UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JOÃO MARCOS FLACH

Lambda Calculus meets Machine Learning

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Luís Lamb

Porto Alegre
August 2023

CIP — CATALOGING-IN-PUBLICATION

Flach, João Marcos

Lambda Calculus meets Machine Learning / João Marcos Flach. – Porto Alegre: PPGC da UFRGS, 2023.

94 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2023. Advisor: Luís Lamb.

1. Machine learning. 2. Lambda calculus. 3. Neural network. 4. Sequence-to-sequence model. 5. Transformer model. 6. Symbolic AI. 7. Neuro-symbolic computation. I. Lamb, Luís. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof[a]. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof. Sérgio Luis Cechin
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"DO IT FOR HER"

— Homer Simpson

AGRADECIMENTOS

Gostaria de primeiramente agradecer a Deus, por me dar forças para terminar meu trabalho quando achei que já não era possível.

Gostaria de agradecer também à minha família por ter me apoiado nesse momento tão difícil. Agradeço especialmente à minha companheira Izabela, com a qual pude contar em todos os momentos. Nossas longas conversas caminhando pela praça sempre conseguiam me ajudar a desembaraçar meus pensamentos e foram fundamentais para que eu conseguisse entregar esta dissertação; à minha querida filha Matilda, que sempre carregava em seu rosto um sorriso para ensolarar meus dias mais nublados; e à Furiosa e Cidinha, que nunca deixaram de demonstrar todo o carinho que sentem por mim.

Além disso, agradeço aos meus amigos que sempre estiveram do meu lado nessa jornada, especialmente meu consultor oficial de assuntos acadêmicos, Renan, e meu camarada Lucas. Também gostaria de agradecer aos parceiros do meu time de hockey Blades que, mesmo durante a pandemia, mantiveram contato e ajudaram a enfrentar esse período tão difícil na vida de todos nós.

Por fim, gostaria de agradecer a todos que, direta ou indiretamente, me ajudaram nessa jornada.

ABSTRACT

Neural Networks are usually used in the context of machine learning to solve statistical problems and calculate approximations. However, its use for symbolic learning has been increasing in the past years, placing them in the Neurosymbolic realm. To study the capabilities of Neural Networks in this kind of task, several works have already explored the ability of these networks to learn mathematical operations, such as addition and multiplication, logic inference, such as theorem provers, and even execution of computer programs. The latter is known to be a task too complex for neural networks. Therefore, the results were not always successful and often required the introduction of biased elements in the learning process, in addition to restricting the scope of possible programs to be executed. In this work, we will be interested in studying the ability of Neural Networks to learn how to execute programs as a whole. To achieve this, we propose a different approach. Instead of using an imperative programming language with complex structures, we use the Lambda Calculus (λ-Calculus), a simple yet Turing-Complete mathematical formalism, which serves as the basis for modern functional languages. As the execution of a program in λ-Calculus is based on reductions, we will show that learning how to perform these reductions is enough to execute any program.

Keywords: Machine learning. lambda calculus. neural network. sequence-to-sequence model. transformer model. symbolic AI. neuro-symbolic computation.

Cálculo Lambda encontra Aprendizado de Máquina

RESUMO

Redes neurais são geralmente utilizadas para resolver problemas estatísticos e calcular aproximações. No entanto, seu uso para aprendizagem simbólica vem aumentando nos últimos anos, colocando-as no domínio Neuro-Simbólico. Para estudar as capacidades das Redes Neurais nesse tipo de tarefa, vários trabalhos já exploraram a capacidade dessas redes em aprender operações matemáticas, como adição e multiplicação, inferências lógicas, como provadores de teoremas, e até execução de programas de computador. Esta última tarefa é conhecida por ser muito complexa para redes neurais. Portanto, nem sempre os resultados são bem-sucedidos, e muitas vezes exigem a introdução de elementos enviesados no aprendizado, além de restrição no escopo de programas a serem executados. Neste trabalho, estaremos interessados em estudar a capacidade das Redes Neurais em aprender a executar programas como um todo. Para isso, propomos uma abordagem diferente. Ao invés de usar uma linguagem de programação imperativa, com estruturas complexas, usamos o Lambda Calculus (λ-Calculus), um formalismo matemático simples, mas Turing-Completo, que serve de base para as linguagens funcionais modernas. Como a execução de um programa em λ-Calculus é baseada em reduções, mostraremos que basta aprender a fazer essas reduções para que possamos executar qualquer programa.

Palavras-chave: Aprendizado de máquina. cálculo lambda. redes neurais. modelos sequência para sequência. modelo transformer. IA simbólica. computação neuro-simbólica.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AI	*Artificial Intelligence*
NN	*Neural Network*
LR	*Learning Rate*
GRU	*Gated Recurrent Unit*
LSTM	*Long Short-Term Memory*
RNN	*Recurrent Neural Network*
LC	*Lambda Calculus*
LT	*Lambda Term*
DB	*de Bruijn*
OBR	*One-Step Beta Reduction*
MBR	*Multi-Step Beta Reduction*
LS	*Lambda Set*
RLS	*Random Lambda Set*
CBLS	*Closed Bool Lambda Set*
OBLS	*Open Bool Lambda Set*

LIST OF SYMBOLS

λ	Lambda Abstraction on infix notation
α	Alpha
β	Beta
$f, x, y, z, ...$	Lambda Variables
V	Set of Lambda Variables
$M, N, P, M', N', ...$	Lambda Terms
Λ	Set of Lambda Terms
Λ_{DB}	Set of Lambda Terms on de Bruijn Notation
@	Lambda Application
L	Lambda Abstraction on prefix notation
FV	Free Variables
$=_\alpha$	Alpha Equivalence
$\rightarrow_\beta$	Beta Reduction
$\twoheadrightarrow_\beta$	Multi-Step Beta Reduction
$=_\beta$	Beta Equivalence

CONTENTS

1 INTRODUCTION

In the field of machine learning, one key objective is to understand the best way to approach the task of learning from data. One approach, which has been referred to as rule-based inference, emphasizes the use of explicit logical rules to reason about the data and make predictions or decisions. The other perspective, known as statistical learning, involves using mathematical models to automatically extract patterns and relationships from the data (RUMELHART; MCCLELLAND, 1986).

Traditionally, neural networks have been employed in the statistical learning, solving problems such as speech recognition, machine translation, handwriting recognition, etc., rather than the symbolic learning. However, recent advancements in the field have resulted in the introduction of models that are changing the landscape and allowing us to tackle a wider range of problems, including symbolic ones, using neural networks. When neural networks are applied to symbolic problems, the result is a hybrid approach that combines the advantages of both. This combination of the two approaches falls under the realm of Neurosymbolic AI (KAUTZ, 2022). This field combines the advantages that each of the paradigms presents (GARCEZ; LAMB; GABBAY, 2009), and there has been increasing focus on the area in the last years (GARCEZ; LAMB, 2020).

In this work, we intend to explore the capacity of Machine Learning models, specifically, the Transformer (VASWANI et al., 2017), to learn to perform computations, a symbolic field that has been traditionally seen as being too complex for neural networks to handle. For this, we use a simple but powerful formalism, the Lambda Calculus (λ-Calculus), as the underlying framework (BARENDREGT, 1984).

1.1 Goals

The idea of training a machine learning model to perform computations is relatively new and involves teaching the model to understand the underlying logic and rules involved in mathematical operations. The majority of the works in this field tend to restrict the domain of the programs the model can take as input. Therefore, our research question is: **“Can a Machine Learning model learn to perform computations?”**.

