its mathematical parts.

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 $\theta^\star \in \mathbb{R}^m$ $(m \geq p)$ to minimize the loss $F_{\theta^\star}$ on the whole dataset:

$$\begin{aligned} f: \quad & \Theta \times \mathcal{Z} \longrightarrow \mathbb{R} \\ & \theta, (X, y) \longrightarrow f_\theta(X, y) \end{aligned}$$

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

Our strategy to get closer and closer from $\theta^\star$ is to perform gradient descent 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 challenge is to access these entries without loading the full tables for each computing of the gradient of f. All the following is oriented in this direction.

I.3.2. A query is relational and mathematical

Relational

We have presented relational operators in Section I.1.1. In order to fit the minimization presented above, the query has to end as an SUM-aggregation of a projection of the loss

attribute in a given table, later presented as the observation table in Section I.3.5. It takes the form of Formula I.6.

$$G_{SUM(loss)}(\Pi_{loss}(Observations)) \tag{I.6}$$

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.7.

$$\sigma(Z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}. \tag{I.7}$$

One can find the list of the operation we have implemented (and their derivative) in our query differentiation system in Table I.1.

I.3.3. Differentiable Programming on relational queries

Differentiable programming is represented in Figure 1. Applying this representation to differentiation of relational queries lead to Figure I.6 where the query is split into its mathematical function f and the tables on which the query occurs.

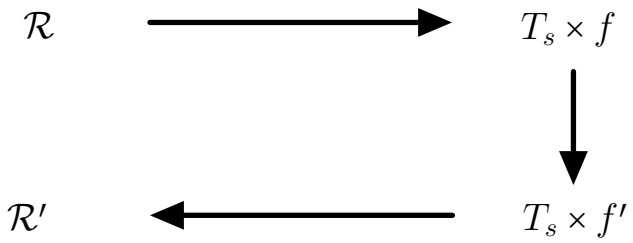


Figure I.6.: Path to differentiation. The direct differentiation of $\mathcal{R}$ to $\mathcal{R}'$ does not exist yet so we introduce a novel way to do it.

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 observation pertains to the optimizations available during both compilation and execution time of the relational database in which such differentiation occurs.

Secondly, differentiable programming is highly programmatic. When external tools call relational queries, returning the gradient as a query enables composability. In Section I.4.3, 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.3.4. TOTAL JOIN operator

Many database management tools are a sort of implementation of the model presented in I.1.1. 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 $\langle\theta\rangle$ 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_2, 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.1: INNER JOIN.

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

Listing I.2: TOTAL JOIN.

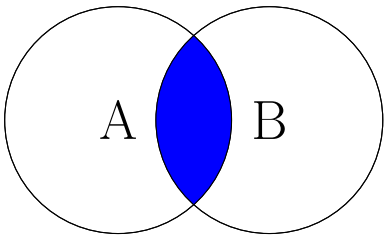


Figure I.7.: INNER JOIN representation. Without any restriction on the tables, there might be values for the K attribute in the B table that are not present in the A table.

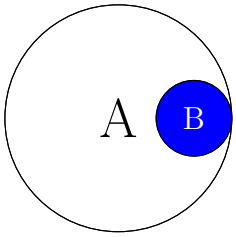


Figure I.8.: TOTAL JOIN representation. With restrictions on the tables, all the lines of B are concerned by this join operation.

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 allow us to define *PolyStar* (see Section I.3.5). The difference is highlighted in Figures I.7 and I.8. This operator also allow us to avoid dealing with NULL values as presented in Section I.1.1 with strict conditions on the tables relationships.

I.3.5. PolyStar

In the following section, we introduce the definition of the *PolyStar*. This a is key concept in order to properly separate the mathematics and the relational part of a query, which is the core of our differentiation approach. Lets start by the introduction of the *Polytree*, 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.9. 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.

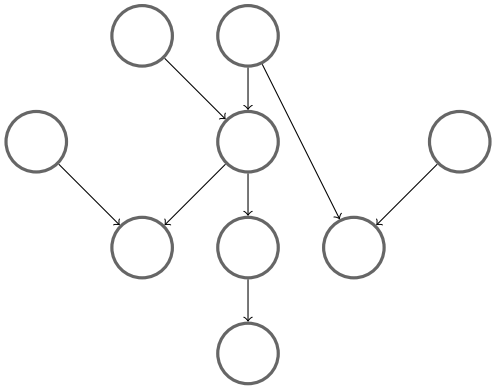


Figure I.9.: A generic Polytree.

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.3.

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

Listing I.3: Creating broadcasts

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 an edge in SQL, which is the implementation of the Cartesian Join presented in Formula I.1:

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

Listing I.4: SQL cross edge creation.

Definition 6 (PolyStar)**.** *Let's define a PolyStar* $P\star = (P, n)$ *as a Polytree* P *with cross-edges and* n *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.
From the special focus on a specific node, that we now call the observation node, we can classify the other nodes. Let $(P, ot) \in P\star$, we call

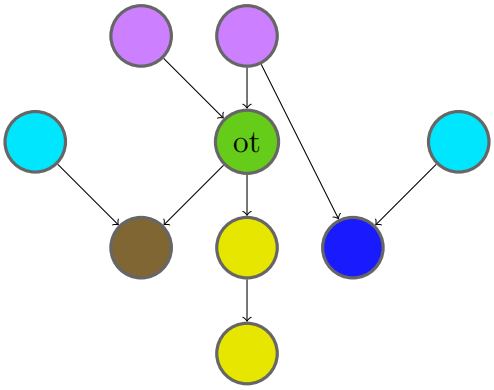


Figure I.10.: A PolyStar with its corresponding coloration.

- 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 is not an observation-cross node and that $ot \longrightarrow d$.
- a *full* node all the remaining nodes of P.

These definitions uniquely define nodes in the PolyStar, a generic example is given in Figure I.10. Two concrete examples on properly defined models are presented in Sections I.5.3 and I.5.4. These names (upstream, downstream ...) are introduced with the PolyStar, and are inspired by the flux' paths in the table tree. This table classification is built in order to fit the stochastic gradient framework. As presented in Section II, modern gradient based methods are based on an estimation of the gradient, as the full one is too large to be computed. The granularity of the estimated gradient is the observation table one, this explains why every other table is colored regarding their relationships to the observation table. The purpose of the query is to build a numerical vector (related to an attribute to fit with notations from Section I.1.1) in the observation table and minimize its sum over the observation table, as presented in Formula I.6. The observation table corresponds to the $\mathcal{Z}$ set of Section I.3.1. 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 left table used in the cross operation.
- An input from an observation-cross table reduces to a vector of the size of the left 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.

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. Concretely, this property is highlighted in Listings I.8 and I.9 on actual examples. 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 model is crucial in enabling us to properly define the quantity we aim to minimize.

I.4. A dedicated programming language: Adsl

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. 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 AD, making it fully differentiable. Then, the compilation between these specialized languages and target languages needs to be implemented. Although the implementation of the compilation from and to Adsl for each programming language is necessary for this method, the automatic differentiation system only needs to be implemented once. We have opted for the second approach, as illustrated in Figure I.11.

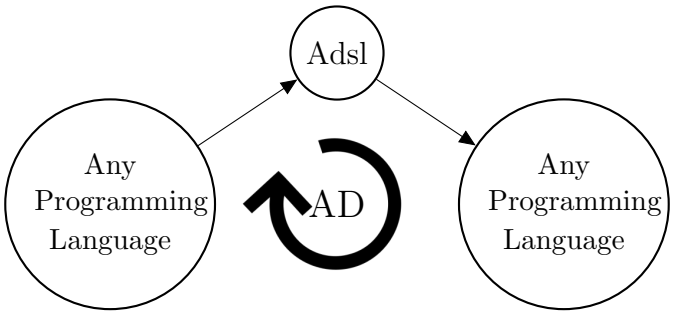


Figure I.11.: Adsl automatic differentiation schema.

We introduce Adsl[1], which is **A** **D**ifferentiable **S**ub **L**anguage that is intended to lower relational language. It is a language where automatic differentiation is a first class citizen. This idea is similar to [Abadi and Plotkin, 2019] [Hu et al., 2020] [Mak and Ong, 2021]. Adsl 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. Its major specificity is its aggregators and broadcasts between tables support. As we crafted it from scratch, we designed it with specific properties, presented in Sections I.4.1 and I.4.2 that make Adsl differentiation as easy as possible. It also created a novel gradient estimator described in Chapter IV.

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

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.

$\langle\ S\ \rangle$ ::=

$\langle\ v \leftarrow e \rangle$	Variable assignment
$\langle\ tup \leftarrow v \rangle$	Variable tupling
$\langle\ Cond\ (\ v \quad \Psi \quad P_T \quad P_E \quad \Phi) \rangle$	Conditional
$\langle\ For\ (\ \tau \quad \chi \quad rev \quad S \quad P \quad \Xi) \rangle$	Loop
$\langle\ Return\ v\ \rangle$	Output of a program

$\langle\ e\ \rangle$ ::=

$\langle\ v\ \rangle$	Variable
$\langle\ f\ \rangle$	Scalar
$\langle\ \oplus \quad tup \rangle$	Variable Addition
$\langle\ Call\ op\ tup\ \rangle$	Function Call
$\langle\ Param\ i\ \rangle$	Parameter access
$\langle\ Const\ i\ \rangle$	Constant access
$\langle\ v \quad \triangleleft \quad \beta \quad \rangle$	Broadcast Projector
$\langle\ v \quad \triangleright \quad \alpha \quad \rangle$	Aggregation Projector
$\langle\ Pred\ \rangle$	Predicate

$\langle\ Pred\ \rangle$ ::=

$\langle\ \wedge \quad v \quad w \rangle$	And
$\langle\ \vee \quad v \quad w \rangle$	Or
$\langle\ \neq \quad v \quad w \rangle$	Inequality
$\langle\ v \leq w \rangle$	

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.

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 (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 the following :

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

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 and cannot be updated.

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:

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

Example 2 (Adsl conditional)**.** *Let's consider the following pseudo code of conditional in Listing I.5.*

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

Listing I.5: pseudo code of a program containing a conditional.

Its Adsl form is

$$
\begin{aligned}
&\langle x_0 \leftarrow \ldots \rangle \\
&\langle Cond(v \quad \Psi \quad P_T \quad P_E \quad \Phi) \rangle \\
&\langle z \leftarrow \oplus \quad y \quad 1 \rangle
\end{aligned}
$$

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. This is key in order to satisfy the wished properties defined in Definitions 7 and 8. 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 are a key feature of Adsl as it is attended to perform automatic differentiation of relational programming languages. A broadcast from a table to another clones 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 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

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

Aggregators are the opposite of broadcasts where the values are aggregated (the default aggregator is the *sum*) into a small table. If no relationship between the tables specified, the natural one applies: the aggregation implied by the foreign key of the input table matching the primary key of the target table.

Formally aggregators and broadcasts can be seen as a multiplication by a 0-1matrix. $\langle w \leftarrow v \triangleleft \beta \rangle$ 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 on gives:

$$\begin{pmatrix} 1.8 \\ 1.2 \\ 1.3 \\ 1.8 \\ -2.2 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1.2 \\ 1.8 \\ -2.2 \end{pmatrix} \tag{I.8}$$

This matrix point of view is helpful to understand their behavior, but in practice Adsl broadcasts and aggregators are not implemented as matrices.

In the two following we present characteristics of Adsl that have a direct impact on the differentiation process of the language.

Loops

Loops are a key part of Adsl as they unlock complex model creation and are a major enhancement compared to [Schüle et al., 2019]. Moreover loops are the most resource consuming operations, thus a special care has been done to design it. 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. 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.6: pseudo-code of a program containing a loop.

Its Adsl form is

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

with

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

The Scan operator, also known as prefix-reduce (Scan is the name popularized by the MPI library [Message Passing Interface Forum, 2021]), 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.12.

By example, the Scan operator can implement exponential smoothing [Gardner, 1985]

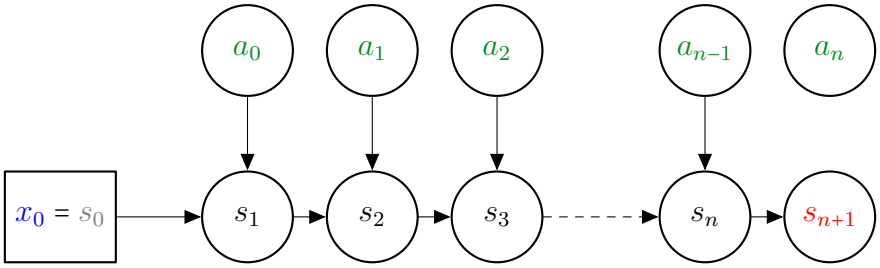


Figure I.12.: Scan Representation. The state s is initialized with x_0 and is updated while iterating the function f over the vector A.

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

$$s_0 = x_0$$
$$s_t = f_\alpha(s_{t-1}, x_t)$$

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.5.6 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.4.1. 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 (IR), 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 language and Zygote [Innes, 2018] also rely 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.

I.4.2. 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 IR, 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:

$$
\begin{aligned}
&x \leftarrow \mathit{Param}_0\\
&y \leftarrow \mathit{Param}_1\\
&a \leftarrow e^x\\
&b \leftarrow \oplus \quad x \quad y\\
&z \leftarrow a \times b\\
&\mathit{Return} \quad z
\end{aligned}
\qquad \text{becomes} \qquad
\begin{aligned}
&x \leftarrow \mathit{Param}_0\\
&y \leftarrow \mathit{Param}_1\\
&x_1, x_2 \leftarrow x\\
&a \leftarrow e^{x_1}\\
&b \leftarrow \oplus \quad x_2 \quad y\\
&z \leftarrow a \times b\\
&\mathit{Return} \quad z
\end{aligned}
$$

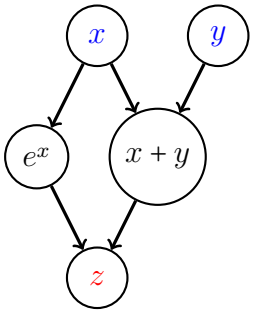


Figure I.13.: SSA graph of $f_1(x, y)$.

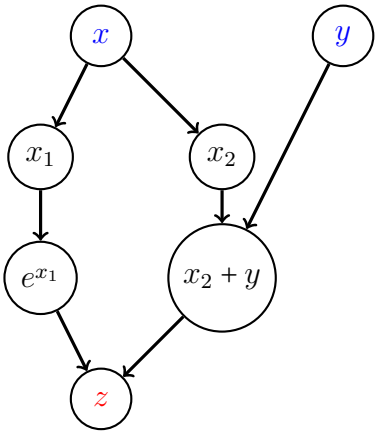


Figure I.14.: SSA-SA graph of $f_1(x, y)$.

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.

It is important to note that even though Figures I.13 and I.14 are depicted as graphs, they represent the function f_1 and are not PolyStars. PolyStars pertain to the tables utilized in a query rather than the mathematical aspect of the function.

The usefulness of such property might seem unclear at first sight. 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 3. *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 use in the objective function. This approach might not seem relevant for deep neural networks with billions of parameters.

I.4.3. Automatic differentiation

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

The objective is to automatically compute the Jacobian presented in Equation I.3. Automatic differentiation is applied on an intermediate representation of a program introduced as the Wengert lists. We present them and explain how an SSA-SA Adsl program is an adequate representation of such a list.

Wengert Lists

A Wengert list [Wengert, 1964] 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 SSA variables. These variables are called Wengert variables. It is as basic as it can be. All loops are unrolled. There are no conditional statements, it just keeps the branch chosen at the execution.

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.

Theorem 1 (Wengert lists and Adsl equivalence)**.** *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 behavior can be represented in Adsl using the following statements:

$$\langle v_t \longleftarrow True \rangle$$
$$\langle Cond(v_T \quad \Psi \quad P_T \quad P_E \quad \Phi) \rangle$$

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 is 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 regrading the minor modifications made to the already SSA-SA program.