Considering that computer programs do not have a fixed size, for the machine learning part, we use a sequence-to-sequence (seq2seq) model, which can take inputs and produce outputs of any length. Specifically, we use a model that has been widely

used for several kinds of applications and also tested for symbolic tasks, the Transformer (VASWANI et al., 2017).

For the computations, we use the λ-Calculus, a formalism that, although simple and compact, can perform any computation, according to the Church-Turing thesis (SIPSER, 1996). In essence, the λ-Calculus can be seen as a programming language consisting of terms (λ-terms) that can be subject to reductions. The λ-Calculus actually is at the core of many modern programming languages, especially the functional ones (MICHAELSON, 2011). The λ-terms can be viewed as programs, and the reductions can be interpreted as computations performed within the formalism. Applying a single reduction to a term represents an one-step computation in the λ-Calculus. On the other hand, a full computation involves applying reductions successively on a term that has a normal form until it reaches it, i.e., no more reductions are possible. With these ideas in mind, we propose two hypotheses aimed at answering our research question:

- **H1: The Transformer model can learn to perform a one-step computation on Lambda Calculus.**
- **H2: The Transformer model can learn to perform a full computation on Lambda Calculus.**

It is clear that the hypothesis **H1** is easier to validate than **H2** since a one-step computation is simpler to perform than a full computation. Thus, the purpose of these hypotheses is to gradually enhance our comprehension of the subject matter and enable us to provide an answer to our research question.

1.2 Related Work

In the work of Zaremba and Sutskever (2014), seq2seq models are used to learn to evaluate short computer programs using an imperative language with the *Python* syntax. But their domain of programs is restricted as their programs are short and can use just some arithmetic operations, variable assignment, if-statements, and for loops (not nested). Every program *prints* an integer as output. Our goal is rather not to limit the domain of programs that our model can learn, focusing solely on the syntactic operations performed to achieve the result.

There are some other studies that also claim to have developed models that learn algorithms or learn to execute computer code, including Kaiser and Sutskever (2015),

Graves, Wayne and Danihelka (2014), and Trask et al. (2018). However, the domain of these works is restricted to some arithmetical operations or sequence computations (copying, duplicating, sorting, etc.). Additional works concentrate on acquiring an understanding of program representation. For example, Maddison and Tarlow (2014) builds generative models of natural source code, while Mou et al. (2014) applies neural networks to determine if two programs are equivalent.

In Vaswani et al. (2017), the Transformer model was first introduced, bringing several key advancements and improvements compared to the state-of-the-art seq2seq models prevalent at that time. This new model boasted improved parallelism, reduced sequential processing requirements, and the ability to handle longer sequences, among other things.

These innovative features have contributed to the widespread adoption of the Transformer model in various Machine Learning applications, including some that involve symbolic mathematics. In a study by Lample and Charton (2019), the Transformer model was applied to learn how to symbolically integrate functions, yielding promising results. The authors demonstrated that the model was capable of learning how to compute integrals in a way that was both accurate and efficient, outperforming existing methods in many cases. This study highlights the versatility and potential of the Transformer model, making it a valuable tool for tackling a wide range of machine learning tasks, especially in areas that require symbolic reasoning and mathematical operations.

Also, recent developments in chatbot technology have been enabled by the Transformer model. One example of a chatbot that has emerged as a result of this development is the *ChatGPT* [a], which is based on a state-of-the-art AI model, the GPT-3, from Brown et al. (2020). The chatbot can answer questions about a variety of subjects (ROOSE, 2022), and it can also perform some basic symbolic reasoning. However, the symbolic reasoning is still limited, as it gives some incorrect answers to very simple questions.

In the present work, we shift the paradigm from the imperative paradigm that all other works have used to the functional paradigm, which is the case for the λ-Calculus. With this, we try to abstract the idea of learning to compute computer programs to learning to perform reductions on λ-terms. With this approach, we were able to obtain an accuracy of 88.89% for learning the one-step β-reduction on completely random terms and 99.73% on terms that represent boolean expressions. Also, for the full computation

[a] available at <https://openai.com/blog/chatgpt/>

task, we were able to obtain an accuracy of 97.70% for terms that represent boolean expressions. If we consider the string similarity metric, where we compare how many characters of the λ-term the model predicted right, the majority of our results had a similarity above 99%. With these results, we think that this change in the paradigm and the use of the Transformer model are two improvements that can be led into account in future research.

1.3 Dissertation Structure

The remainder of the current study is organized as follows:

- Chapters 2 and 3 provide most of the essential background information required to understand this dissertation. In particular, Chapter 2 presents a concise introduction to the λ-Calculus. The chapter covers the fundamental aspects of the formalism, including its syntax, semantics, and core concepts. On the other hand, Chapter 3 focuses on the Artificial Intelligence and Machine Learning field, providing an overview of Neural Network models, with a specific emphasis on Sequence-to-Sequence (seq2seq) models, especially the Transformer, which we are going to be utilizing in this study.
- Chapter 4 provides a detailed overview of the methodology that is going to be employed in the experiments conducted in this dissertation. It explains the various steps involved in the experimental process, such as data generation and preparation, model selection, and evaluation metrics. The chapter also discusses the experimental design, including the research questions being investigated and the hypotheses being tested, as well as the specific techniques and tools used to conduct the experiments.
- Chapter 5 focuses on the process of generating the datasets that are going to be utilized in the experiments conducted in this dissertation. The chapter describes the methods used to generate these datasets, including selecting relevant parameters and the specific algorithms and techniques employed.
- Chapters 6 and 7 presents the outcomes of the experiments conducted in this dissertation, along with an analysis and discussion of the findings. They summarize the analysis and interpretation of the data obtained from the experiments and the extent to which they support the hypotheses and research questions.

- Finally, Chapter 8 provides a summary of the key findings and their significance, as well as the potential paths for future research.

2 LAMBDA CALCULUS

The Lambda Calculus (λ-calculus or LC) is known to be a simple and elegant foundation for computation. It is a formal system based on functions and is based on function abstraction, which captures the notion of function definition, and function application, which captures the notion of the application of a function to its parameters. It is the base for modern functional programming languages, like Racket, Haskell, and others (MICHAELSON, 2011). It was introduced by Alonzo Church in the 1930s, and has become one of the main computational models (HINDLEY; CARDONE, 2006).

In this chapter, we first present a brief history of the λ-calculus, talk about its importance and give an overall look at its main aspects. Then, we introduce its traditional syntax and a tree notation used to propose a prefix notation for the λ-terms. Later, we show some definitions, based on the notation of Machado (2013), which was based on the definitions by Barendregt (1984). Next, we present some encodings for integer and boolean arithmetic, as well as some examples of computations using the λ-calculus. Finally, we show another alternative notation for the formalism.

2.1 History and Importance

The effervescence of ideas and theories for the foundations of mathematics in the early XX century, and the aim of solving one of the greatest mathematical problems of the time - The *Entscheidungsproblem*, German word for the *Decision Problem* - ended up with two different ways of giving the problem a solution, almost at the same time. Each of those ways turned out to be the answer to a subject that was in the mind of philosophers, logicians, and mathematicians for centuries: “What can a machine do?”. Also, it was the beginning of one of the greatest discoveries of all times: the computer (DAVIS, 2001).

To give the *Entscheidungsproblem* a solution, each author had to propose an idea of what effective computability is. In Cambridge, Alan Turing proposed the Turing Machine (TURING, 1936). Across the ocean, in Princeton, Alonzo Church proposed the Lambda Calculus (CHURCH, 1936). Although Church had proposed a version of the LC years before (CHURCH, 1932), it was in 1936 that he proposed the version that we know today (HINDLEY; CARDONE, 2006). Later, Turing proved that the two formalisms were actually equivalent (TURING, 1937).