Which concludes the proof. □

The equivalence between Wengert lists and Adsl, 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.

Once the equivalence between Wengert lists and SSA-SA Adsl is established, we present the two main approaches to perform automatic differentiation on a program . Automatic differentiation on Wengert lists compute 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 $|\langle v \longleftarrow \text{Param i} \rangle$ statement. Automatic differentiation can be implemented in two modes: reverse of forward. Both rely on the chain rule, depicted in Equation I.9, but they do not run through it in the same direction.

The chain rule applied to $h = f \circ g$:

$$h' = (f \circ g)' = (f' \circ g) \cdot g'. \tag{I.9}$$

In the subsequent sections, we provide a formalization of two distinct approaches to applying the chain rule for differentiating a program, known as the reverse and forward modes. We begin by discussing the reverse mode, as it has been selected for implementing automatic differentiation in Adsl.

Reverse mode

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

$$\overline{\omega} = \frac{\partial f}{\partial \omega}.u, \tag{I.10}$$

$\overline{\omega}$ can be seen as the sensitivity of the output $f(x) \in \mathbb{R}^q$ in the cotangent direction with respect to ω.

For the intermediary Wengert variable ω, one can access all the variables $(\omega_1 \dots \omega_L)$ that directly use ω to be computed with the functions $(f_1 \dots f_L)$:

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

L can be greater than 1 as a Wengert list is not necessarily SA. The chain rule gives us:

$$\begin{aligned}\overline{\omega} &= \frac{\partial f}{\partial \omega}.u \\ &= \sum_{l=1}^{L} \frac{\partial f}{\partial \omega_l}\frac{\partial \omega_l}{\partial \omega}.u \\ &= \sum_{l=1}^{L} \overline{\omega_l}\frac{\partial f_l}{\partial \omega}.\end{aligned}$$

This is called reverse accumulation as the adjoint of ω accumulates all the incoming adjoints of its children in the execution graph. On 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. Thanks to the SA property of Adsl, the reverse accumulation does not exist anymore, an intermediary Wengert variable can not have more than one child. This is replaced by the tupling operation and the adjoint of tupling is the sum of the adjoints:

$$\begin{aligned} x_1, \dots x_i &\leftarrow x \\ &\dots \\ \overline{x} &\leftarrow \overline{x_1} + \dots + \overline{x_i} \end{aligned}$$

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

$$\begin{aligned} x &\leftarrow Param_0 \\ y &\leftarrow Param_1 \\ x_1, x_2 &\leftarrow x \\ a &\leftarrow e^{x_1} \\ b &\leftarrow \oplus \quad x_2 \quad y \\ z &\leftarrow a \times b \\ &Return \quad z \end{aligned}$$

reverse mode gives

$$\begin{aligned} x &\leftarrow Param_0 \\ y &\leftarrow Param_1 \\ x_1, x_2 &\leftarrow x \\ a &\leftarrow e^{x_1} \\ b &\leftarrow \oplus \quad x_2 \quad y \\ z &\leftarrow a \times b \\ \overline{z} &\leftarrow 1 \\ \overline{a}, \overline{b} &\leftarrow \overline{z}b, \overline{z}a \\ \overline{x_2}, \overline{y} &\leftarrow \overline{b} \\ \overline{x_1} &\leftarrow \overline{a}e^{x_1} \\ \overline{x} &\leftarrow \oplus \quad \overline{x_1} \quad \overline{x_2} \\ &Return \quad \overline{x}, \overline{y} \end{aligned}$$

When $q = 1$, the cotangent becomes a scalar and one can choose 1 which simplifies to $\overline{\omega} = \frac{\partial f}{\partial \omega}$. For the parameters θ, the choice of $u = 1$ gives us what we are looking for. In the problem we aim to tackle and optimization in general, we mainly have $q = 1$. One can minimize scalars but not vectors because $q = 1$ is the only case where $\mathbb{R}^q$ has a total order.

Forward mode

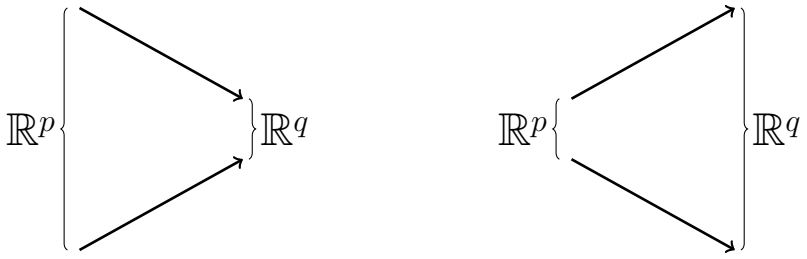
In the forward mode of AD, we start from an initial tangent $u \in \mathbb{R}^p$ and the input $x \in \mathbb{R}^p$ both in the input space. For an intermediary Wengert variable ω that is formally computed from the input x by f_ω ($\omega = f_\omega(x)$), the tangent of ω is defined as:

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

As a Wengert list is SSA, one can access the function f_{WL} that creates it and the input variables $\omega_{WL} = (\omega_1 \dots \omega_L)$. For ω_{WL}, one can also define $f_{\omega_{WL}}$ that gives $f_\omega = f_{WL} \circ f_{\omega_{WL}}$. Then the chain rule gives us:

$$\begin{aligned}
\dot{\omega} &= \frac{\partial \omega}{\partial x}.u = \frac{\partial f_\omega}{\partial x}.u = \frac{\partial f_{WL} \circ f_{\omega_{WL}}}{\partial x}.u \\
&= \frac{\partial f_{WL}}{\partial f_{\omega_{WL}}} \circ f_{\omega_{WL}}.\frac{\partial f_{\omega_{WL}}}{\partial x}.u \\
&= \frac{\partial f_{WL}}{\partial f_{\omega_{WL}}}.(\dot{\omega_1} \dots \dot{\omega_L}) \\
&= \sum_{l=1}^{L} \frac{\partial \omega}{\partial \omega_l}\dot{\omega_l}.
\end{aligned}$$

With this formula, one can compute $f(x)$ and its u-tangent simultaneously. In order to obtain $\frac{\partial f}{\partial x_i} \in \mathbb{R}^q$, one has to use the tangent e_i: to compute the gradient p passes are necessary. As a consequence, when $p \gg q$ forward mode is prohibitive. Even though recent work seems promising [Baydin et al., 2022], implementations by [Belouze, 2022] on Adsl did not perform well in our experiment context (see Section III.3).



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

I.4.4. 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 $\overline{P}$:

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

that joins the forward pass $\vec{P} = [s_i \quad \text{for} \quad i = 0 \dots n \quad \text{if} \quad s_i \neq Return]$ 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 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 $\overline{f} = \frac{\partial f}{\partial f} = 1$.

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

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 [Innes, 2018] are not SA and rely on adjoint accumulation. Even though our approach increases the number of variables, it simplifies the compilation progress. Moreover, the differentiation of a tupling statement is the foundation of the gradient estimator presented in Section IV.4.1.

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 call functions can be found via the chain rule and the basic derivative formula. We give a list in Table I.1 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.1.: Adjoint of calls

Adjoint of 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 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

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

The Scan adjoint is implemented in the JAX [Bradbury et al., 2018] automatic differentiation library but is barely documented. We provide a comprehensive way to formalize the adjoint of such operator. If one apply reverse mode automatic differentiation on the unrolled trace of the scan, it 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:

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

with

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

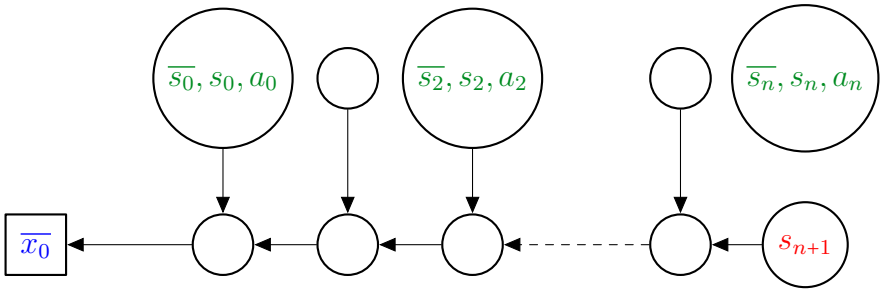


Figure I.16.: Adjoint of Scan Representation

The adjoint of a Scan is thus another Scan, Figure I.16 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.11:

$$\langle For(\tau \quad \chi \quad false \quad S \quad P \quad \Xi)\rangle \tag{I.11}$$

with

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

Its adjoint is the constructed Adsl loop I.12:

$$\overline{\langle For(\tau \quad \chi \quad false \quad S \quad P \quad \Xi)\rangle} = \langle For(\tau \quad \overline{\Xi} \quad true \quad \overline{S} \quad \overline{P} \quad \overline{\chi})\rangle \tag{I.12}$$

with

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

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.17. Thus we have hard-coded their gradient to be 1.

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.

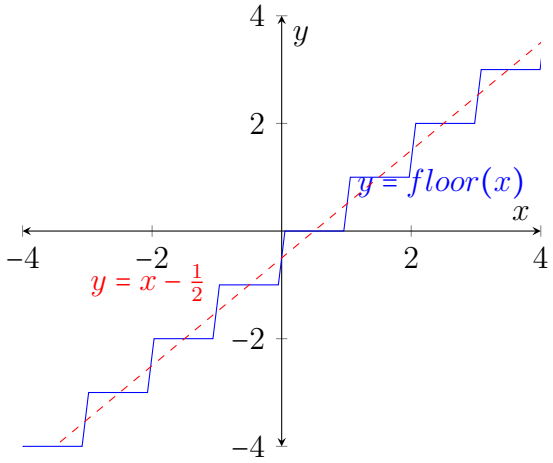


Figure I.17.: The floor function rounds down its input to the nearest integer. Its derivative is zero almost everywhere while its general trend is linear.

Adjoints 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 the following Adsl Program:

$$
\begin{array}{llllll}
\langle v & \leftarrow & Param & 0 & \rangle \\
\langle w & \leftarrow & Param & 1 & \rangle \\
\langle z & \leftarrow & Call & random.normal & v & w & \rangle \\
\langle Return & z & \rangle
\end{array}
$$

Even though we do not have a appropriate adjoint for *random.normal*, we can apply the reparameterization trick from Equations I.13 and I.14, which rely on a rewriting of the gaussian random variable.

$$\frac{\partial.}{\partial\mu}\mathcal{N}(\mu,\sigma) = \frac{\partial.}{\partial\mu}[\mu + \sigma\mathcal{N}(0,1)] = 1 \tag{I.13}$$

$$\frac{\partial.}{\partial\sigma}\mathcal{N}(\mu,\sigma) = \frac{\partial.}{\partial\sigma}[\mu + \sigma\mathcal{N}(0,1)] = \mathcal{N}(0,1). \tag{I.14}$$

In Adsl, we need to rewrite the statements in the following form:

$$
\begin{array}{llllll}
\langle v & \leftarrow & Param & 0 & \rangle \\
\langle w & \leftarrow & Param & 1 & \rangle \\
\langle zero & \leftarrow & 0.0f & \rangle \\
\langle one & \leftarrow & 1.0f & \rangle \\
\langle n & \leftarrow & Call & random.normal & zero & one & \rangle \\
\langle m & \leftarrow & Call & mul & n & w & \rangle \\
\langle z & \leftarrow & Call & mul & v & m & \rangle \\
\langle Return & z & \rangle
\end{array}
$$

With this formulation, the differentiation of these statements is thus 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* ...

Adjoints of predicates conditional

In the previous Section **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.18. 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 [Williams, 2004, Sutton et al., 2000, Schulman et al., 2017].

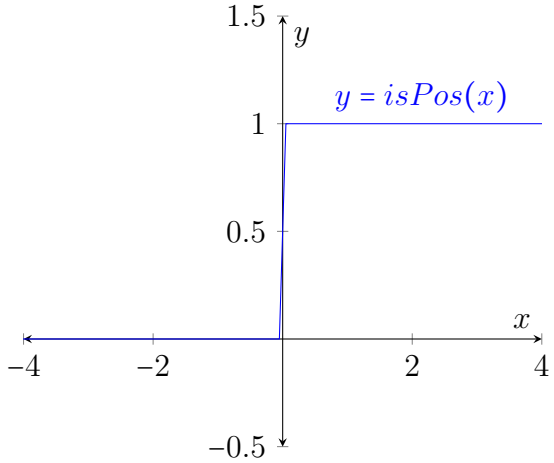


Figure I.18.: $isPos$ function

Adsl closure by differentiation

All the previous adjoint are built with other Adsl statements: the derivative of an Adsl program is another Adsl program. This property 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: [Mari et al., 2021] [Cerezo and Coles, 2021]. 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 on relational programming languages where query optimization is an extensive subject [Trummer and Koch, 2016] [Parsana and Atkotiya, 2019]. 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 AS systems relying on this mode.

I.5. Implementation

In this section we present the results of the implementation 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 scenario.

There is an open version of Envision [2] and its documentation is public[3].

At a higher level it is a Python-like implementation of SQL. The main specificity and advantage 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 point is that all these relationships are known at compilation time and the user do not need to wait for the execution errors to adapt its query. On the opposite, SQL implementations create many intermediate tables on the fly, every Envision table is reified and directly queryable.

In Listing I.7 we present how the Upstream table can be created from the Observations table thanks to a simple Aggregation of one of its non primary key, 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.

```
/// Table creation:
table Upstream[Category] = by Observations.Category

Upstream.Y = ...

/// TOTAL JOIN broadcast:
```

[2] https://try.testing.lokad.com/
[3] https://docs.lokad.com

```
Observations.Y = Upstream.Y
```

Listing I.7: broadcasts in Envision

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

In Envision, Differentiable Programming is a first class citizen. By integrating automatic differentiation in the Envision compiler itself, 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 [Wei et al., 2020] 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. Any mathematical operation is supported in the optimization blocks. For the relational part, the only limitation is to satisfy the PolyStar of Definition 6 form for the used tables.

In the following we present two implementations of differentiable programming through Envision. The first one is a toy example that introduces an illustrating model. The second one is a model daily used at Lokad to forecast demand.

I.5.3. Toy Example: the relational linear regression

The following example is very simple and aims to illustrate differentiable programming on relational programming languages.

We present a simple example of a categorical model, which is the relational linear regression. A traditional linear regression predicts a numerical quantity using a numerical input and two parameters, namely the slope and the intercept. The relational linear regression extends it with the slope being shared among all categories of a categorical feature. The intercept remains shared among all observations.

$$\hat{y}(cat, x) = a_{cat} \times x + b. \tag{I.15}$$

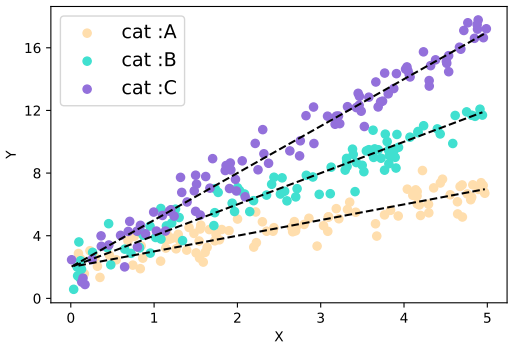


Figure I.19.: Relational linear regression applied with the *cat* attribute made of three different values A, B or C.

A linear regression has 2 parameters, while a relational linear regression has $1 + n_s$ parameters, with n_s the cardinality of the categorical feature. Instead of only modeling the relationship between two variables, it utilizes the underlying structure of the data to provide a more accurate representation of the features relationship.

Equation I.15 formalizes it while the Listing I.8 implements the toy model in Envision. The *autodiff* keyword opens the automatic differentiation block code. This example was presented in [Peseux, 2021] applied on the highly categorical dataset of Chicago taxi rides [of Chicago, 2022].

```
autodiff Observations with
    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

    return error^2
```

Listing I.8: autodiff block in Envision of the toy example

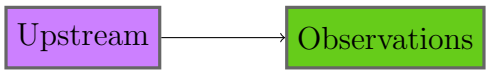


Figure I.20.: PolyStar of the toy example. By construction in the Listing I.7, the primary key of the Upstream table is a foreign key of the Observations table, which is a sufficient condition to allow the depicted broadcast.

Figure I.19 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.

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.20. The mathematical part is the formula of the raw linear regression, and the choice of the l^2 norm.

I.5.4. Production Example

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

We have successfully deployed to production such categorical model at Lokad in order to weekly forecast sales of Celio, a large retail company. The dataset is open sourced (with anonymization) and presented [4]. The dataset contains 3 years of history and concerns $100k$ different items. The dataset contains multiple categorical features for each item. The objective is to forecast sales at the item level. The implemented categorical model is similar[5] to the following:

$$\begin{aligned}\hat{y}(item, week) = \quad & \theta_{store(item)} \times \theta_{color(item)} \times \theta_{size(item)} \times \\ & \Theta[group(item), WeekNumber(week)].\end{aligned} \tag{I.16}$$

$\Theta[group(item), WeekNumber(week)]$ is a parameter vector that can be seen as a function that aims to capture the annual seasonality for a given group of items:

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

Even though notations are quite clear, one can consult Section III.1.2 for further precision on one-hot encoding. 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.22. 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 Figure I.21.

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.9.

[4] soon

[5] we do not disclose the actual model for confidentiality reasons.

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

Figure I.21.: Adsl code of the categorical model depicted in Equation I.16. 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.

```
autodiff Items epochs:10 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.thetaSeason = GroupWN.theta

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

    /// 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.9: 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.

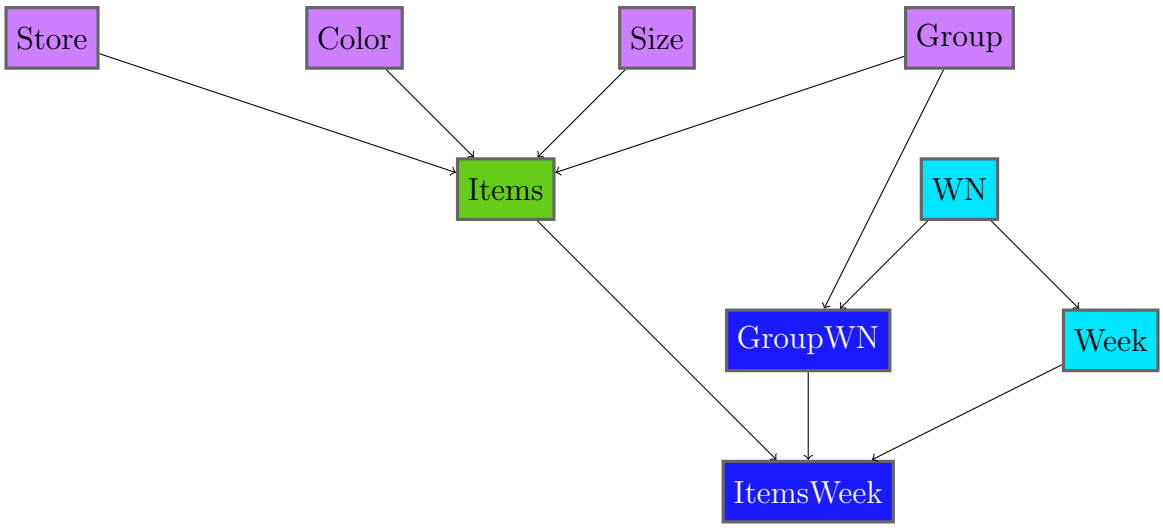


Figure I.22.: PolyStar from Celio's implementation. Items is the observation table and all the other tables are defined by their relationship with it. The upstream tables broadcast into Items as their primary keys are foreign keys for Items.

In order to retrieve the *color* parameter related to a specific item, one can use the TOTAL JOIN operator as presented in Listing I.10.

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

Listing I.10: 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.

I.5.5. Mathematical insights

on multiplicative models

Such model is multiplicative, we have tested other forms like the additive one inspired by the Prophet library [Taylor and Letham, 2017]. 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.4.

on adaptive optimizers

Optimizers are properly introduced in Section II.3
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:

$$amp \leq 2 \times \alpha \times updates. \tag{I.17}$$

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:

$$\begin{aligned}\hat{y}'(item, week) = \quad & e^{\theta'_{store(item)}} \times e^{\theta'_{color(item)}} \times e^{\theta'_{size(item)}} \times \\ & \Theta[group(item), WeekNumber(week)].\end{aligned}$$

I.5.6. 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.11.

```
autodiff Items epochs:10 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.thetaSeason = GroupWN.theta

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

    /// 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.11: 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.

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

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, Stochastic Gradient Descent (SGD) [Robbins and Monro, 1951], 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 version only requires a small subset of the data to be loaded into memory at a time.
This chapter aims to provide a comprehensive understanding of SGD and its practical applications in training machine learning models. The chapter also presents a unified view of the adaptive optimizers introduced by [Defossez et al., 2020] and their application to SGD. The proposed approach provides convergence guarantees for a non-convex function, relying on the unbiasedness of the estimator. This framework will be used as the theoretical foundation for chapters III and IV.
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 some of the best optimizers for performing in this context and how they can be unified to give convergence guarantees under other regularity conditions.
The main purpose of this chapter is to provide a comprehensive overview of the current understanding of the SGD approach from a theoretical perspective. This chapter serves as the foundation for the following two chapters, which introduce novel gradient estimators.

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 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

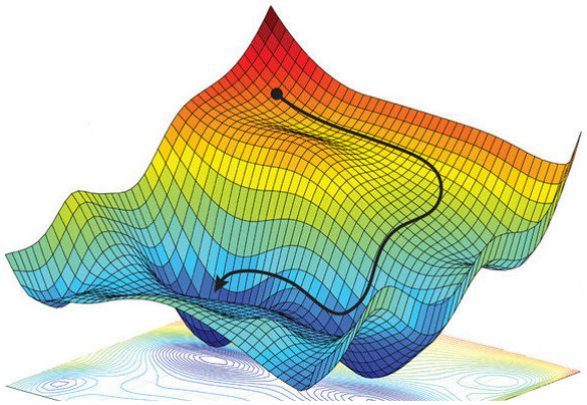


Figure II.1.: Gradient descent illustration on a representation of an arbitrary function $f : \mathbb{R}^2 \longrightarrow \mathbb{R}$. The parameter update following gradient descent allow the diminution of the objective function, as depicted by the decrease in the vertical location of $(x, y, f(x, y))$. Illustration directly taken from [Amini et al., 2018]

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. 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:

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

A comprehensive representation of gradient descent is thus given in Figure II.1.
All the work presented in Chapter I provides an automatic means to access the gradient of functions computed in relational programming languages. Section I.2.4 presented automatic differentiation systems on other environments like Pytorch [Paszke et al., 2019] or Julia [van Merrienboer et al., 2018], 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.

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. For this section , we suppose that F_θ is smooth and convex.
F_θ is L-smooth means that

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

in this set up, $\alpha_t = \frac{1}{L}$ gives us convergence properties.

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

$$F(\theta_t) - F(\theta^\star) \leq L \frac{\|\theta_0 - \theta^\star\|^2}{t}. \tag{II.3}$$

Proof. By L-smoothness of F_θ

$$\begin{aligned} F(\theta_{t+1}) - F(\theta^t) &\leq F(\theta_t - \frac{1}{L} \nabla_\theta F(\theta_t)) - F(\theta^\star) \\ &\leq -\frac{1}{L} \|\nabla F(\theta_t\|^2 + \frac{L}{2L^2} \|\nabla F(\theta_t\|^2 \\ &\leq -\frac{1}{2L} \|\nabla F(\theta_t\|^2 . \end{aligned}$$

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

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

Convexity of F gives us:

$$\Delta_t \leq \nabla F(\theta_t)(\theta_t - \theta^\star) \leq \|\nabla F(\theta_t)\| \, \|(\theta_t - \theta^\star)\| . \tag{II.5}$$

II.4 and II.5 gives:

$$\begin{aligned} \Delta_{t+1} &\leq \Delta_t - \frac{\Delta_t^2}{2L \|\theta_t - \theta^\star\|} \\ \Delta_{t+1} &\leq \Delta_t - \frac{\Delta_t^2}{2L \|\theta_1 - \theta^\star\|} . \end{aligned}$$

We get the final result by induction:

$$F(\theta_t) - F(\theta^\star) \leq L \frac{\|\theta_0 - \theta^\star\|^2}{t}.$$

□

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. In Section II.4, we will explore a non-convex setting.

II.1.4. Limitations

There are certain optimization problems that cannot be solved by gradient descent alone. Discrete optimization problems, such as those involving combinatorial or mixed-integer programming, are not suitable for gradient descent as it is a continuous optimization algorithm. In mixed integer programming, the decision variables can only take on integer values, which makes the problem inherently discrete. Gradient descent requires the objective function to be continuous and differentiable, which is not the case for mixed integer programming problems. Additionally, the search space for mixed integer programming problems is non-convex, making it difficult for gradient descent to find the global optimum. One way could be to convert the discrete optimization problem into a continuous one. However, this approach can introduce several disadvantages. First, it can result in a loss of accuracy, as the discretization error is not taken into account. Second, it may lead to an increased computational complexity, as the number of variables and constraints may increase significantly. Third, it can result in an infeasible solution, as the continuous relaxation may not satisfy the original problem constraints. Fourth, the solution obtained may not be an integer, which is often required for practical applications. Finally, the continuous solution may not be easily interpretable, which can make it difficult to understand the underlying problem and make decisions based on the solution. Overall, while converting a discrete optimization problem into a continuous one may offer some benefits, it is important to consider the potential drawbacks before choosing this approach. Therefore, specialized algorithms, such as branch and bound [Land and Doig, 2010] or branch and cut, are required to solve mixed integer programming problems.

II.2. Stochasticity

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

II.2.1. Motivation

Lets consider the minimization problems presented in the first Chapter. Our objective is to minimize the sum of a vector present in the observation table of a PolyStar:

$$G_{SUM(loss)}(\Pi_{loss}(Observation)) \tag{II.6}$$

It means that the minimization problem can be written as

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

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. Attempting to load the full dataset and compute the gradient for each observation for a single gradient descent step is impractical due to the memory constraints. 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:

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

In the following, we describe how to construct such estimators and under which assumptions we still have convergence properties.

Observation

Given that the objective function is expressed as a sum over an observation table in Equation II.6, 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 burden. 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 defined the linearized computational graph of f_θ, in which the nodes represent intermediate variables and the edges represent the mathematical operations. It is 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. Examples are given in Figures I.13 and I.14.
Our objective is to compute 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 [Bauer, 1974] 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.2 and Equation II.7 with f_θ the function to minimize with respect to θ.

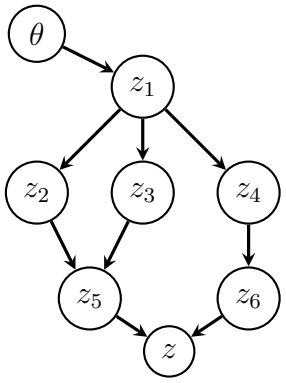
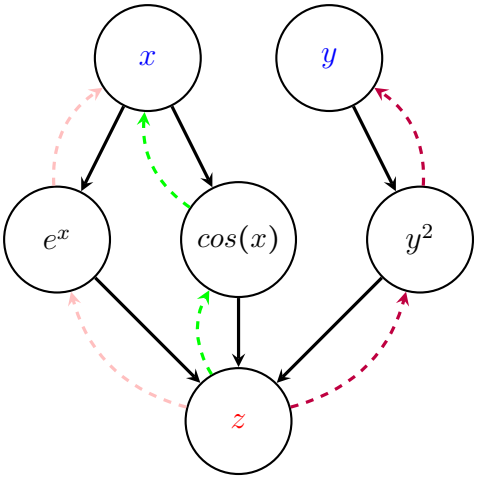


Figure II.2.: Example of the LCG of an objective function f_θ from the parameter θ to the output node z.

$$\nabla_\theta f = \sum_{\theta \longrightarrow z} \prod_{z_k \longrightarrow z_l} \frac{\partial z_l}{\partial z_k}. \tag{II.7}$$

$z_k \longrightarrow z_l$ 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 paths 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.3.



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

Figure II.3.: LCG of $f_2(x, y) = y^2 e^x \cos x$. The dashed lines represent backpropagation

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 [Oktay et al., 2020] and is generalized in Chapter IV.

II.2.2. Gradient stochasticity applied on relational data and PolyStar

First, we develop how the stochasticity obtained from the observations apply in our case. 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 stochastic gradient descent by partitioning the dataset using the rows of the observation table. 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 Section III.3.3, utilizing the same tables as the PolyStar defined in Figure I.22.

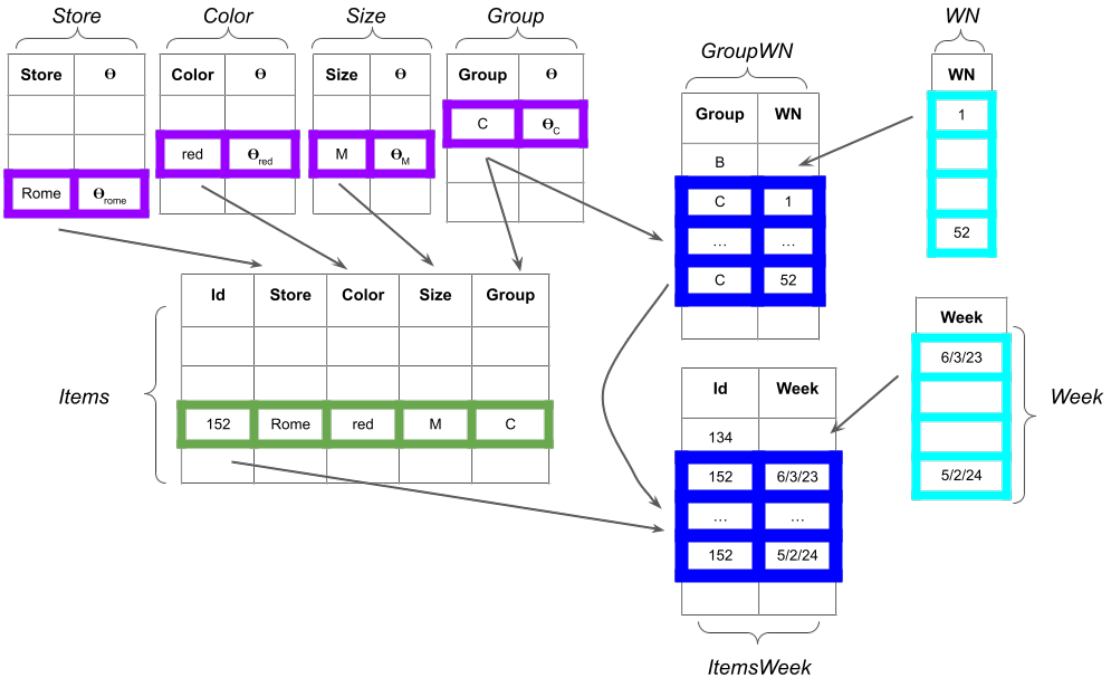


Figure II.4.: Data loading from the Celio dataset with model from Equation I.16. For each line of the Items table, one can access the corresponding line of the upstreams tables (Store, Color, Size, Group). The Full tables WN and Week are fully loaded, while the upstream-cross table GroupWN and observation-cross ItemsWeek are partly loaded.