Later, the Church-Turing Thesis was proposed, saying that the computational capacity of the Turing Machine - and the Lambda Calculus - is the maximum limit of any computational model (SIPSER, 1996). As the notion of algorithms and computable functions are intuitive, this thesis is not demonstrable. However, since it is accepted as true for computer science, it is also known as the Church-Turing Hypothesis (DIVERIO; MENEZES, 2009).

Although the Turing Machine is the most famous and influential computational model, the Lambda Calculus has its importance too. Not only by its historical importance but also because it is a very small and expressive formalism and is the base for several languages that follow the functional programming paradigm, such as Lisp, OCaml, Haskell, among others (MACHADO, 2013). It also serves as a core formalism for other formalisms, such as type theory (CHURCH, 1940).

2.2 Concept

Lambda Calculus is a formal system that captures the core notion of functions - its definition and application (MACHADO, 2013). Here, it is important to distinguish between the two main notions of functions: functions as graphs and functions as rules. Functions as graphs is the notion taught in school, where a function is defined by sets of pairs, like (x, f(x)). The other notion, function as rules, is considered the process of going from the argument to the value, a process that is encoded by the function definition. It is easy to see that the second notion is much closer to computer science than the first one since you are looking at the function as rules, i.e., as an algorithm. The notion that the LC regards is the latter. (BARENDREGT, 1984)

As was said, LC takes only the necessary part of functions to build its theory. Consider this function, written as we normally write function:

$$f(x) = x + 1$$

This is the **definition** of a function f [a]. After defined, we can apply it to an element of its domain (in this case, a number) to produce a value:

[a] One can argue that this is not effective the definition of f, but an equation in which f is in (DIVERIO; MENEZES, 2009): $f(x) - x + 1 = 0$

$$f(1) = 1 + 1$$
$$f(1) = 2$$

This is the **application** of the function f. When we apply a function to an argument, we produce a value. In fact, f is just the name of a function that, given a number, returns its successor. If we write $g(x) = x + 1$ or $h(x) = x + 1$, they are, in fact, the same function, with different names.

So, as shown, a function is not its name. You have to separate the name of the function from the function itself. But with ordinary mathematics (the one we learn in school), we can not write the function separate from its name. We need a new notation, and that is the main goal of the Lambda Language. With this language, we can write the function that f, g and h represents as:

$$\lambda\, x.\, x + 1$$

This term is saying that it is a function (because of the λ), it takes one argument that we are calling x, and returns its successor[b]. The general form of Lambda Terms (λ-terms or LTs) would be:

$$\lambda\ < argument > .\ < body >$$

Without a name, a function can be applied to its argument by just writing the function itself and then the argument. Usually, a blank space is left in between. So, $f(1)$ could be written as:

$$(\lambda\, x.\, x + 1)\ \ 1$$

Later, we are going to see how we compute terms like this one, using the same

[b]It is important to note that in pure LC, which we are going to use in this work, we do not have numbers or mathematical operations. So, this term actually is not part of the lambda language (because of the number 1 and the + operation). But, for the sake of understanding the concept of functions, we ignore this for now

notion of how we would compute $f(1)$, i.e., substituting x in the body of the function.

2.3 Syntax

In this section, we present the syntax of λ-terms (LTs). We start with the formal definition, as follows [c]:

Definition (λ-terms)**.** Let V be a countable set of names. The set of λ-terms Λ is the smallest set such that

$$\frac{x \in V}{x \in \Lambda} \quad \text{(VAR)}$$

$$\frac{x \in V \qquad M \in \Lambda}{(\lambda x.M) \in \Lambda} \quad \text{(ABS)}$$

$$\frac{M \in \Lambda \qquad N \in \Lambda}{(M\ N) \in \Lambda} \quad \text{(APP)}$$

The first rule introduces that every variable is an LT. The second rule says that given a variable x and an LT M, the term $(\lambda x.M)$ is an LT, and expresses the notion of function abstraction, being a function that receives one parameter - called x - and returns M as the result. The third rule says that given two LT M and N, $(M\ \ N)$ is an LT, and expresses the notion of function application, meaning that the function M is being called with N as its parameter (MACHADO, 2013). Using this definition, it is possible to recursively write an infinite number of terms. Here are some examples:

1. x
2. y
3. $x\ \ y$
4. $\lambda\, x.\, x$
5. $(\lambda\, x.\, x)\ \ (\lambda\, y.\, y)$
6. $\lambda\, x.\, (x\ \ (\lambda\, y.\, y))$
7. $\lambda\, x.\, x\ \ \lambda\, y.\, y$
8. $\lambda\, x.x\ y$

[c] In this definition, parenthesis are used around the abstractions and applications, but we use it just when it is necessary to avoid ambiguity

9. $x\ y\ z$
10. $\lambda f.((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))$

Note that the terms 5, 6, and 7 are very similar, and only differ from the parenthesis, which dictates the order of the operations. The term 5 is an application of two abstractions and the term 6 is an abstraction of an application. The term 7, however, is not clear. So, when the parentheses are missing, the following rules apply (MACHADO, 2013):

1. The scope of a λ abstraction extends to the rightmost term, until "stoped" by a parentheses. For example:
   $\lambda x.x\ y = \lambda x.(x\ y) \neq (\lambda x.x)\ y$
2. The application is left associative. For Example:
   $x\ y\ z = (x\ y)\ z \neq x\ (y\ z)$

It is important to notice that in this version of λ-Calculus, the pure version, the formalist does not have any type of data, like numbers or booleans, neither it provides any primitive operations, like *sum*, *multiplication*, *and*, *or*, etc. It only has the concept of functions and their application. Everything else is built upon those concepts.

2.3.1 Lambda Terms as Trees

As seen in Machado (2013) and Jung (2004), lambda terms can be seen as trees, with abstractions and applications as internal nodes and variables as leaves. With this representation, ambiguities and the need for parenthesis are eliminated.

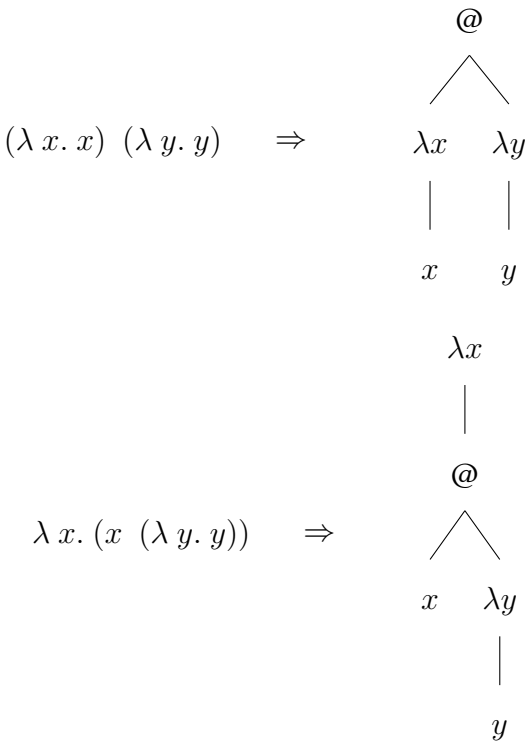
For this, we consider: A variable is a leaf, and an application is a binary node (represented by @) with the left term as the left branch and the right term as the right branch. And the abstraction is a binary node (represented by λ), where the left branch must be a single variable, representing the variable of the term and the right child is an LT, representing the body of the term. Often, the abstraction is seem abbreviated with the node represented by $\lambda\ x$. Table 2.1 shows how to convert from the original notation (string) to the tree notation.