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 relationship 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.4. By doing so, the execution of stochastic gradient descent is accelerated, as data handling consumes a non-negligible portion of computing resources, as detailed in Section I.2.5. This perspective emphasizes the necessity of introducing a differential layer in relational programming languages, as accomplished in Chapter I.
Regarding the function stochasticity of the Celio model, the formal representation of the model under a LCG is given in Figure II.5. 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.

II.3. Optimizers

An optimizer is essentially an algorithm that iteratively adjusts the model's parameters to minimize the objective function. It takes as input the parameter θ_t, the computed gradient and possibly other parameters and that outputs a new value of θ_{t+1}.

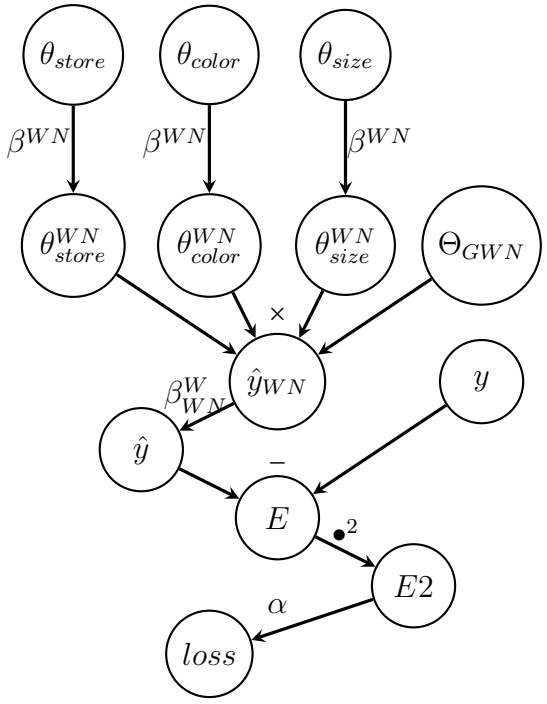


Figure II.5.: LCG of the Celio model presented in Equation I.16. Its is an alternative representation of the Adsl statements of Figure I.21. The parameters nodes are θ_{store}, θ_{color}, θ_{size} and Θ_{GWN} whereas $loss$ is the output node. In this model, there are no duplicated variables, which means that there is one and only one path from a parameter to the output node.

It thus plays a crucial role in the minimization process, as it is responsible for updating the model's parameters based on the gradient of the objective function.
It uses the gradient of the objective function with respect to the model's parameters to determine the direction of the 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 choice of optimizer 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. These optimizers are presented in chronological order of their historical development.

II.3.1. Optimizers

Vanilla

Vanilla 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. Addi-

tionally, 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 trial and error. 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.

Adagrad

Adagrad [Duchi et al., 2011] is an optimization algorithm for gradient-based optimization that adapts the learning rate for each parameter based on the historical gradients. 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 have a smaller learning rate and parameters that have small gradients will have 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:

$$\theta_{t+1} = \theta_t - \frac{\alpha g_t}{\sqrt{\epsilon + \sum {g_t}^2}}$$

The 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 learning rate continues to decrease as the sum of squares of the past gradients grows larger, which can cause the learning rate to become too small, making it difficult for the optimizer to escape from local minima.

Adam

Adam was introduced by [Kingma and Ba, 2014]. One of the main contribution of this optimizer is the introduction of momentum m_t and r_t estimation of the gradient. It relies on the momentum by using the moving average of the gradient instead of the computed gradient.

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

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

$$
\lim_{g_t \to \infty} \frac{\hat{m}_{t+1}}{\sqrt{\epsilon + \hat{v}_{t+1}}} = \frac{1-\beta_1}{1-\beta_1^{t+1}} g_t \times \sqrt{\frac{1-\beta_2^{t+1}}{(1-\beta_2) g_t^2}} = sign(g_t) \frac{1-\beta_1}{\sqrt{1-\beta_2}} \frac{1-\beta_1^{t+1}}{\sqrt{1-\beta_2^{t+1}}} < 1 \tag{II.8}
$$

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 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.
However, recent research [Defossez et al., 2020] suggests that AdaGrad and Adam optimizers are essentially two variants of the same optimizer. This optimizer and convergence guarantees using it is presented in the following.

II.4. Convergence guarantees with adaptive optimizers

Theorem 2 ensures the convergence of the gradient descent in the smooth and convex case but it does not give guarantees if we use *stochastic* gradient descent. 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 [Defossez et al., 2020].
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 estimate 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 $(f_t)_{t \in \mathbb{N}^*}$. We de-

note $\mathbb{E}_{n-1}[.]$ 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.
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:

$$\forall \theta \in \mathbb{R}^p; \quad F(\theta) \geq B, \tag{II.9}$$

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:

$$\forall \theta \in \mathbb{R}^p; \quad \|\nabla f(\theta)\|_\infty \leq M \tag{II.10}$$

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

$$\forall \quad \theta, \theta'; \quad \|\nabla F(\theta) - \nabla F(\theta')\|_2 \leq L \|\theta - \theta'\|_2 . \tag{II.11}$$

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.12 introduced by [Defossez et al., 2020], which is a generalization of adaptive optimizers into a single one.

$$\begin{aligned} m_{t+1} &= \beta_1 m_t + g_t \\ r_{t+1} &= \beta_2 r_t + g_t^2 \\ \theta_{t+1} &= \theta_t - \alpha_t \frac{m_{t+1}}{\sqrt{r_{t+1} + \epsilon}}, \end{aligned} \tag{II.12}$$

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.

Theorem 3 (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:

$$\mathbb{E}_{t-1}[g^2] \leq \tilde{v}_\epsilon, \tag{II.13}$$

and

$$g^2 \leq v_\epsilon. \tag{II.14}$$

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

$$\mathbb{E}_{t-1}[\frac{Gg}{\sqrt{v_\epsilon}}] \geq \frac{G^2}{\sqrt{\tilde{v}_\epsilon}} - 2R\mathbb{E}_{t-1}[\frac{g^2}{v_\epsilon}]. \tag{II.15}$$

Proof. By linearity

$$\mathbb{E}_{t-1}[\frac{Gg}{\sqrt{v_\epsilon}}] = \mathbb{E}_{t-1}[\frac{Gg}{\sqrt{\tilde{v}_\epsilon}}] + \mathbb{E}_{t-1}[Gg(\frac{1}{\sqrt{v_\epsilon}} - \frac{1}{\sqrt{\tilde{v}_\epsilon}})]. \tag{II.16}$$

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 $x > 0$:

$$ab \leq \frac{a^2}{2}x + \frac{b^2}{2x}, \tag{II.17}$$

so lets apply Inequality II.17 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.13:

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

Now lets apply Inequality II.17 on A_2 with

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

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.13:

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

Summing the inequalities gives:

$$\mathbb{E}_{t-1}[|\, Gg(\frac{1}{\sqrt{v_\epsilon}} - \frac{1}{\sqrt{\tilde{v}_\epsilon}}) \,|] \leq \mathbb{E}_{t-1}[|\, A_1 \,| + |\, A_1 \,|] \leq \frac{G^2}{2\sqrt{\tilde{v}_\epsilon}} + 2R\mathbb{E}_{t-1}[\frac{g^2}{v_\epsilon}], \tag{II.18}$$

II.16 and II.18 can thus complete the lemma proof:

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

□

Proof. (of Theorem 2)
Lets start with the smoothness of F between θ_{t+1} and θ_t:

$$F(\theta_t) \leq F(\theta_{t-1}) - \alpha\langle G. \frac{g}{\sqrt{v_\epsilon}}\rangle + \alpha^2 \frac{L}{2} \left\| \frac{g}{\sqrt{v_\epsilon}} \right\|_2^2, \tag{II.19}$$

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

$$\mathbb{E}_{t-1}[F(\theta_t] \leq F(\theta_{t-1}) - \alpha\mathbb{E}_{t-1}[\langle G. \frac{g}{\sqrt{v_\epsilon}}\rangle] + \alpha^2 \frac{L}{2} \mathbb{E}_{t-1}[\left\| \frac{g}{\sqrt{v_\epsilon}} \right\|_2^2]. \tag{II.20}$$

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

$$\mathbb{E}_{t-1}[F(\theta_t] \leq F(\theta_{t-1}) - \frac{\alpha}{2R\sqrt{t}} \|G\|_2^2 + (2\alpha R + \frac{\alpha^2 L}{2}) \mathbb{E}_{t-1}[\left\| \frac{g}{\sqrt{v_\epsilon}} \right\|_2^2], \tag{II.21}$$

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

$$\mathbb{E}[F(\theta_T]) \leq F(\theta_0) - \frac{\alpha}{2R\sqrt{T}} \sum_{t=0}^{T-1} \mathbb{E}[\left\| \nabla F(\theta_t)_2^2 \right\|] + (2\alpha R + \frac{\alpha^2 L}{2}) \sum_{t=0}^{T-1} \mathbb{E}[\left\| \frac{g}{\sqrt{v_\epsilon}} \right\|_2^2]. \tag{II.22}$$

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

$$\frac{a}{b} \leq \ln(b) - \ln(a), \tag{II.23}$$

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:

$$\sum_{t=0}^{T} \frac{\nabla f_t(\theta_{t-1})^2}{\epsilon + v_t} \leq \ln(1 + \frac{v_T}{\epsilon}). \tag{II.24}$$

Using it in Equation II.22 gives:

$$\mathbb{E}[F(\theta_T]) \leq F(\theta_0) - \frac{\alpha}{2R\sqrt{T}} \sum_{t=0}^{T-1} \mathbb{E}[\left\|\nabla F(\theta_t)_2^2\right\|] + (2\alpha R + \frac{\alpha^2 L}{2}) p \ln(1 + \frac{TR^2}{\epsilon}). \tag{II.25}$$

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) 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.

Conclusion

In conclusion, this chapter has provided a comprehensive understanding of stochastic gradient descent 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 sthochatsity 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 stochastic gradient descent, 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 stochastic gradient descent. 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 [Peseux et al., 2022].

Introduction

Tabular data represents 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 [Chen and Guestrin, 2016, Ostroumova et al., 2018] 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 [Gorishniy et al., 2021]. Some deep learning approaches [Popov et al., 2020, Frosst and Hinton, 2017] [Luo et al., 2021] 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 [Borisov et al., 2021].
One of the possible explanation is that deep learning excels on homogeneous data where embeddings can be learned [Rehman, 2021, Gupta and Agrawal, 2022] via stochastic gradient descent 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. 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. 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. 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 and pertains to the

supply chain, with only a few publicly available datasets in this 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 stochastic gradient descent 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 [Borisov et al., 2021] and [Gorishniy et al., 2021], 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 [Mathov et al., 2022]. Even for simpler tasks such as binary or multiclass classification and regression, deep learning did not surpass yet tree models on tabular data yet as presented in [Shwartz-Ziv and Armon, 2021] [Borisov et al., 2021]. Tabular data is depicted as the last "unconquered castle" by [Kadra et al., 2021] and multiple works asses the crucial need of further development in this direction [Fayaz et al., 2022]. This holds even though various architectures such as MLP, ResNet, Transformer, NET-DNF ... have been applied to them, as soon as the dataset is actually categorical, i.e. it has mainly nominal attributes [Hayashi, 2020].
We 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 [Borisov et al., 2022], 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 [Arik and Pfister, 2021] [Song et al., 2019] [Popov et al., 2020].
All these attempts did not outperform the standards models such as XGBoost from [Chen and Guestrin, 2016] or CatBoost from [Ostroumova et al., 2018] which are still the state-of-the-art in this domain [Borisov et al., 2021]. The evaluation is performed on the Adult Census Income (ACI) dataset [Kohavi, 1997], which is frequently utilized to showcase the handling of categorical data. deep learning approaches assume that every input feature is relevant to every observation, which can explain their disappointing results on categorical data. In deep learning architectures, all parameters are typically updated at every iteration except through the use of methods such as Dropout [Hinton et al., 2012] or LayerOut [Goutam et al., 2020] 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.

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
...	...	...	...
n	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 mathematical operations 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 the value. 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. 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 [Cheng et al., 2016].
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.

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

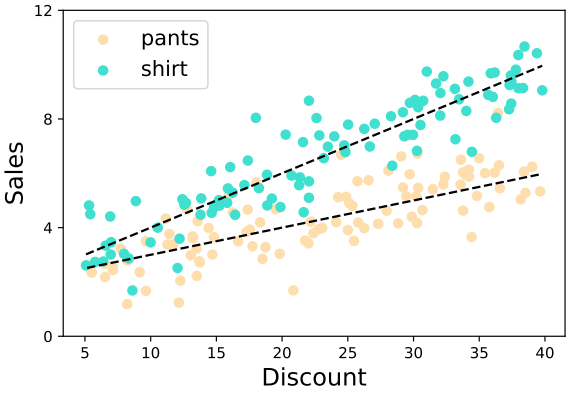


Figure III.1.: Relational linear regression.

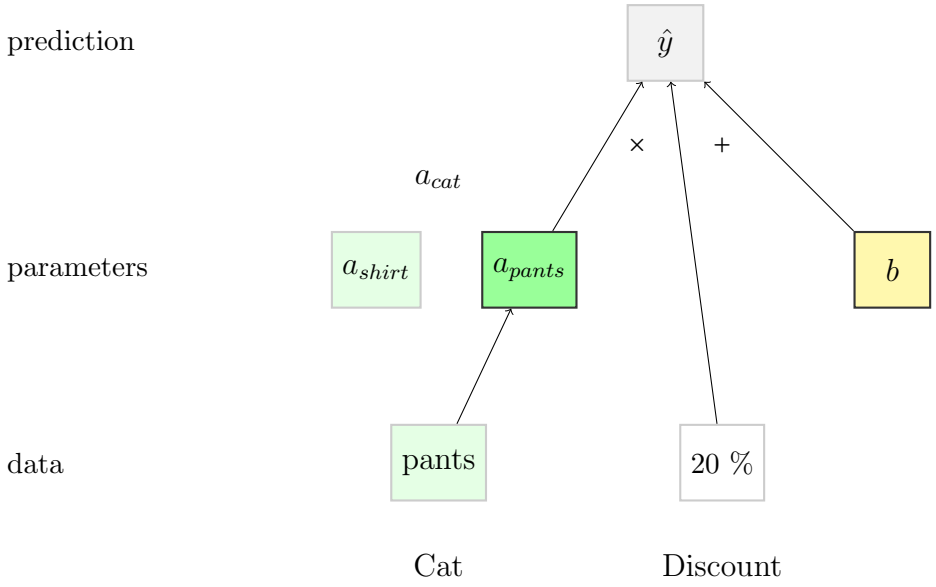
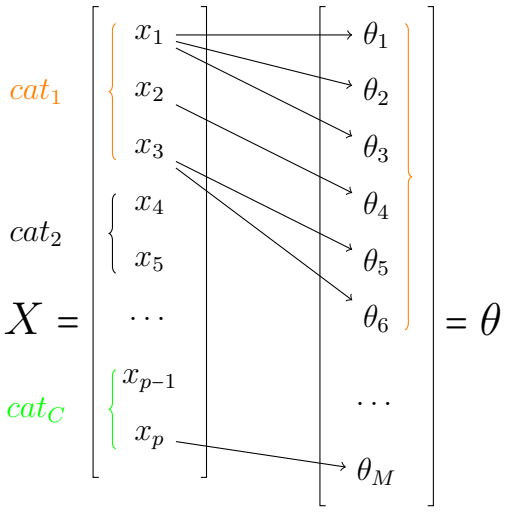
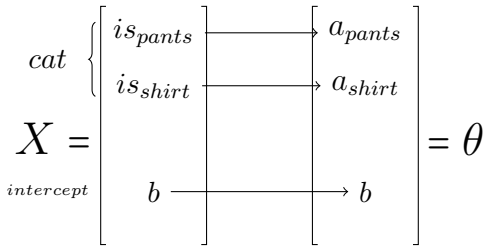


Figure III.2.: Relational linear regression accessing parameters. The slope parameters are not concerned by every observation: parameter a_{pants} is used only for the pants data.