Now we can see that the tree representation of terms 5 and 6 are, in fact, different:

Table 2.1: Convertion table of Lambda Terms from the infix representation to the tree representation.

Type	String	Tree	Abbr.
Var	v	v	
Abs	$(M\ \ N)$	@ M N	
App	$\lambda\, v.\, M$	λ v M	λv M

Source: The authors.



2.3.2 Prefix Notation

The representation of LT as trees eliminates any ambiguity that we may have. But, as subsequent chapters show, a way of representing LT as strings is needed for our work. Prefix notation (aka Polish Notation) is a very useful way to do this since it also removes the need for parenthesis (HAMBLIN, 1962).

So, a 1-to-1 mapping for trees and sequences (strings) is needed. To achieve this, we propose the use of the Polish Notation, traversing the tree with the preorder ordering, where a node is visited, then its left children, and then its right children. This generates terms without parenthesis but adds the application symbol (@), which is implicit in the infix notation when the application is written as the juxtaposition of two lambda terms. We also change the λ symbol for this representation, using the uppercase letter "L". Table 2.2 shows how to convert from the original notation (infix) to the prefix notation.

Table 2.2: Convertion table of Lambda Terms from the infix representation to the prefix representation.

Type	Infix Notation	Tree	Prefix Notation
Var	v	v	v
Abs	$(M\ N)$	@ M N	@ M N
App	$\lambda\, v.\, M$	λ v M	L v M

Source: The authors.

So it is possible to write complex terms without the need for parentheses and without ambiguity. Rewriting the terms 5 and 6 from the previous sections in prefix forms:

$$(\lambda\, x.\, x)\ \ (\lambda\, y.\, y) \equiv @\ \mathrm{L\ x\ x\ L\ y\ y}$$
$$\lambda\, x.\, (x\ \ (\lambda\, y.\, y)) \equiv \mathrm{L\ x\ @\ x\ L\ y\ y}$$

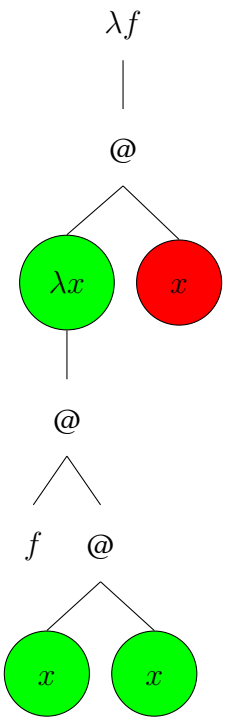
For this work, we consider λ-terms in the three mentioned forms (infix notation, prefix notation, and tree notation) as interchangeable. We use the infix notation for a more user-friendly representation. To enhance visualization of the terms, we utilize tree notation. On the other hand, for the learning tasks outlined in Chapter 1, we use prefix notation. The reason we chose prefix notation for the learning tasks is that it offers a well-organized structure, derived from a tree-like representation. This structure allows

for a clearer and more straightforward representation of expressions. Furthermore, the prefix notation has a distinct advantage over other notations, as it is unambiguous and eliminates the need for parentheses. We expect that this makes it easier to process expressions, particularly for the purposes of learning and understanding complex mathematical concepts.

2.4 Bound vs Free Variables

A variable in a λ-term can be bounded or free, depending on its relative position. This position is relative to the scope of the abstractions of the term. In a term with the format $\lambda x.M$, we say that M is the scope of the parameter x, and every occurrence of x inside M is said to be bound. Otherwise, the variable occurrence is said to be free (MACHADO, 2013).

In other words, if we want to see if a variable x is bound or free, we can look at the term as a tree and go from the variable in question to the root of the tree. If in this path we find a lambda abstraction with x as the variable (λx), we say that this x is bound. Otherwise, it is free. A term can have both bounded and free occurrences of a variable. For instance, in the following term, the two xs in green are bounded by the green abstraction. However, the red x is not bounded by any abstraction, therefore, it is a free variable.



The formal definition of the free variables (FV) of a term is the following:

Definition (Free Variables)**.** $FV(M)$ is the set of free variables in M and can be defined inductively as follows:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.M) &= FV(M) - \{x\} \\ FV(M\ \ N) &= FV(M) \cup FV(N) \end{aligned}$$

In addition, the λ-term M is said to be closed if $FV(M) = \emptyset$, i.e., every variable is bounded by some abstraction. Otherwise, M is said to be open.

2.5 Substitution

The foundation behind function application is the substitution operation. In this operation, we substitute the parameters of the function for the term that is passed as argument (MACHADO, 2013). It is similar to what is done in mathematics when we calculate the result of a function $f(x)$, substituting x by a number inside the body of the function. Here, the syntax is the following:

$$M[x := N] \tag{2.1}$$

meaning that we must substitute every free occurrence of x inside M by N. Here are some examples of substitutions:

1. $x\ [x := z] = z$
2. $y\ [x := z] = y$
3. $\lambda x.y\ [y := z] = \lambda x.z$
4. $\lambda x.y\ [x := z] = \lambda x.y$
5. $\lambda x.x\ [x := z] = \lambda x.x$
6. $\lambda x.y\ [y := \lambda y.y] = \lambda x.\lambda y.y$

7. $\lambda x.y\ [y := x] = \lambda x.x \quad ?$

Notice that some terms remain the same after the substitution. That is because there are no occurrences of the substitution variable free in that term. Also, substitution 7 has a question mark because the substitution captured the new variable as bound, and it should not be considered a valid substitution. This problem is called *capture of free variable*. The formal definition of substitution is as follows:

Definition (Substitution)**.**

$$x[y := N] = \begin{cases} N & \text{if } x = y \\ x & \text{if } x \neq y \end{cases}$$

$$(\lambda x.M)[y := N] = \begin{cases} \lambda x.M & \text{if } x = y \\ \lambda x.(M[y := N]) & \text{if } x \neq y \text{ and } x \notin FV(N) \end{cases}$$

$$(M\ P)[y := N] = M[y := N]\ P[y := N]$$

The only source for the capture of a free variable is when we apply a substitution to a function abstraction. The definition above for $(\lambda x.M)[y := N]$ avoids the capture of a free variable by simply leaving undefined the result of performing the substitution in case $x \in FV(N)$.

2.6 Alpha Equivalence

Let us start this section with two examples of mathematical functions:

$$f(x) = x^2 - 4$$
$$g(z) = z^2 - 4$$

If we take the pairs $(x, f(x))$ and $(z, g(z))$, or if we take a look at their graphs, we can see that both f and g represent the same mathematical function - a function that takes one argument, square it, and subtract 4. The name of the parameter is unimportant for the definition of the function. This also happens in LC. If we take the two functions:

$$\lambda x.x\ y$$

$$\lambda z.z\ y$$

We can see that they both represent the same function - a function that takes one argument and applies it to y. However, they are syntactically different terms. The α-equivalence states that terms that differ only by the choice of names for the bound variables are considered equivalent to each other (MACHADO, 2013). We write $M =_\alpha N$ to state that M and N are α-equivalents, and $M \neq_\alpha N$ to state otherwise. The formal definition is as follows:

Definition (α-equivalence)**.** The α-equivalence is the smallest equivalence relation on Lambda terms such that

$$\frac{y \notin FV(M)}{\lambda x.M =_\alpha \lambda y.(M[x := y])}\ (\alpha)$$

$$\frac{M =_\alpha M'}{M\ N =_\alpha M'\ N}$$

$$\frac{N =_\alpha N'}{M\ N =_\alpha M\ N'}$$

$$\frac{M =_\alpha M'}{\lambda x.M =_\alpha \lambda x.M'}$$

The first rule (axiom α) denotes the idea of equivalence of terms by only changing the bounded names. The other rules denote that this equivalence on a subterm extends for the whole term.