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 stochastic gradient descent 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 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 [Rudin, 2019,Ribeiro et al., 2016]. 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 [Blanco-Justicia et al., 2020]. 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).



(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. The arrows serve to summarize the interpretability of the model.

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 [Chen, 2009]. 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 article, 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 stochastic gradient descent 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. Its specificity is to take into account the categorical features of the model.

III.1.3. Problem and 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 $\mathbf{C}_i \in \bigtimes_{c=1}^{C} A_c$ where each A_c is an alphabet, i.e. a finite set of $|\, A_c \,|$ symbols. Thanks to one-hot encoding, one can turn the attribute vectors $\mathbf{C}_i$ into numerical features $X_i \in \{0;1\}^p$ where $p = \sum_{c=1}^{C} |\, A_c \,|$. Let's define an arbitrary order on the union of all the alphabets: $\{s_k\}_{k \leq p} = \bigcup_c A_c$. Then:

$$\forall (C_i, \cdot) \in \mathcal{Z} \quad \forall k \leq p \quad \mathbf{C}_i = s_k \Leftrightarrow X_i^k = 1. \tag{III.2}$$

Figures III.3 illustrate the formal definition. In Figure III.3a, $\{\theta_i\}_{i=1..6}$ are related to the first feature cat_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. We aim to find the best parameter $\theta^\star \in \mathbb{R}^m$ $(m \geq p)$ to minimize the loss F_θ on the whole dataset.

$$f: \quad \Theta \times \mathcal{Z} \longrightarrow \mathbb{R} \qquad\qquad \theta^\star = \arg\min_\theta \quad F_\theta = \arg\min_\theta \sum_{X,y \in \mathcal{Z}} f_\theta(X,y)$$
$$\theta, (X,y) \longrightarrow f_\theta(X,y) \qquad\qquad = \arg\min_\theta \sum_{i=1\ldots n} f_\theta(X_i, y_i)$$

III.1.4. Gradient descent issues with categorical features

In classical stochastic gradient descent, an unbiased estimator of the gradient is used. Instead of computing the complete gradient on all observations, the observations are divided into *batches* and the gradient is estimated on them:

$$\nabla_\theta F = \frac{1}{n} \sum_{obs} \nabla_\theta f_{obs} = \frac{1}{n} \sum_{i \leq n} \nabla_\theta f(X_i, y_i) = \frac{1}{n} \sum_{batch} \sum_{obs \in batch} \nabla_\theta f_{obs}. \tag{III.3}$$

In large datasets, it is computationally expensive to compute the exact gradient. To address this, the gradient is estimated by observation batches. Gradient descent methods are effective in finding the minimum notably because the gradient estimator on a batch is unbiased, as proven by [Defossez et al., 2020]. 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 instance, the gradient $\nabla_{a_{pants}} F$ reduces to:

$$\nabla_{a_{pants}} F = \frac{1}{n} \sum_{obs} \nabla_{a_{pants}} f_{obs} = \frac{1}{n} \sum_{batch} \sum_{\substack{obs \in batch \\ cat(obs)=pants}} \nabla_{a_{pants}} f_{obs}. \tag{III.4}$$

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 μ_{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. This numerically zero gradient does not convey any information and it should not be used for parameters updates. This atomic property of categorical attributes is completely ignored when using standard SGD approaches. Notice, that this problem is further amplified among successive batches when some improved deep learning optimization operators are

introduced such as gradient with momentum. In this case the structural zeros of categorical data are broadcast among successive batches, leading to biased gradient estimation.

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

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 modified. The encoding of categorical data should not part of the gradient-exposed portion of the model and should not infer on the model's parameters updates.

III.1.5. Convergence guarantees of stochastic gradient descent

In the preceding chapter, a setting was presented which provided convergence guarantees while using gradient descent. It was explained that both the traditional linear regression, which minimizes the mean squared error, and its relational version satisfy the assumptions outlined in Section II.4. We have $F_{\vec{a},b} = \sum_{x_i,c_i,y_i}(a_{c_i}x_i + b - y_i)^2$ which is positive. Then for the second one the decomposition of the expected gradient $\nabla_\theta F$ as a sum satisfies it. Finally the third one is also verified, with m_x the maximum of all the x_i present in the dataset:

$$\forall \quad \vec{a}, b, \vec{a}', b'; \quad \frac{\partial F(\vec{a}, b)}{\partial b} - \frac{\partial F(\vec{a}', b')}{\partial b} = 2\sum_i (\vec{a}_{c_i} - \vec{a}'_{c_i})x_i + (b - b')$$
$$\|\nabla_b F(\vec{a}, b) - \nabla_b F(\vec{a}', b')\|_2 \leq 2nm_x \times (\|\vec{a} - \vec{a}'\|_2 + \|b - b\|_2).$$

The same logic applies with $\nabla_a F$. Thus the relational version of the linear regression converges if optimized with one version of the adaptive optimizers of [Defossez et al., 2020].

III.2. Solution: gradient estimator for categorical features

III.2.1. GCE definition

The problem with stochastic gradient descent 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. 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 a 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. It gives the following loss:

$$\tilde{F}_\theta = \frac{1}{p}\sum_{k=1}^{p} \sum_{X,y \in S_k} \frac{1}{\#S_k} f_\theta(X, y), \tag{III.6}$$

with this loss objective function, randomly sampling from $\mathcal{Z}$ and simply summing the gradients of the parameters 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 $\tilde{F}_{\binom{z}{m}\theta}$ from Equation III.7 properly does it.

$$\tilde{F}_{\binom{z}{m}\theta} = \frac{1}{p}\sum_{k=1}^{p} \sum_{X,y \in S_k \cap \binom{z}{m}} \frac{1}{\#[S_k \cap \binom{z}{m}]} f_\theta(X, y). \tag{III.7}$$

$\tilde{F}_{\binom{z}{m}\theta}$ is a random estimator of $\tilde{F}_\theta$ where m observations (over the $\mathcal{Z}$) are uniformly drawn. It is the gradient estimator for categorical features (GCE) used by Algorithm 1. This estimator is unbiased, proof can be found in the following Section III.2.2:

$$\mathbb{E}[\nabla \tilde{F}_{\binom{z}{m}}] = \nabla \tilde{F}_\theta.$$

This is a sufficient condition for convergence in the previously presented setting Section II.4 as soon as the target loss satisfies regularity conditions. The loss function depicted in Equation III.6 seems similar to 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 resumes to $\tilde{F}_\theta$:

$$\begin{aligned}
\tilde{F}_\theta &= \frac{1}{p}\sum_{k=1}^{p} \sum_{X,y \in s_k} \frac{1}{\#s_k} f_\theta(X, y) \\
&= \frac{1}{p}\sum_{k=1}^{p} \frac{1}{C} \sum_{X,y \in s_k} f_\theta(X, y) \\
&= \frac{1}{p \times C} \sum_{X,y \in \mathcal{Z}} f_\theta(X, y) \\
&= \frac{1}{\#\mathcal{Z}} \sum_{X,y \in \mathcal{Z}} f_\theta(X, y) \\
&= F_\theta,
\end{aligned}$$

as the p 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:

$$\theta_t = \theta_{t-1} - \alpha g_t.$$

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 shows 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 the Section III.3.

Algorithm 1 gradient estimator for categorical features

Require: $\mathcal{Z}$: data
Require: $update(\cdot,\cdot)$: chosen optimizer
Require: θ_0: Initial parameter vector
$t \leftarrow 0$
while θ_t not converged **do** $t \leftarrow t+1$
Divide Z in $Batches$
for batch ∈ Batches **do**
for symbol ∈ Alphabet **do**
$c_{symbol} \leftarrow 0$
end for
$\mathbf{g} \leftarrow \vec{0}$
for X, y ∈ batch **do**
$c_{symbol(X)} \leftarrow c_{symbol(X)} + 1$
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
$\theta_t \leftarrow \theta_{t-1}$
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
end for
end while

III.2.2. GCE unbiasedness proof

To properly prove that GCE is unbiased, we first need to prove that it is well defined. We first prove that one needs a finite time to before drawing a some observations related to a given 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. 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:

$$\mathbb{P}(\tilde{K} = 1) = 1 - (\frac{|Z| - |T|}{|Z|})^m = P_1 \tag{III.8}$$

$$\mathbb{P}(\tilde{K} = n) = (1 - P_1)^{n-1} P_1. \tag{III.9}$$

Theorem 4 (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:

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

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 4. *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

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

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

Remark 5. *Thanks to Theorem 4 we can ignore the first draws M_0 such as $M_0 \cap T = \varnothing$*

One notes

$$\begin{aligned} M_k &= (M_k \cap T) \sqcup (M_k \cap (Z \backslash T)) \\ &= (M_k^T) \sqcup (M_k \cap (Z \backslash T)), \end{aligned}$$

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

and

$$\bar{K} = \mid \{k \leq K \mid M_k^T \neq \varnothing\} \mid$$

Thanks to Remark 5 we have $\bar{K} \geq 1$. Then the proposed estimator is $\hat{a}$:

$$\hat{a} = \frac{1}{\bar{K}} \sum_{k=1}^{K} avg(M_k^T).$$

Theorem 5 (Unbiased estimator)**.** *$\hat{a}$ is an unbiased estimator of s_T*

Proof.

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

□

III.2.3. 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. Lets consider

a a batchsize of 3, so the first iteration concerns the 3 first line 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:

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

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. The difference between g_θ and $\tilde{g}_\theta$ have a bigger impact on the parameter updates when there are multiple categorical parameters.

III.3. Experimental and Results

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 public datasets listed in Table **??** as well as a private dataset from the supply chain domain for our evaluations. The chosen metrics for evaluating performance are the mean squared error (MSE) for regression tasks and error rate (i.e., 1 – *Accuracy*) for classification tasks.

Dataset	Chicago	ACI	compas	DGK	Forest Cover	KDD99	UsedCars
instances	194m	48k	7.2k	72k	15k	494k	38k
max cardinality	7.9k	42	341	1k	40	66	1.1k

Table III.2.: Datasets characteristics

It should be noted that during the training of the models using GCE, the corrected loss $\tilde{F}_\theta$ is utilized, while the standard metric is employed to evaluate the performance on the test dataset.

III.3.1. Deep Learning

In our study, we utilized PyTorch [Paszke et al., 2019] 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:
Adult Census Income (ACI) dataset [Kohavi, 1997] aims to predict the wealth status of individuals, Compas dataset predicts the likelihood of re-offending among criminal defendants, Forest Cover dataset [Blackard and Dean, 1999] predicts the forest cover type based on categorical characteristics of $30m^2$ forest cells, KDD99 dataset [Archive, 1999] aims at predicting cyber-attacks, Don't Get Kicked (DGK) dataset [Kaggle, 2022] predicts whether a car purchased at auction is a good or a bad buy. Used Cars datasets from Belarus is presented in III.3.2.
In order to only measure the impact of GCE, we only use those 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 [Gorishniy et al., 2021]. We have tested three different optimizers with their default settings: SGD (vanilla), AdaGrad and Adam. Tests have been run on several batch sizes: $2^{5...10}$. To record 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. 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. This 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, as reported in [Borisov et al., 2021]. However, 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 [Peseux, 2021] that provides access to the gradients of categorical models. Stochastic optimization using Adam and an relational linear regression were compared on two publicly available datasets: the Chicago Taxi ride dataset [of Chicago, 2022] and the Belarus used car dataset [Lepchenkov, 2019].
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

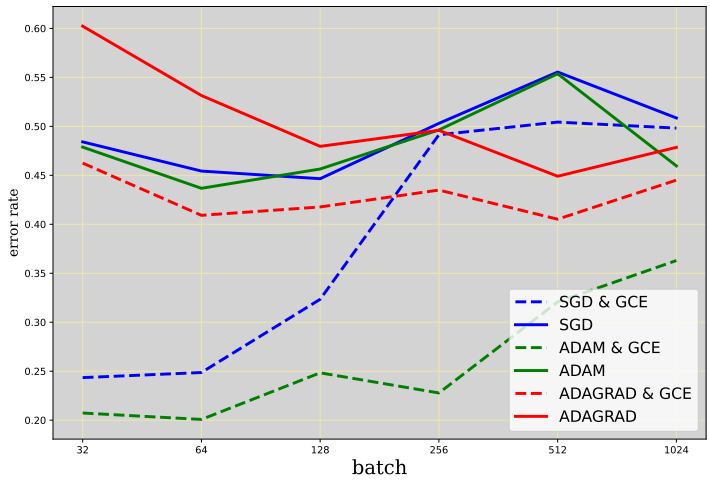


Figure III.4.: Results (error rate) on the ACI dataset with the ResNet-like network. The dashed curves represent experiments with **GCE** and show an improvement on the loss for every optimizer used.

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. It contains vehicle attributes. 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 seen in Table **??**, the relational batch performed better with our proposition based on Algorithm 1 with the following setting: 30 epochs ; optimizer Adam with default setting ; batch size of 1. Experiment was reproduced 20 times.

Dataset	Adam	Adam & GCE
Chicago Ride	35.58 ± 1.11	**9.45 ± 1.63**
Used Cars	7.10 ± 2.45	**0.08 ± 0.01**

Table III.3.: Results (RMSE) with categorical models

III.3.3. Categorical models on a real case

We applied GCE on the categorical model presented in Section I.5.4. It is a multiplicative model daily used on retail data. We employed Adam optimizer with its default values along with GCE and stochastic gradient descent for updating the parameters. The use of GCE results in a significant improvement in the performance of the categorical model as compared to the 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.4. 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 reel a and b, which proves that this difference cannot be properly controlled. However this is not harmful because II.4 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.

III.4. Categorical model initialisation

Appropriate initialization of a model is of utmost importance as it 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.

III.4.1. Two features example

Consider the multiplicative model presented in Section III.3.3 without the size vector. 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 captures the seasonality. Now, we focus on the initialization of θ_{store} and θ_{color}, which

are meant to be independent of the seasonality. Thus we would like

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

with $A^{s \times c} = \{item \mid store(item) = s \wedge color(item) = c\}$
Let's note 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 $A \in \mathbb{R}^{n_c \times n_s}$
Note that not every matrix A can be factorized this way, as $rank(\vec{c} \otimes \vec{s}) \leq n_c + n_s$
But let assume that such $\vec{c}, \vec{s}$ exist with $\|\vec{c}\| = 1$ Then

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

$\forall i \leq n_c \quad A.\vec{1} = (\sum_j v_j)\vec{u}$ then

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

Then $\vec{s}$ is obtained with $\exists i_1 \forall j \leq n_s \quad s_j = \frac{a_{i_1 j}}{c_{i_1}}$

III.4.2. Generalization to Singular Value Decomposition

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

$$M = U \Sigma V^\star$$

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.1).
By definition, Σ can be written:

$$\Sigma = \sum_i \sigma_i P_i.$$

Applying it to the SVD gives:

$$M = \sum_i \sigma_i U_i \otimes V_i. \tag{III.15}$$

The best approximation of the matrix M through an outer product is given by the term with the largest singular value, that is, $\sigma_1 U_1 \otimes V_1$.
The technique of matrix decomposition provides a good starting point for initializing the parameters of any multiplicative model, such as the one presented in Section I.5.4.
SVD can also be extended to higher-order tensors, i.e., arrays of data with more than two dimensions.
For a third-order tensor, SVD can be computed by unfolding the tensor along one of its modes to form a matrix, then applying SVD to that matrix. This will produce three matrices: two orthogonal matrices and a diagonal matrix. These three matrices correspond to the three modes of the original tensor, and can be used to reconstruct the

original tensor.
For higher-order tensors, the process is similar. The tensor is unfolded along one of its modes to form a matrix, and SVD is applied to that matrix to produce a set of orthogonal matrices and a diagonal matrix. These matrices correspond to the modes of the original tensor, and can be used to reconstruct the original tensor.
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.

Conclusions

This chapter focuses on the challenge of using stochastic gradient descent 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 dataset is made publicly available for further evaluation and study. 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.
The main contribution of this chapter is to shed light on the under-appreciation and neglect of categorical data in both public datasets and machine learning as a whole. Despite their widespread use in many key areas, such as health and supply chain, categorical data have not received adequate attention in the field. By highlighting this issue, we hope to inspire further research and encourage the development of methods that specifically address the unique challenges posed by categorical data. For example, one potential solution could be the use of GCE, which is specifically designed to handle this type of data. In sum, our aim is to bring categorical data to the forefront of machine learning research and spur the development of new techniques that better account for the specific requirements of these data.

IV. Gradient code stochasticity, overfitting and memory consumption

The work presented in this chapter is based on ***TODO***

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 enables the training of such models. 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.
In addition to the issue of resource consumption, overfitting to 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 [Hinton et al., 2012]. Dropout involves temporarily turning off certain nodes in the execution graph.
To address the issue of memory consumption without relying on checkpointing, a novel gradient estimator was proposed in [Oktay et al., 2020]. This is called Randomized Automatic Differentiation (RAD). RAD, which is unbiased, is constructed by drawing random paths through the backpropagation execution graph, effectively acting as back-propagation dropout. The unbiasedness of the estimator is crucial as it ensures that the convergence properties of the model are preserved. The use of a uniform distribution for drawing random paths is just one possibility, and other distributions may lead to better learning results. The "best" distribution is dependent on the learning stage and can be optimized through heuristics that emphasize the most important paths for the gradient. 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 outperform 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 sweep $\vec{P}$ followed by a backward sweep $\overleftarrow{P}$. The backward sweep relies on values computed during the forward sweep, and the choice must be made between storing them or recomputing them. Checkpointing is a technique that balances time and memory usage during the execution of an adjoint program.
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, not all values are stored, but are instead recomputed from the most recent checkpoint when needed.

$$
\begin{aligned}
x, y &\longleftarrow \dots \qquad // \quad \textit{Checkpoint?} \\
z &\longleftarrow x \times y \\
&\dots \\
\overline{z} &\longleftarrow \dots \\
\overline{x} &\longleftarrow \overline{z} \times y \\
\overline{y} &\longleftarrow \overline{z} \times x
\end{aligned}
$$

Listing IV.1.: Reverse mode automatic differentiation and checkpointing

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

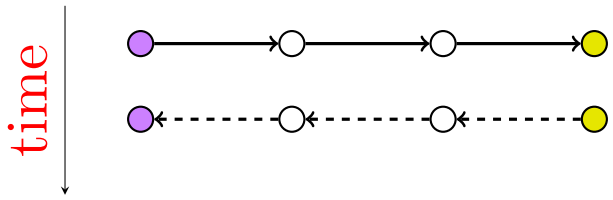


Figure IV.1.: STORE-ALL. The forward path are represented as the full lines while the dashed ones represent the backpropagation. All intermediate variables are stored, enabling the direct computation of the backward pass at the cost of increased memory requirements.

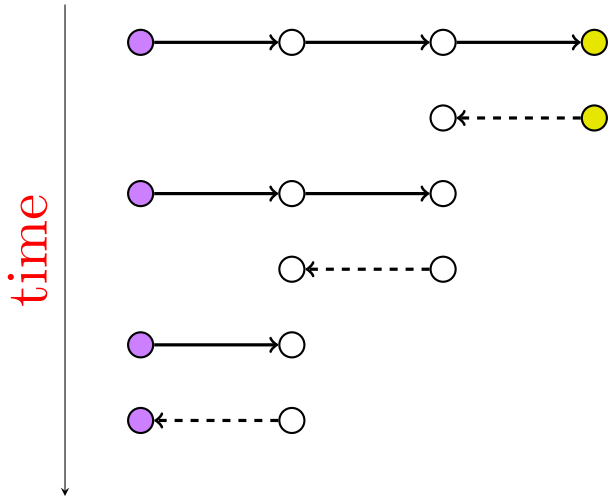


Figure IV.2.: RECOMPUTE-ALL. No intermediate variables are saved, one needs to recompute each one of them from the start (in purple) to determine the adjoint.

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 [Griewank, 1994], 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* and *recompute* solutions may satisfy the SA property. If the *store* 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. Moreover, the white box models enabled by differentiable programming on relational programming languages are small in size when considering the observation table level. 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 and writes the vector W. 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 [Teikari et al., 2018, Cunneen et al., 2019]. 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 [Badue et al., 2021]. 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 [Lahmeri et al., 2021]. 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, overfitting is a common problem in machine learning, where the model learns to fit the training data too closely, resulting in poor performance on new, unseen data. Essentially, the model becomes too complex, and instead of generalizing patterns, it simply memorizes the examples from the training set. This can occur when a model is trained on a limited or biased dataset, or when the model is too complex relative to the amount of data available. As a result, the model may pick up on noise or random variations in the training data that are not actually useful for making predictions on new data. A graphical illustration of overfitting is given in Figure IV.3.

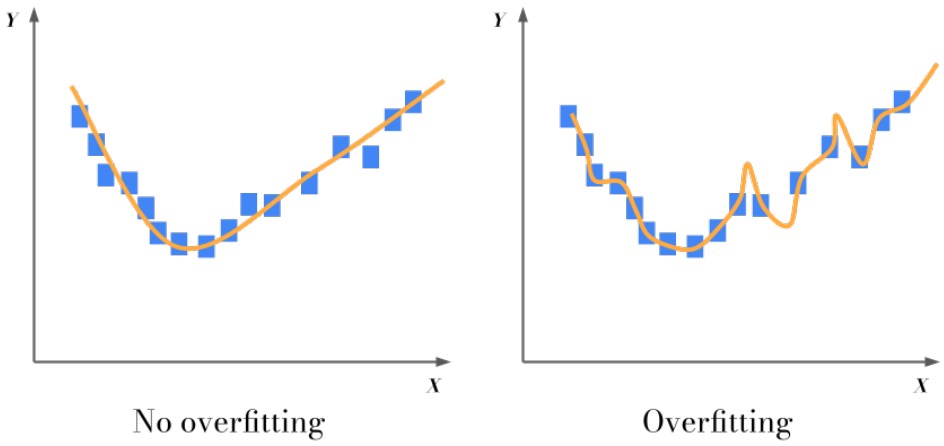


Figure IV.3.: Illustration of overfitting with a model that aims to predict Y in function of X. On the left, the model represented with the orange line capture the pattern of the data without being exact on every observation. On the right, the model is exact on every training observation but loose all its generalization capabilities.

One common approach to prevent overfitting is dropout [Hinton et al., 2012] which can help prevent the model from memorizing the training data too closely. It is a regular-

ization trick that randomly deactivate 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. 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 [Baldi and Sadowski, 2013], the key point is that the averaging of the network gives a unbiased estimate of the gradient, which is sufficient in certain conditions.
During the training process with dropout, the number of utilized weights in the network is reduced, which theoretically could result in a decrease in the network's size. 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. 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 preserves the output of the model, while introducing stochasticity to the gradient not from the dataset split into batches, but from the code itself. 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 it self. 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.7, the gradient is decomposed as a sum of all the path contributions. This decomposition can be generalized as Equation IV.1, regardless of the provenance 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 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 employs a uniform distribution across all possible paths. However, we suggest that not all gradient paths are equally important at any given time. Therefore, we aim to advance 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 notation 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 reduce 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 [Defossez et al., 2020] 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 $I_t \sim (p^t_1, \dots, p^t_N)$:*

$$g_{\mathcal{I}_t} = \begin{cases} \frac{1}{p^t_I} g_{\theta,I_t}, & \text{if } p^t_I > 0 \\ 0, & \text{otherwise} \end{cases} \tag{IV.2}$$

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 [Defossez et al., 2020]. $g_{\mathcal{I}_t}$ 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 remind 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 [Thompson, 1933], 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 memory consumption reduction, 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 4. *Lets consider the function* $f_3 : \mathbb{R} \to \mathbb{R}$ *:* $f_3(x) = x^2(2 + \cos(4x))$.

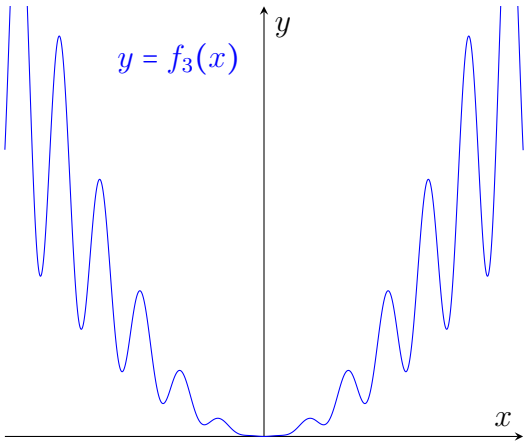


Figure IV.5.: Representation of $f_3(x) = x^2(2 + \cos(4x))$.

This function 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.*

$$\frac{\partial f_3}{\partial x} = 2x(2 + \cos(4x)) - 4x^2 \sin(4x) \tag{IV.3}$$

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 example underscores 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 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 selecting random projection matrices that sample it.

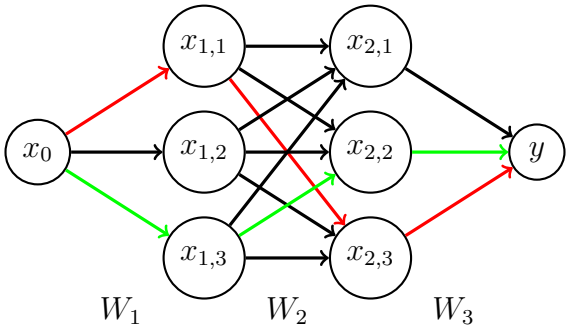


Figure IV.6.: Dense Layers with two selected paths. The graph being directed, selecting a forward path is equivalent to select a backward one.

Equation II.7 gives that the full gradient is the sum of the contributions of all the paths in the dense architecture. Let recall that parameters are the edges of the Figure IV.6 while it is the parent nodes in an execution graph. We get $y = W_3W_2W_1x_0$.
To get a path in a neural layer, one can multiply each layer by a $P_i = e_ie_i^T$ which is the outer product of standard basis vectors.

$$\frac{\partial y}{\partial \theta} = \frac{\partial W_3W_2W_1x_0}{\partial \theta} = \frac{\partial y}{\partial W_3}P3\frac{\partial W_3}{\partial W_2}P3\frac{\partial W_2}{\partial W_1}P1 \tag{IV.4}$$

$$\frac{\partial y}{\partial \theta} = \frac{\partial W_3W_2W_1x_0}{\partial \theta} = \frac{\partial y}{\partial W_3}P2\frac{\partial W_3}{\partial W_2}P2\frac{\partial W_2}{\partial W_1}P3 \tag{IV.5}$$

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 modern neural networks, a single path is insufficient to 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

The search for a good path is computationally demanding, as finding the exact best path implies to evaluate all possible paths. An approximation is then to draw and evaluate a subset of path and choose the best path of these subset. But even in this case, repeating the procedure for 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. 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.

$$\forall t \leq T \quad I_t \sim D^\epsilon_{i^{qk_{\max}}_{\max}}, \tag{IV.8}$$

with $i^t_{\max} = \arg\max_{i \in P_m} \|g_{\theta,i}\|$ and $t = qk_{\max} + r$ (euclidean division).
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, 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

Algorithm 2 SPAD

Require: $\mathcal{Z}$ (data)
Require: $\theta \in$ model (model parameters)
Require: epochs, iterations, $k_{\max}$, m, ϵ (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
$i_{\max} = \arg\max_{j \leq m} \|g_{\theta,j}(batch)\|$
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
draw I according to $(p_{\theta,1}, \ldots p_{\theta,m})$
update θ with $\frac{1}{p_{\theta,1}} g_{\theta,I}(batch)$
end for
$k \leftarrow k + 1$
end for
$epoch \leftarrow epoch + 1$
end while
Return: θ

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 quiet large values of m. The parameter $k_{\max}$ determines the number of consecutive iterations that the chosen gradient path is used. 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 [Goutam et al., 2020]. 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 were only compatible with neural networks.

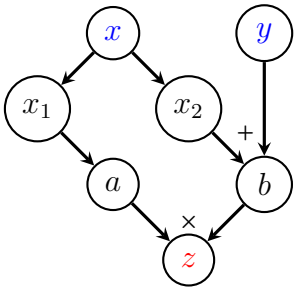
Compatibility with GCE

This chapter is presented as a bridge between Chapters I and III. It has clearly be 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 backpropagation nodes. Consequently, this regularization technique is now available for this set of white box models.

IV.4.1. 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.7 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 5. They make possible the construction of program 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.7 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 5. *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.7:*



$x \leftarrow Param_0$
$y \leftarrow Param_1$
$x_1, x_2 \leftarrow x$
$a \leftarrow e^{x_1}$
$b \leftarrow x_2 + y$
$z \leftarrow a \times b$
$Return \quad z$

Figure IV.7.: (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.

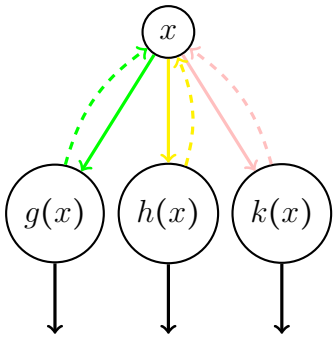
The tupling of variables in order to fulfill the SA property results in the gradient being expressed as a sum as proved in Equations IV.9. This is a key aspect of reverse mode automatic differentiation, also known as backpropagation. It is given by letting x be a parameter of f tupled in N variables $\{x_i\}_{i=1..N}$, Equation II.7 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 yields $\bar{x} = \frac{\partial f}{\partial x}$ called the adjoint of x, which is our objective. Figure IV.8 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 backpropagation. 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.8.: (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.4.2. 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.4.4, one can use SPAD and substitute the differentiation of a tupling with this new statement:

$$\langle \overline{tup \leftarrow v} \rangle^{SPAD} = \langle \overline{v_1 \dots v_t \leftarrow v} \rangle^{SPAD} = \langle \overline{v}^{SPAD} \leftarrow \oplus_{SPAD} \quad \overline{v_1} \dots \overline{v_t} \rangle$$

This introduces the challenge of determining the adjoint for such a statement. 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 optimiza-

tion 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.4.1 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.

Algorithm 3 SPAD in practice

Require: $\mathcal{Z}$ (data)
Require: $\theta \in$ model (model parameters)
Require: epochs, iterations, $k_{\max}$, m, ϵ (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 θ 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: θ

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 at ϵ or less of 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 $f(\boldsymbol{X}_T) - \arg\min_{X \in \mathcal{Z}} f(X) < \epsilon$.*

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 functions used for evaluation are described in A.3 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 [Kingma and Ba, 2014] 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 5, when the input x is used twice in the function, m is set to 2.