Some authors, including Machado (2013), differentiates *pre-terms* from *terms*. The pre-terms are what its been presented as terms and the terms are actually an equivalence class of α-equivalent pre-terms. In this work, we use the word *term* indistinctly to address both pre-terms and terms.

The same operations defined before extend now to the α-equivalent class. So, now we can perform the substitution $\lambda x.y\ [y := x]$, since we can beforehand change the name of the bound variables.

In this work, however, we use the Barendregt convention (BARENDREGT, 1984). This convention states that, for the sake of simplicity, we can assume that the name of the bound variables will always be unique. This assumption eliminates the possibility of capturing free variables and allows the definition for the substitution for the lambda abstraction to be much simpler: $(\lambda x.M)[y := N] = \lambda x(M[y := N])$

2.7 Beta Reduction

The β-reduction captures the notion of the function application, using the substitution operation. But before the definition of the β-reduction, the definition of a *redex* must be given. Informally, a redex is a part of a term where a substitution can occur, i.e., we have an λ-abstraction followed by any other term. Formally:

Definition (*Redex*)**.** A *redex* (reducible expression) is any subterm in the format

$$(\lambda x.M)\ N$$

for which the respective contractum is

$$M[x := N]$$

In addition, if a term does not have any redexes, the term is a normal form. Otherwise, the term is reducible.

Definition (β-reduction)**.** The β-reduction ($\rightarrow_\beta$) is the smallest equivalence relation on Lambda terms such that

$$(\beta)\ \ (\lambda x.M)\ N \rightarrow_\beta M[x := N]$$

$$\frac{M \rightarrow_\beta M'}{M\ N \rightarrow_\beta M'\ N}$$

$$\frac{N \rightarrow_\beta N'}{M\ N \rightarrow_\beta M\ N'}$$

$$\frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}$$

The $\to_\beta$ can be seen as just one-step computation and is related to the hypotheses **H1**. The multi-step reduction is denoted by $\twoheadrightarrow_\beta$ and is related to the hypotheses **H2**. It is defined as the reflexive and transitive closure of $\to_\beta$, as follows:

Definition (Multi-step β-reduction)**.** The multi-step β-reduction ($\twoheadrightarrow_\beta$) is the smallest relation on Lambda terms such that

$$M \twoheadrightarrow_\beta M$$

$$\frac{M \to_\beta N}{M \twoheadrightarrow_\beta N}$$

$$\frac{M \to_\beta N \qquad N \to_\beta P}{M \twoheadrightarrow_\beta P}$$

Another relation between terms is the β-equivalence, which gives the notion of terms that have the "same value". It is defined as follows:

Definition (β-equivalence)**.** $=_\beta$ is the smallest relation on terms such that

$$\frac{M \twoheadrightarrow_\beta P \qquad N \twoheadrightarrow_\beta P}{M =_\beta N}$$

A term can have a normal form, i.e., be reducible with β-reductions until it reaches its normal form, but it can also not have a normal form, i.e., it can enter a loop and never reach a normal form. For example, the term $(\lambda x.x\ x)\ (\lambda x.x\ x)$ does not have a normal form, because it β-reduces to itself. We say that a term M has a normal form when exists a term N such that $M =_\beta N$ and N is a normal form.

Also, a term can have more than one redex, meaning that when we try to apply the β-reduction on a term, we can have multiple possibilities. It is useful to have a strategy to select which redex we want to reduce at each step of the computation. Formally:

Definition (Evalutation Strategy)**.** An evaluation strategy is a function that chooses a single redex for every reducible term.

The two most usual evaluation strategies are: (i) *lazy evaluation*, where the redex chosen is the leftmost, outermost redex of a term; and (ii) *strict evaluation*, where the redex chosen is the leftmost, innermost redex of a term.

Choosing the evaluation strategy is very important to clearly define which redex to reduce through the β-reduction. Furthermore, it is not just a matter of personal preference, since there is a theorem that says that if a term M has a normal form P, then the *lazy evaluation strategy* will always reach P from M, in a finite number of β-reductions. Therefore, in this work, we always use the *lazy evaluation strategy* when performing β-reductions on terms, to assure that, if the term has a normal form, we are able to reach it.

2.8 Encodings

Lambda terms can be used to represent abstract ideas, such as numbers, lists, boolean formulas, structures, trees, etc. The notion of encoding is well-known in Computer Science. For example, our modern computers operate on binary code, i.e., inside our computers, there are only zeroes and ones. But with only two digits, we are able to represent integers, floating points, strings, trees, etc, by defining some encodings. For instance, in 8 bits, 00000001 can represent the integer 1, and 10000000 the integer 128. But we can use two's complement to also represent negative numbers and now 10000000 represents the number -128 (SHUTE, 1993). There is not a "right" encoding, just different encoding and patterns that usually every computer architecture follows.

In LC, the idea is the same. We can use the structure of function abstractions and applications to encode representations for numbers, booleans, strings, etc. As in binary, there is no "right" encoding, just different ones. But there are some well-known encodings, such as Church encoding or Scott encoding. In this work, we utilize the Church Encoding as it is the most widely recognized codification, and its encoding for the desired data and operations is simple and straightforward.

For instance, Church defined numbers in the following encoding:

$$\begin{aligned} \lambda f.\ \lambda x.\ x &\equiv 0 \\ \lambda f.\ \lambda x.\ f\ x &\equiv 1 \\ \lambda f.\ \lambda x.\ f(f\ x) &\equiv 2 \\ \lambda f.\ \lambda x.\ f(f(f\ x)) &\equiv 3 \\ &\vdots \end{aligned} \tag{2.2}$$

And boolean values can be defined as:

$$\begin{aligned} \lambda a.\ \lambda b.\ a &\equiv true \\ \lambda a.\ \lambda b.\ b &\equiv false \end{aligned} \tag{2.3}$$

Notice that $false$ has the same encoding as 0 - their encodings are α-equivalent. This is not a problem, since for binary this also happens. 01100001 can represent the number 97 or the letter a, depending on the context. So, if we are working with booleans, $\lambda f.\ \lambda x.\ x$ means $false$, and if we are working with numbers, $\lambda f.\ \lambda x.\ x$ means 0.

With the encodings defined, we can write algorithms, in the form of functions, that operate on those data. Here, again there is not a "right" function for an operation. We can define functions over the numbers defined, such as the *successor*, *addition*, and *multiplication* operations, as:

$$\begin{aligned} \lambda n.\ \lambda a.\ \lambda b.\ a\ (n\ a\ b) &\equiv succ \\ \lambda n.\ \lambda m.\ \lambda a.\ \lambda b.\ (n\ a\ (m\ a\ b)) &\equiv add \\ \lambda n.\ \lambda m.\ \lambda a.\ \lambda b.\ (n\ (m\ a)\ b) &\equiv mult \end{aligned} \tag{2.4}$$

We can also define functions over booleans, such as:

$$\begin{aligned} \lambda p.\ \lambda q.\ (p\ q\ p) &\equiv and \\ \lambda p.\ \lambda q.\ (p\ p\ q) &\equiv or \\ \lambda p.\ \lambda a.\ \lambda b.\ (p\ b\ a) &\equiv not \end{aligned} \tag{2.5}$$

In the context of this work, a computation is considered a "meaningful computation" only if it is based on some known encoding.

2.9 Computation

Now we can demonstrate how we can use the λ-C to execute some computations, using the lazy evaluation strategy.