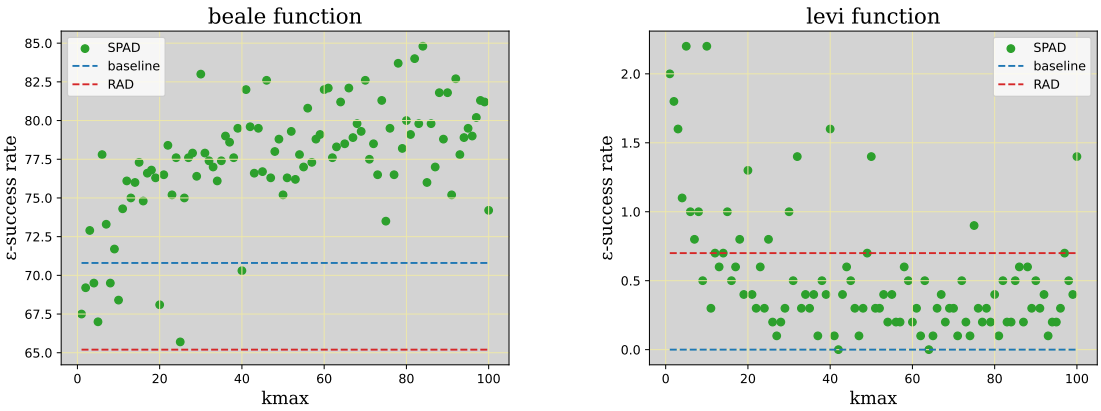
Function	baseline	RAD	$\text{SPAD}_{k_{\max}=5}$	$\text{SPAD}_{k_{\max}=50}$
Ackley	**12.2** %	0.1 %	2.1 %	1.6 %
Beale	70.8 %	65.2 %	67.0 %	**75.2** %
Levi	0.0 %	0.7 %	**2.2** %	1.4 %
Schaffer_2	**14.2** %	8.8 %	10.1 %	11.4 %

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.

Function	baseline	RAD	$\text{SPAD}_{k_{\max}=5}$	$\text{SPAD}_{k_{\max}=50}$
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.

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.



(a) $k_{\max}$ impact on the ϵ-success of beale function minimization, with $\epsilon = 0.05$.

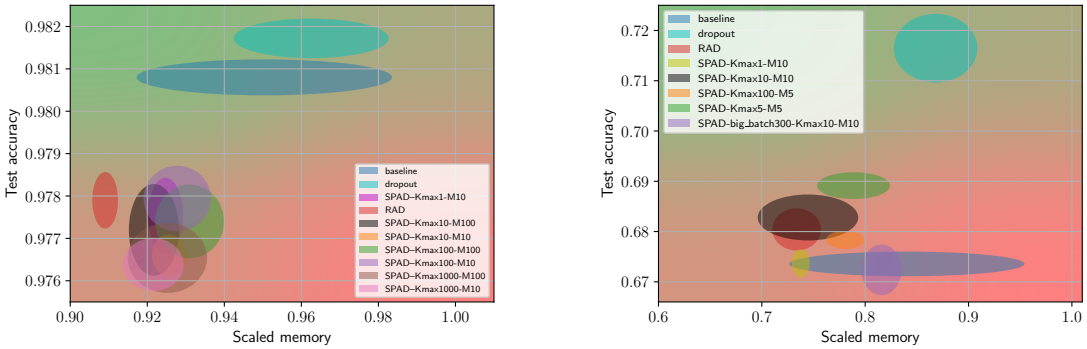
(b) $k_{\max}$ impact on the ϵ-success of levi function minimization, with $\epsilon = 0.05$.

Figure IV.9.: ϵ-success as a function of $k_{\max}$. On these graphs, the higher the better. Boths experiments show an impact of $k_{\max}$ on the ϵ-success of the gradient descent. On Figure IV.9a on the beale function, a bigger $k_{\max}$ upgrades the optimization while it is the opposite on Figure IV.9b and the levi function. In both cases, the better results are obtained with a version of SPAD that outperforms the baseline and RAD.

In Figure IV.9, we present the ϵ-success rate for varying values of $k_{\max}$ on the beale and the levi functions. On 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 increasing 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.4.1, 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 [Oktay et al., 2020], which does not include any early stopping. Doing so would increase the memory requirements that we want to avoid. However such 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, 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



(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.10.: 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. We aim to get the higher accuracy with the lowest memory usage. A run is considered as strictly better than another if it reaches higher accuracy with less memory. Otherwise one cannot rank two runs. The superior results are located in the upper left quadrant of the graph, indicated by the green color. With regards to the MNIST dataset, as shown in Figure IV.10a, 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 be utilized for an excessive number of iterations and might lose relevance at a specific stage. On IV.10b which concerns the CIFAR10 dataset, some versions of SPAD like ($k_{\max} = m = 10$) are strictly better than the baseline.

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.
Figure IV.10 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 the higher accuracy on testing with the lowest memory consumption, i.e. ending in the green zone. On these examples, the many variants of SPAD are competitive with the baseline and RAD, and it achieves strictly superior results on CIFAR10. Note that all the runs share the same neural architectures, which is a Fully connected networks on MNIST and a convolutional one on CIFAR10. More details are given in A.4.1.
Concerning overfitting, detailed results on the CIFAR10 dataset are presented in Figure IV.11, while more details on the MNIST dataset are given in the appendices. They tend to confirm that our method effectively reduces it 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 decrease 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 attain the highest test accuracy with significant memory consumption.

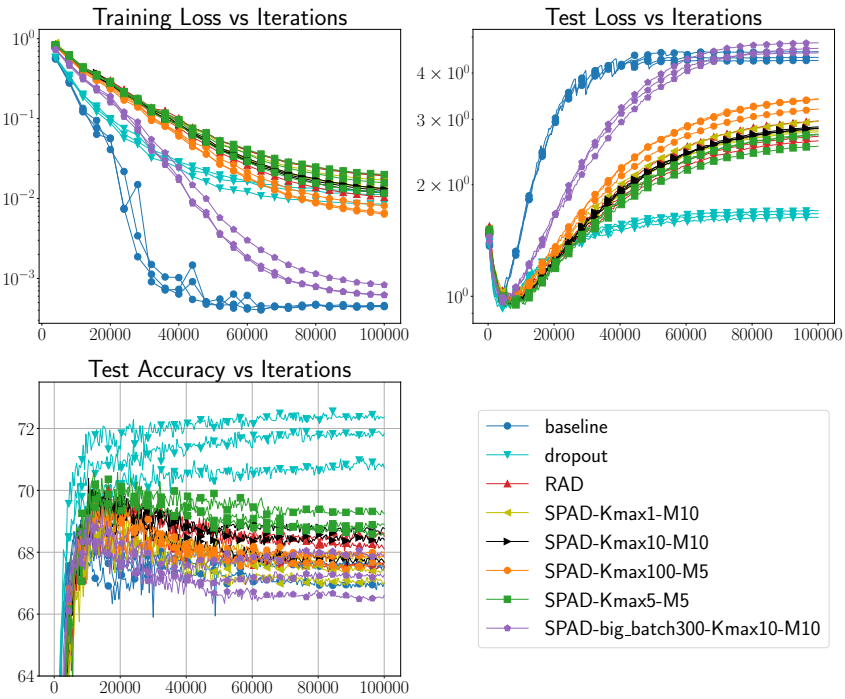


Figure IV.11.: Learning curves of the SmallConvNet on CIFAR10. The baseline model exhibits rapid performance improvement on the training dataset, while its performance on the testing dataset deteriorates just as quickly. This behavior is characteristic of overfitting, whereas the various versions of SPAD effectively mitigate this undesired decrease in generalization.

This approach effectively mitigates overfitting, as the testing loss increases at a much slower rate compared to other heuristics in Figures A.2 and A.3 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.10b. While this approach led to increased memory consumption, it also resulted in heightened overfitting, exhibiting behavior akin to the baseline in both training and testing data sets. This observation is consistent with the findings of [Keskar et al., 2017], 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

$$I_t \sim D^{\epsilon}_{s_{qk_{\max}}} \quad \text{with } s_t = \arg\min_{p \in P_m} \|\nabla f_\theta - g_{\theta,p}\| \tag{IV.10}$$

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.

IV.6. 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 [Liu et al., 2020, Csiba and Richtárik, 2016] 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.

Granularity	GD	SGD	RAD	SPAD	[Liu et al., 2020]
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.

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 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.

[1] $torch.autograd.function.FunctionCtx.save\_for\_backward$

Conclusion

TODO à rédiger en même temps que le plan de l'introduction

Summary

In the introduction we presented the context of this PhD: this PhD was funded and initiated by the french company Lokad
In chapter I ...
In chapter II ...
In chapter III ...
In chapter IV ...

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 in 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.
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 relational 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 choices made during the design 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 have 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 thanks 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.
Finally, 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

[Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[Abadi and Plotkin, 2019] Abadi, M. and Plotkin, G. (2019). A simple differentiable programming language. *Proceedings of the ACM on Programming Languages*, 4:1–28.

[Amini et al., 2018] Amini, A., Soleimany, A., Karaman, S., and Rus, D. (2018). Spatial uncertainty sampling for end-to-end control. *CoRR*, abs/1805.04829.

[Archive, 1999] Archive, U. K. (1999). Kdd99 dataset.

[Arik and Pfister, 2021] Arik, S. O. and Pfister, T. (2021). Tabnet: Attentive interpretable tabular learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(8):6679–6687.

[Badue et al., 2021] Badue, C., Guidolini, R., Carneiro, R. V., Azevedo, P., Cardoso, V. B., Forechi, A., Jesus, L., Berriel, R., Paixão, T. M., Mutz, F., de Paula Veronese, L., Oliveira-Santos, T., and De Souza, A. F. (2021). Self-driving cars: A survey. *Expert Systems with Applications*, 165:113816.

[Baldi and Sadowski, 2013] Baldi, P. and Sadowski, P. (2013). Understanding dropout. *Advances in Neural Information Processing Systems*, 26.

[Bauer, 1974] Bauer, F. L. (1974). Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11:87–96.

[Baydin et al., 2018] Baydin, A., Pearlmutter, B., Radul, A., and Siskind, J. (2018). Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research*, 18:1–43.

[Baydin et al., 2022] Baydin, A., Pearlmutter, B., Syme, D., Wood, F., and Torr, P. (2022). Gradients without backpropagation. *arXiv preprint*, arXiv.

[Baydin et al., 2020] Baydin, A. G., Cranmer, K., Feickert, M., Gray, L., Heinrich, L., Held, A., Melo, A., Neubauer, M., Pearkes, J., Simpson, N., Smith, N., Stark, G., Thais, S., Vassilev, V., and Watts, G. (2020). Differentiable programming in high-energy physics. In *Snowmass 2021 Letters of Interest (LOI), Division of Particles and Fields (DPF), American Physical Society*.

[Belouze, 2022] Belouze, G. (2022). Optimization without backpropagation. *arXiv preprint*, arXiv.

[Blackard and Dean, 1999] Blackard, J. A. and Dean, D. J. (1999). Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and Electronics in Agriculture*, 24:131–151.

[Blanco-Justicia et al., 2020] Blanco-Justicia, A., Domingo-Ferrer, J., Martínez, S., and Sánchez, D. (2020). Machine learning explainability via microaggregation and shallow decision trees. *Knowledge-Based Systems*, 194:105532.

[Borisov et al., 2022] Borisov, V., Broelemann, K., Kasneci, E., and Kasneci, G. (2022). Deeptlf: robust deep neural networks for heterogeneous tabular data. *International Journal of Data Science and Analytics*.

[Borisov et al., 2021] Borisov, V., Leemann, T., Seßler, K., Haug, J., Pawelczyk, M., and Kasneci, G. (2021). Deep neural networks and tabular data: A survey.

[Bradbury et al., 2018] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs.

[Cerezo and Coles, 2021] Cerezo, M. and Coles, P. J. (2021). Higher order derivatives of quantum neural networks with barren plateaus. *Quantum Science and Technology*, 6(3):035006.

[Chen, 2009] Chen, L. (2009). *Curse of Dimensionality*, pages 545–546. Springer US, Boston, MA.

[Chen and Guestrin, 2016] Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[Cheng et al., 2016] Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., Anil, R., Haque, Z., Hong, L., Jain, V., Liu, X., and Shah, H. (2016). Wide and deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, DLRS 2016, page 7–10, New York, NY, USA. Association for Computing Machinery.

[Codd, 1970] Codd, E. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387.

[Csiba and Richtárik, 2016] Csiba, D. and Richtárik, P. (2016). Importance sampling for minibatches. *Journal of Machine Learning Research*, 19.

[Cunneen et al., 2019] Cunneen, M., Mullins, M., and Murphy, F. (2019). Autonomous vehicles and embedded artificial intelligence: The challenges of framing machine driving decisions. *Applied Artificial Intelligence*, 33(8):706–731.

[Defossez et al., 2020] Defossez, A., Bottou, L., Bach, F., and Usunier, N. (2020). On the convergence of adam and adagrad. *arXiv preprint*, arXiv.

[Deletoille, 2022] Deletoille, G. (2022). *Gestion de stock sous contrainte de quantité minimale de commande multi-références*. PhD thesis. Thèse de doctorat dirigée par Adam, Sébastien Informatique Normandie 2022.

[Deng et al., 2020] Deng, C., Ji, X., Rainey, C., Zhang, J., and Lu, W. (2020). Integrating machine learning with human knowledge. *iScience*, 23(11):101656.

[Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159.

[Durut and Rossi, 2011] Durut, M. and Rossi, F. (2011). Communication challenges in cloud k-means. In *Proceedings of XIXth European Symposium on Artificial Neural Networks (ESANN 2011)*, pages 387–392, Bruges (Belgium).

[Fayaz et al., 2022] Fayaz, S. A., Zaman, M., Kaul, S., and Butt, M. A. (2022). Is deep learning on tabular data enough? an assessment. *International Journal of Advanced Computer Science and Applications*, 13(4).

[Frosst and Hinton, 2017] Frosst, N. and Hinton, G. E. (2017). Distilling a neural network into a soft decision tree. *ArXiv*, abs/1711.09784.

[Gardner, 1985] Gardner, E. S. (1985). Exponential smoothing: The state of the art. *Journal of Forecasting*, 4:1–28.

[Gorishniy et al., 2021] Gorishniy, Y., Rubachev, I., Khrulkov, V., and Babenko, A. (2021). Revisiting deep learning models for tabular data. In *NeurIPS*.

[Goutam et al., 2020] Goutam, K., S, B., Gera, D., and Sarma, R. (2020). Layerout: Freezing layers in deep neural networks. *SN Computer Science*, 1:295.

[Griewank, 1994] Griewank, A. (1994). Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1.

[Gupta and Agrawal, 2022] Gupta, M. and Agrawal, P. (2022). Compression of deep learning models for text: A survey. *ACM Trans. Knowl. Discov. Data*, 16(4).

[Hascoët and Pascual, 2013] Hascoët, L. and Pascual, V. (2013). The tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39:20:1–20:43.

[Hayashi, 2020] Hayashi, Y. (2020). Does deep learning work well for categorical datasets with mainly nominal attributes? *Electronics*, 9:1966.

[Hinton et al., 2012] Hinton, G., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint*, arXiv.

[Hu et al., 2020] Hu, Y., Anderson, L., Li, T.-M., Sun, Q., Carr, N., Ragan-Kelley, J., and Durand, F. (2020). Difftaichi: Differentiable programming for physical simulation. *ICLR*.

[Innes, 2018] Innes, M. (2018). Don't unroll adjoint : Differentiating ssa-form programs. *ArXiv*, abs/1810.07951.

[Kadra et al., 2021] Kadra, A., Lindauer, M. T., Hutter, F., and Grabocka, J. (2021). Well-tuned simple nets excel on tabular datasets. In *NeurIPS*.

[Kaggle, 2022] Kaggle (2022). Don't get kicked competitions. https://www.kaggle.com/c/DontGetKicked/overview.

[Keskar et al., 2017] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2017). On large-batch training for deep learning: Generalization gap and sharp minima. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

[Kingma and Ba, 2014] Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *International Conference on Learning Representations*.

[Kohavi, 1997] Kohavi, R. (1997). Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid. *KDD*.

[Lahmeri et al., 2021] Lahmeri, M.-A., Kishk, M. A., and Alouini, M.-S. (2021). Artificial intelligence for uav-enabled wireless networks: A survey. *IEEE Open Journal of the Communications Society*, 2:1015–1040.

[Land and Doig, 2010] Land, A. H. and Doig, A. G. (2010). *An Automatic Method for Solving Discrete Programming Problems*, pages 105–132. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Laue, 2019] Laue, S. (2019). On the equivalence of forward mode automatic differentiation and symbolic differentiation. *CoRR*, abs/1904.02990.

[Lee et al., 2020] Lee, W., Yu, H., Rival, X., and Yang, H. (2020). On correctness of automatic differentiation for non-differentiable functions. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 6719–6730. Curran Associates, Inc.

[Lepchenkov, 2019] Lepchenkov, K. (2019). Used-cars-catalog.

[Li et al., 2018] Li, T.-M., Gharbi, M., Adams, A., Durand, F., and Ragan-Kelley, J. (2018). Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics (TOG)*, 37:1 – 13.

[Liu et al., 2020] Liu, R., Wu, T., and Mozafari, B. (2020). 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*.

[Luo et al., 2021] Luo, H., Cheng, F., Yu, H., and Yi, Y. (2021). Sdtr: Soft decision tree regressor for tabular data. *IEEE Access*, 9:55999–56011.

[Mak and Ong, 2021] Mak, C. and Ong, C. (2021). A differential-form pullback programming language for higher-order reverse-mode automatic differentiation.

[Mari et al., 2021] Mari, A., Bromley, T. R., and Killoran, N. (2021). Estimating the gradient and higher-order derivatives on quantum hardware. *Phys. Rev. A*, 103:012405.

[Mathov et al., 2022] Mathov, Y., Levy, E., Katzir, Z., Shabtai, A., and Elovici, Y. (2022). Not all datasets are born equal: On heterogeneous tabular data and adversarial examples. *Knowledge-Based Systems*, 242:108377.

[Message Passing Interface Forum, 2021] Message Passing Interface Forum (2021). *MPI: A Message-Passing Interface Standard Version 4.0*.

[of Chicago, 2022] of Chicago, C. (2022). Taxi trips of chicago.

[Oktay et al., 2020] Oktay, D., McGreivy, N., Aduol, J., Beatson, A., and Adams, R. P. (2020). Randomized automatic differentiation.

[Ostroumova et al., 2018] Ostroumova, L., Gusev, G., Vorobev, A., Dorogush, A. V., and Gulin, A. (2018). Catboost: unbiased boosting with categorical features. In *NeurIPS*.

[Parsana and Atkotiya, 2019] Parsana, F. and Atkotiya, K. (2019). Study and analyzing an effective query optimization, strategies and algorithms in database systems. In *Conference: Emerging trends in computer science and information technology*.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035.

[Peseux, 2021] Peseux, P. (2021). Differentiating relational queries. In *PhD@VLDB*.

[Peseux et al., 2022] Peseux, P., Berar, M., Paquet, T., and Nicollet, V. (2022). Stochastic gradient descent with gradient estimator for categorical features. *arXiv preprint*, arXiv.

[Popov et al., 2020] Popov, S., Morozov, S., and Babenko, A. (2020). Neural oblivious decision ensembles for deep learning on tabular data.

[Rehman, 2021] Rehman, U. (2021). Relation on nlp with machine and language. *Global Sci-Tech*, 13:39–42.

[Ribeiro et al., 2016] Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "why should i trust you?": Explaining the predictions of any classifier. In *HLT-NAACL Demos*, pages 97–101. The Association for Computational Linguistics.

[Robbins and Monro, 1951] Robbins, H. and Monro, S. (1951). A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400 – 407.

[Roussel et al., 2022] Roussel, R., Edelen, A., Ratner, D., Dubey, K., Gonzalez-Aguilera, J. P., Kim, Y. K., and Kuklev, N. (2022). Differentiable preisach modeling for characterization and optimization of particle accelerator systems with hysteresis. *Phys. Rev. Lett.*, 128:204801.

[Rudin, 2019] Rudin, C. (2019). Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1:206–215.

[Schleich et al., 2019] Schleich, M., Olteanu, D., Abo-Khamis, M., Ngo, H. Q., and Nguyen, X. (2019). Learning models over relational data: A brief tutorial. In Ben Amor, N., Quost, B., and Theobald, M., editors, *Scalable Uncertainty Management*, pages 423–432. Springer International Publishing.

[Schüle et al., 2019] Schüle, M. E., Simonis, F., Heyenbrock, T., Kemper, A., Günnemann, S., and Neumann, T. (2019). In-database machine learning: Gradient descent and tensor algebra for main memory database systems. In *BTW*.

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.

[Shwartz-Ziv and Armon, 2021] Shwartz-Ziv, R. and Armon, A. (2021). Tabular data: Deep learning is not all you need. *Information Fusion*, 81.

[Song et al., 2019] Song, W., Shi, C., Xiao, Z., Duan, Z., Xu, Y., Zhang, M., and Tang, J. (2019). Autoint: Automatic feature interaction learning via self-attentive neural networks. *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*.

[Sutton et al., 2000] Sutton, R., Mcallester, D., Singh, S., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. *Adv. Neural Inf. Process. Syst*, 12.

[Taylor and Letham, 2017] Taylor, S. and Letham, B. (2017). Forecasting at scale. *The American Statistician*, 72.

[Teikari et al., 2018] Teikari, P., Najjar, R. P., Schmetterer, L., and Milea, D. (2018). Embedded deep learning in ophthalmology: making ophthalmic imaging smarter. *Therapeutic Advances in Ophthalmology*, 11.

[Thompson, 1933] Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25:285–294.

[Trummer and Koch, 2016] Trummer, I. and Koch, C. (2016). Multi-objective parametric query optimization. *ACM SIGMOD Record*, 45:24–31.

[van Merrienboer et al., 2018] van Merrienboer, B., Breuleux, O., Bergeron, A., and Lamblin, P. (2018). Automatic differentiation in ml: Where we are and where we should be going. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

[Wei et al., 2020] Wei, R., Zheng, D., Rasi, M., and Chrzaszcz, B. (2020). Differentiable programming manifesto. https://github.com/apple/swift/blob/main/docs/DifferentiableProgramming.md.

[Wengert, 1964] Wengert, R. (1964). A simple automatic derivative evaluation program. *Commun. ACM*, 7:463–464.

[Wes McKinney, 2010] Wes McKinney (2010). Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, pages 56 – 61.

[Williams, 2004] Williams, R. J. (2004). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256.

Mots clés : Données relationnelles, Programmation Différentiable, Différentiation automatique ...

Keywords: Relational data, Differentiable Programming, Automatic Differentiation ...

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 finance, la santé et la chaîne d'approvisionnement, 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 de cette thèse est disponible ici[2].

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 finance, 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.

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

List of Tables

A. Appendices

A.1. SVD thanks to spectral theorem

The spectral theorem states that any symmetric matrix can be diagonalized via an orthogonal matrix. This means that there exists a decomposition of a symmetric matrix M of the form $M = Q\Lambda Q^T$, where Λ is a diagonal matrix of the eigenvalues of M, and Q is an orthogonal matrix composed of the eigenvectors of M.
The SVD of a matrix M can be obtained by applying the spectral theorem to MM^T and M^TM. This leads to the following decomposition:

$$M = U\Sigma V^T \tag{A.1}$$

where U and V are orthogonal matrices, and Σ is a diagonal matrix containing the singular values of M.
Thus, the SVD provides a way to factorize a matrix into three components, which can be used for various applications such as matrix inversion, low-rank approximation, and data compression.

A.2. Deep learning results with GCE

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. 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

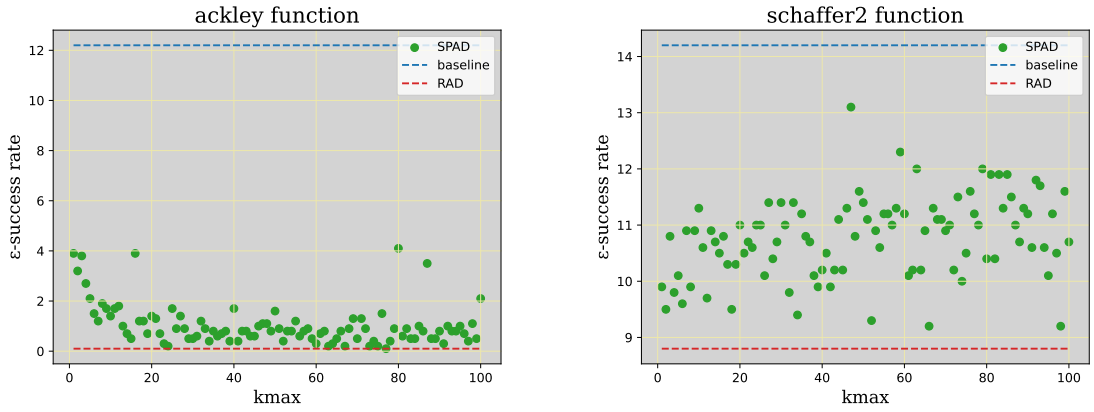
$$beale(x,y) = \left(1.5 - x + xy\right)^2 + \left(2.25 - x + xy^2\right)^2 + \left(2.625 - x + xy^3\right)^2$$

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

$$schaffer_2(x,y) = 0.5 + \frac{\sin^2\left(x^2-y^2\right) - 0.5}{\left[1+0.001\left(x^2+y^2\right)\right]^2}$$



(a) k_{max} on ackley function

(b) k_{max} on schaffer2 function

Figure A.1.: ϵ-success in function of k_{max}. see Figure IV.10 for more context.

A.4. Deep learning results with SPAD

A.4.1. Architecture

The architectures used are directly taken from [Oktay et al., 2020]. 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: $3 \times 32 \times 32$-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)

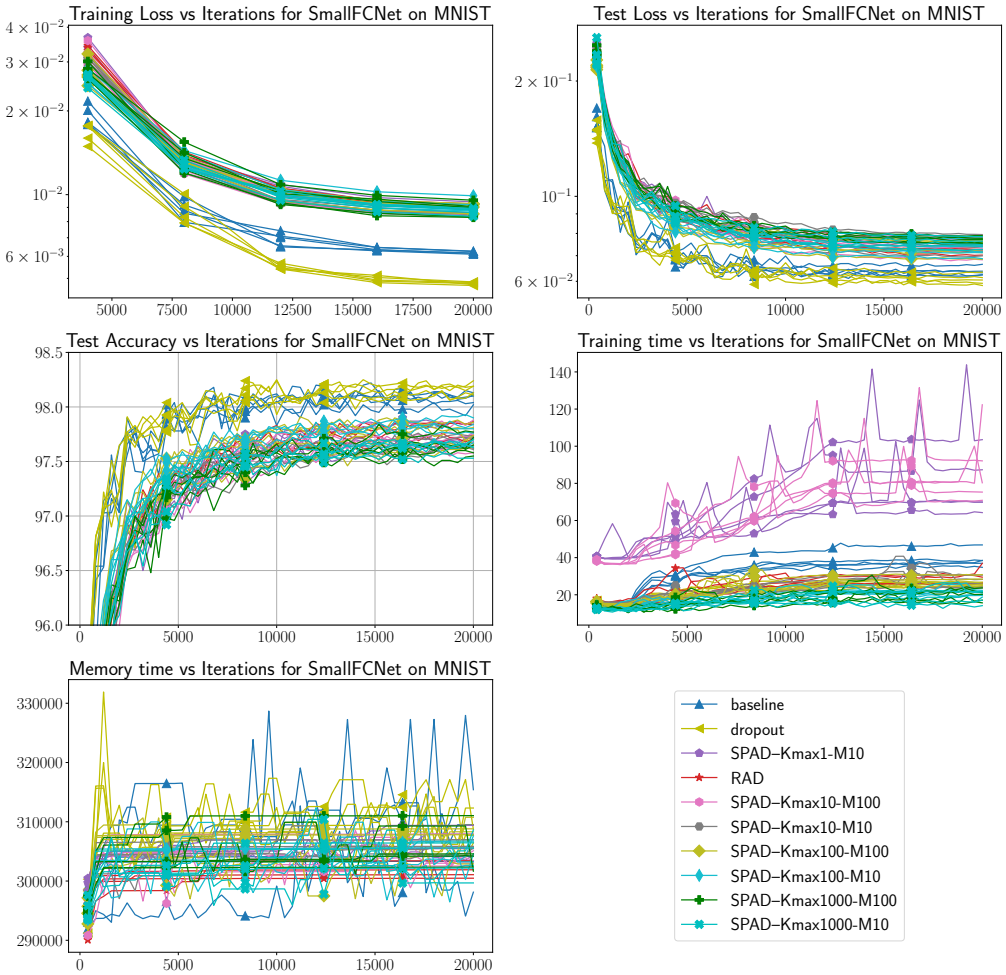


Figure A.2.: Full train/test curves and memory consumption per iteration on MNIST.

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

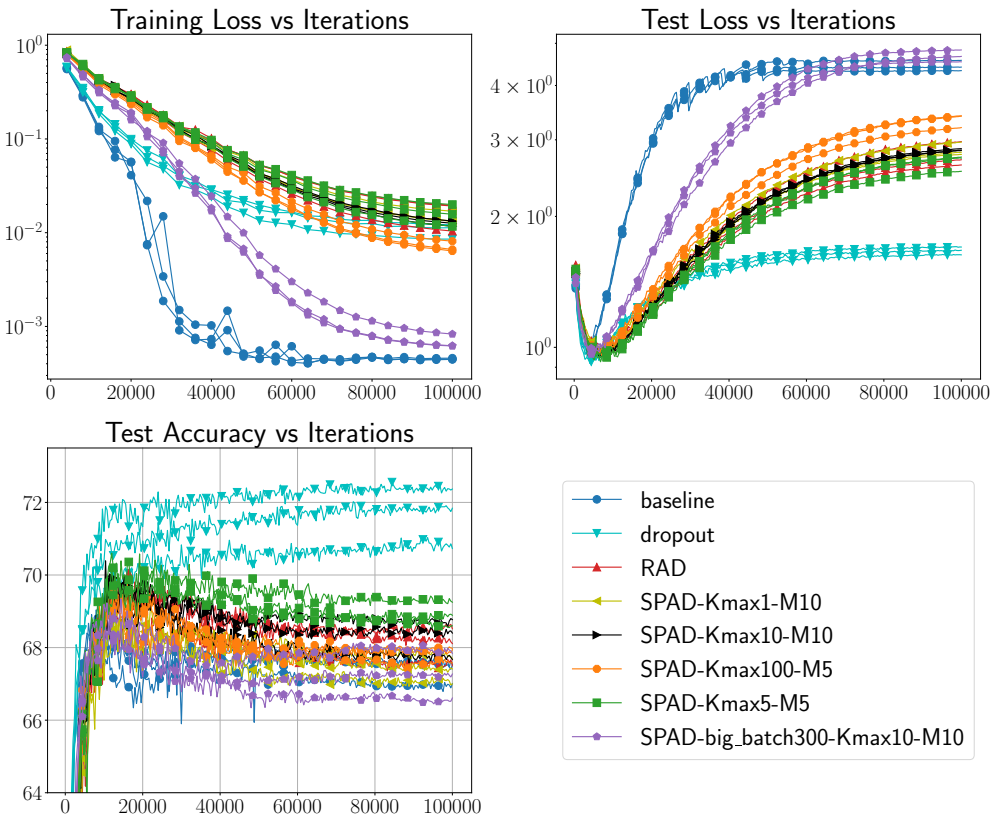


Figure A.3.: Full train/test curves and memory consumption per iteration on CIFAR10.

Figure A.2, 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.3), 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.