Applying the addition with 2 and 3, for example:

$$
\begin{array}{rll}
 & (\lambda\, n.\lambda\, m.\lambda\, a.\lambda\, b.n\ a\ (m\ a\ b))\ (\lambda\, f.\lambda\, x.f\ (f\ x))\ (\lambda\, f.\lambda\, x.f\ (f\ (f\ x))) & \\
\rightarrow_\beta & (\lambda\, m.\lambda\, a.\lambda\, b.(\lambda\, f.\lambda\, x.f\ (f\ x))\ a\ (m\ a\ b))\ (\lambda\, f.\lambda\, x.f\ (f\ (f\ x))) & \\
\rightarrow_\beta & \lambda\, a.\lambda\, b.(\lambda\, f.\lambda\, x.f\ (f\ x))\ a\ ((\lambda\, f.\lambda\, x.f\ (f\ (f\ x)))\ a\ b) & \\
\rightarrow_\beta & \lambda\, a.\lambda\, b.(\lambda\, x.a\ (a\ x))\ ((\lambda\, f.\lambda\, x.f\ (f\ (f\ x)))\ a\ b) & \\
\rightarrow_\beta & \lambda\, a.\lambda\, b.a\ (a\ ((\lambda\, f.\lambda\, x.f\ (f\ (f\ x)))\ a\ b)) & \\
\rightarrow_\beta & \lambda\, a.\lambda\, b.a\ (a\ ((\lambda\, x.a\ (a\ (a\ x)))\ b)) & \\
\rightarrow_\beta & \lambda\, a.\lambda\, b.a\ (a\ (a\ (a\ (a\ b)))) & \\
= & 5 & \text{by def. } (2.2)
\end{array}
$$

And applying the multiplication with 2 and 3:

$$
\begin{array}{rll}
 & (\lambda\, n.\lambda\, m.\lambda\, a.\lambda\, b.n\ (m\ a)\ b)\ (\lambda\, f.\lambda\, x.f\ (f\ x))\ (\lambda\, f.\lambda\, x.f\ (f\ (f\ x))) & \\
\rightarrow_\beta & (\lambda\, m.\lambda\, a.\lambda\, b.(\lambda\, f.\lambda\, x.f\ (f\ x))\ (m\ a)\ b)\ (\lambda\, f.\lambda\, x.f\ (f\ (f\ x))) & \\
\rightarrow_\beta & \lambda\, a.\lambda\, b.(\lambda\, f.\lambda\, x.f\ (f\ x))\ ((\lambda\, f.\lambda\, x.f\ (f\ (f\ x)))\ a)\ b & \\
\rightarrow_\beta & \lambda\, a.\lambda\, b.(\lambda\, x.(\lambda\, f.\lambda\, x.f\ (f\ (f\ x)))\ a\ ((\lambda\, f.\lambda\, x.f\ (f\ (f\ x)))\ a\ x))\ b & \\
\rightarrow_\beta & \lambda\, a.\lambda\, b.(\lambda\, f.\lambda\, x.f\ (f\ (f\ x)))\ a\ ((\lambda\, f.\lambda\, x.f\ (f\ (f\ x)))\ a\ b) & \\
\rightarrow_\beta & \lambda\, a.\lambda\, b.(\lambda\, x.a\ (a\ (a\ x)))\ ((\lambda\, f.\lambda\, x.f\ (f\ (f\ x)))\ a\ b) & \\
\rightarrow_\beta & \lambda\, a.\lambda\, b.a\ (a\ (a\ ((\lambda\, f.\lambda\, x.f\ (f\ (f\ x)))\ a\ b))) & \\
\rightarrow_\beta & \lambda\, a.\lambda\, b.a\ (a\ (a\ ((\lambda\, x.a\ (a\ (a\ x)))\ b))) & \\
\rightarrow_\beta & \lambda\, a.\lambda\, b.a\ (a\ (a\ (a\ (a\ (a\ b))))) & \\
= & 6 & \text{by def. } (2.2)
\end{array}
$$

Applying the boolean operations in the boolean expression $(true\ \ and\ \ (notfalse))$. To translate to the LC, we have to rewrite the operands as functions as follows: $(and\ true\ (not\ \ false)\ ))$.

$$
\begin{array}{rll}
 & (\lambda p.\ \lambda q.\ (p\ q\ p))\ (\lambda a.\ \lambda b.\ a)\ ((\lambda p.\ \lambda a.\ \lambda b.\ (p\ b\ a))\ (\lambda a.\ \lambda b.\ b)) & \\
\rightarrow_\beta & (\lambda q.\ ((\lambda a.\ \lambda b.\ a)\ q\ (\lambda a.\ \lambda b.\ a)))\ ((\lambda p.\ \lambda a.\ \lambda b.\ (p\ b\ a))\ (\lambda a.\ \lambda b.\ b)) & \\
\rightarrow_\beta & (\lambda a.\ \lambda b.\ a)\ ((\lambda p.\ \lambda a.\ \lambda b.\ (p\ b\ a))\ (\lambda a.\ \lambda b.\ b))\ (\lambda a.\ \lambda b.\ a) & \\
\rightarrow_\beta & (\lambda b.\ ((\lambda p.\ \lambda a.\ \lambda b.\ (p\ b\ a))\ (\lambda a.\ \lambda b.\ b)))\ (\lambda a.\ \lambda b.\ a) & \\
\rightarrow_\beta & (\lambda p.\ \lambda a.\ \lambda b.\ (p\ b\ a))\ (\lambda a.\ \lambda b.\ b) & \\
\rightarrow_\beta & \lambda a.\ \lambda b.\ ((\lambda a.\ \lambda b.\ b)\ b\ a) & \\
\rightarrow_\beta & \lambda a.\ \lambda b.\ ((\lambda b.\ b)\ a) & \\
\rightarrow_\beta & \lambda a.\ \lambda b.\ a & \\
= & true & \text{by def. } (2.3)
\end{array}
$$

It is easy to get lost in the parenthesis when doing large computations. Here is the previous example using the prefix notation:

$$
\begin{array}{rll}
 & @\,@\,L\,p\,L\,q\,@\,@\,p\,q\,p\,L\,a\,L\,b\,a\,@\,L\,p\,L\,a\,L\,b\,@\,@\,p\,b\,a\,L\,a\,L\,b\,b & \\
\rightarrow_\beta & @\,L\,q\,@\,@\,L\,a\,L\,b\,a\,q\,L\,a\,L\,b\,a\,@\,L\,p\,L\,a\,L\,b\,@\,@\,p\,b\,a\,L\,a\,L\,b\,b & \\
\rightarrow_\beta & @\,@\,L\,a\,L\,b\,a\,@\,L\,p\,L\,a\,L\,b\,@\,@\,p\,b\,a\,L\,a\,L\,b\,b\,L\,a\,L\,b\,a & \\
\rightarrow_\beta & @\,L\,b\,@\,L\,p\,L\,a\,L\,b\,@\,@\,p\,b\,a\,L\,a\,L\,b\,b\,L\,a\,L\,b\,a & \\
\rightarrow_\beta & @\,L\,p\,L\,a\,L\,b\,@\,@\,p\,b\,a\,L\,a\,L\,b\,b & \\
\rightarrow_\beta & L\,a\,L\,b\,@\,@\,L\,a\,L\,b\,b\,b\,a & \\
\rightarrow_\beta & L\,a\,L\,b\,@\,L\,b\,b\,a & \\
\rightarrow_\beta & L\,a\,L\,b\,a & \\
= & \mathit{true} & \text{by def. (2.3)}
\end{array}
$$

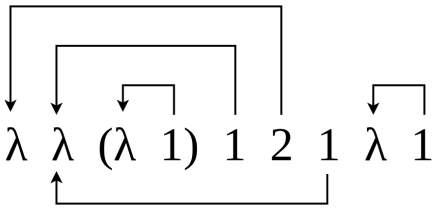
2.10 De Bruijn Index

The De Bruijn (DB) index is a tool to define λ-terms without having to name the variables (BRUIJN, 1972). This eliminates the need to worry about variable names when performing a substitution and the need for the alpha-equivalence definition. This approach can be beneficial for us, since the terms are agnostic to the variable naming and are simpler, in the sense that they are shorter.

Basically, in this notation, the variable names are replaced by natural numbers. The abstraction no longer has a variable name, and every occurrence of a variable is represented by a number, that indicates at which abstraction it is binded. These nameless terms are called *de Bruijn terms*, and the numeric variables are called *de Bruijn indices* (PIERCE, 2002). For the sake of simplicity, we denote the free variables with the number 0, and the indices of the bound variables start at 1. This notation works by assuming that each *de Bruijn indice* corresponds to the number of binders (abstractions) that the variable is under. In other words, the abstraction that the variable is bound can be found by counting the abstractions from the leaves to the root of the tree. Figure 2.1 illustrates how the indices are calculated based on each variable position. Some examples of de Bruijn terms are:

1. 0
2. $0\ \ 0$
3. $\lambda\ 1$
4. $(\lambda\ 1)\ \ (\lambda\ 0)$
5. $\lambda\ (\lambda\ (1\ 2))$
6. $\lambda\ ((\lambda\ 2\ (1\ 1))\ (\lambda\ 2\ (1\ 1)))$
7. $\lambda\ (\lambda\ 1\ (\lambda\ 1))\ (\lambda\ 2\ 1))$

Figure 2.1: Figure illustrating how the de Bruijn indexes are calculated, based on their relative positions.



Source: The authors.

The syntax for the de Bruijn terms is similar to the traditional λ-Calculus, and can be formalized as:

Definition (de Bruijn λ-terms)**.** The set of de Bruijn λ-terms Λ_{DB} is the smallest set such that

$$\frac{n \in \mathbb{N}}{n \in \Lambda_{DB}} \quad \text{(VAR)}$$

$$\frac{M \in \Lambda_{DB}}{(\lambda\ M) \in \Lambda_{DB}} \quad \text{(ABS)}$$

$$\frac{M \in \Lambda_{DB} \qquad N \in \Lambda_{DB}}{(M\ N) \in \Lambda_{DB}} \quad \text{(APP)}$$

The β-reduction on this notation is not trivial, since the operation changes the relative position of the variables inside the terms. So, the indices must be recalculated for every reduction. Although one of the goals of this work is for the model to learn to perform the β-reduction over de Bruijn terms, for our dataset generation, we do not perform the β-reduction directly on de Bruijn terms. Rather, we convert them to the traditional notation, perform the reduction and then convert them back to de Bruijn. So, for the sake of simplicity, we do not explore the definition for the β-reduction on de Bruijn terms.

2.10.1 Conversion Algorithms

Since we could not find an algorithm for converting between traditional notation and De Bruijn notation, we propose the following two algorithms:

```
1  def to_de_bruijn(term, stack=[]):
2      if isinstance(term, Var):
3          i = get_index(stack, term)
4          return Var(i)
5      elif isinstance(term, Abs):
6          new = stack.copy()
7          new.append(term.var)
8          return Abs(Var('_'), to_de_bruijn(term.body, new))
9      elif isinstance(term, App):
10         return App(to_de_bruijn(term.left, stack), to_de_bruijn(term.right, stack))
```

The above algorithm is a recursive algorithm, that traverses the lambda term tree, deleting the variable from the abstractions and substituting the variable names in their bodies for DB indexes. It has a stack to store the order that the variables appeared on the term. The algorithm works as follows: on lines 2, 4, and 8, it checks if the current node is a variable, abstraction, or application, respectively. If it is a variable, on line 3 it gets an index from the stack, based on the current node using the function *get_index*, which returns the position of the variable on the stack. Then, it returns the index calculated as a variable on line 4. If it is an abstraction, on line 6 it creates a copy of the current stack for this recursive execution, then on line 7 it appends the variable of the abstraction to the new copy of the stack and on line 8 it returns the abstraction with its variable deleted and the recursion applied to its body. Finally, if it is an application, on line 10 it just continues the recursion for both terms, left and right.

```
1  def from_de_bruijn(term_tree, stack=[], c=0):
2      if is_variable(term_tree):
3          v = get_var(stack, index(term_tree))
4          return v
5      elif is_abstraction(term_tree):
6          v = alphabet[c]
7          stack.append(v)
8          return Abs(v, from_de_bruijn(term_tree.body, stack, c + 1))
9      elif is_application(term_tree):
10         return App(from_de_bruijn(term_tree.left, stack, c),
11                    from_de_bruijn(term_tree.right, stack, c))
```

The above algorithm is also a recursive algorithm that traverses the lambda term tree, assigning variable names for the abstractions and substituting the DB indexes based on their corresponding bindings. It has a stack to store the names of the variables for all the abstractions the current node is under and a counter to store the position of the

next variable name we can use from the alphabet. The algorithm works as follows: on lines 2, 5, and 9, it checks if the current node is a variable, abstraction, or application respectively. If it is a variable, i.e., a DB index, on line 3, it gets a variable from the stack, using the function *get_var*, which returns the variable in the stack that is on the index of the current node. Then, it returns it on line 4. If it is an abstraction, on line 6 it assigns a new variable name from the alphabet to the current node, then on line 7, it appends this variable to the stack. Then, on line 8 it continues the recursion on the body of the abstraction, increasing the counter since it used a variable from the alphabet. If it is an application, on lines 10 and 11 it just continues the recursion for both terms, left and right.

For this algorithm, we must define two things: an alphabet and the order of its elements. For this work, we use the English alphabet, both lowercase and uppercase, minus the uppercase letter "L", which we use for the lambda symbol. For the order, we can define any order we want. In this work, we use two orders: (i) the alphabetical order, where the next variable name to be used is always the next variable in the alphabet and (ii) the random order, where the next variable name to be used is randomly assigned.

2.10.2 Computation

As we have done with the traditional notation, with de Bruijn notation we can also write the terms in prefix notation. Here is the computation example from the previous section in the de Bruijn prefix notation:

$$
\begin{aligned}
& @\,@\,\mathrm{L}\,\mathrm{L}\,@\,@\,2\,1\,2\,\mathrm{L}\,\mathrm{L}\,2\,@\,\mathrm{L}\,\mathrm{L}\,\mathrm{L}\,@\,@\,3\,1\,2\,\mathrm{L}\,\mathrm{L}\,1 \\
\rightarrow_\beta \quad & @\,\mathrm{L}\,@\,@\,\mathrm{L}\,\mathrm{L}\,2\,1\,\mathrm{L}\,\mathrm{L}\,2\,@\,\mathrm{L}\,\mathrm{L}\,\mathrm{L}\,@\,@\,3\,1\,2\,\mathrm{L}\,\mathrm{L}\,1 \\
\rightarrow_\beta \quad & @\,@\,\mathrm{L}\,\mathrm{L}\,2\,@\,\mathrm{L}\,\mathrm{L}\,\mathrm{L}\,@\,@\,3\,1\,2\,\mathrm{L}\,\mathrm{L}\,1\,\mathrm{L}\,\mathrm{L}\,2 \\
\rightarrow_\beta \quad & @\,\mathrm{L}\,@\,\mathrm{L}\,\mathrm{L}\,\mathrm{L}\,@\,@\,3\,1\,2\,\mathrm{L}\,\mathrm{L}\,1\,\mathrm{L}\,\mathrm{L}\,2 \\
\rightarrow_\beta \quad & @\,\mathrm{L}\,\mathrm{L}\,\mathrm{L}\,@\,@\,3\,1\,2\,\mathrm{L}\,\mathrm{L}\,1 \\
\rightarrow_\beta \quad & \mathrm{L}\,\mathrm{L}\,@\,@\,\mathrm{L}\,\mathrm{L}\,1\,1\,2 \\
\rightarrow_\beta \quad & \mathrm{L}\,\mathrm{L}\,@\,\mathrm{L}\,1\,2 \\
\rightarrow_\beta \quad & \mathrm{L}\,\mathrm{L}\,2 \\
= \quad & \mathit{true} \qquad\qquad \text{by def. (2.3)}
\end{aligned}
$$

As we can see, this notation is shorter than the traditional notation. Also, we do not have to worry about the names of the variables. However, the β-reduction is harder on this notation. So, we use this notation to see the difference in performance for our model to learn the β-reduction over different representations of terms.

3 ARTIFICIAL INTELLIGENCE

Artificial Intelligence (AI) is an interdisciplinary field that focuses on the development of intelligent models that can perform tasks that typically require human intelligence, such as perception, reasoning, and decision-making. AI systems can be designed to learn from experience and improve their performance over time, leading to the development of various applications in areas such as healthcare, finance, transportation, natural language applications, etc.

In this chapter, we present an overview of the specific fields and methods of AI we use in this work. We start by presenting the Machine Learning subfield, followed by a presentation on Neural Networks. Next, we introduce the Neurosymbolic domain. We then proceed to discuss the limitations of purely neural architectures and examine sequence-to-sequence models. Finally, we provide a description of the Transformer, which is the model we use in this research.

3.1 Machine Learning

Machine Learning (ML) is a subset of AI that involves the development of algorithms and models that can learn from data to make predictions or decisions. ML algorithms can be trained on vast amounts of data, allowing them to identify patterns and relationships in the data and improve their accuracy over time (GOODFELLOW; BENGIO; COURVILLE, 2016). There are three main types of ML algorithms: supervised learning, unsupervised learning, and reinforcement learning (BISHOP; NASRABADI, 2006).

Supervised learning algorithms are trained on data where the output or target variable is known. These algorithms can be used to make predictions about new, unseen data, such as classifying images or predicting stock prices. Unsupervised learning algorithms are also trained on data, but the output or target variable is unknown. These algorithms can be used to identify patterns and relationships in the data, such as clustering data into groups or detecting anomalies in data. Reinforcement learning algorithms are designed to learn from interactions with an environment, where the algorithm receives a reward or penalty for each action it takes. These algorithms can be used in various applications, such as game-playing and robotics. This work focuses on supervised learning, particularly on connectionist AI (neural network models).

3.2 Neural Networks

Artificial Neural Networks (NNs) were inspired by the structure and function of the human brain and are designed for processing large amounts of data to identify patterns and relationships. Their fundamental unit is the Neuron, which essentially "activates" when a linear combination of its inputs surpasses a certain threshold. A Neural Network is merely a collection of interconnected neurons whose properties are determined by the arrangement of the neurons and their individual characteristics (RUSSELL; NORVIG, 2021).

These neurons are often organized in layers. The input data is fed into the first layer, and the output of each neuron in a given layer is used as the input for the next layer until the final layer produces the output of the network. The connections between the neurons are represented by weights that are updated during the training process to minimize the error between the predicted output and the actual output.

NNs have been applied to a wide range of tasks, including image classification, speech recognition, and natural language processing, among others. One of the main advantages of NNs is their ability to model non-linear relationships between inputs and outputs. This makes NNs a powerful tool for solving complex real-world problems.

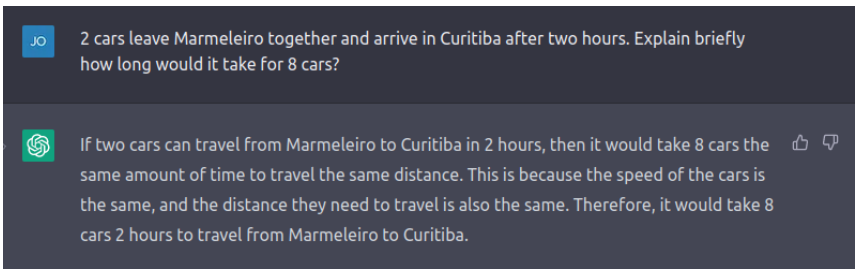
3.3 Neurosymbolic AI

As seen in the introduction of this work, even state-of-the-art neural models have trouble with symbolic reasoning. One example is the *ChatGPT* [a], which utilizes an advanced AI model based on the Transformer architecture, known as GPT-3, developed by Brown et al. (2020). According to Roose (2022), this chatbot has the ability to provide answers to inquiries on diverse topics. As shown in Figure 3.1, it also can handle some symbolic reasoning. Nevertheless, the capacity for symbolic reasoning remains restricted, as it generates incorrect responses to elementary questions, as seen in Figure 3.2. Even within our domain of study, the chatbot tries to answer questions, but it presents difficulty in giving the correct response, as illustrated in Figure 3.3.

What we can devise from this is that even though the chatbot is excellent at answering general questions, it does not perform well when asked to carry out symbolic reasoning. So, an architecture that takes advantage of both symbolic AI and neural mod-

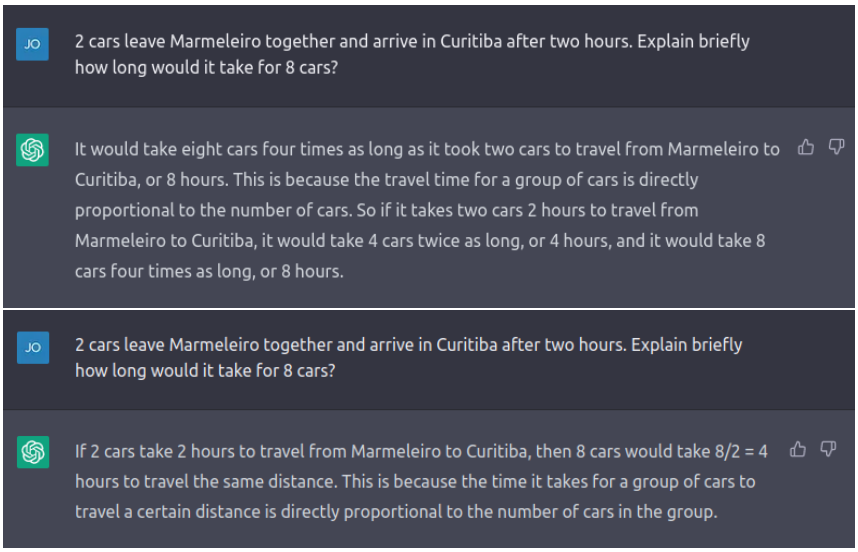
[a] available at <https://openai.com/blog/chatgpt/>

Figure 3.1: An example of a question answered right by a state-of-the-art chatbot, the ChatGPT. The reasoning behind the answer is correct and probably based on statistical data that the model was trained on.



Source: Image generated by the authors, at <https://openai.com/blog/chatgpt/>, on 05/01/2023

Figure 3.2: An example of a question answered incorrectly twice by a state-of-the-art chatbot, the ChatGPT. The correct answer should be two hours, not eight. The reasoning behind the answers is incorrect and probably based on statistical data that the model was trained on.



Source: Image generated by the authors, at <https://openai.com/blog/chatgpt/>, on 05/01/2023

els could still give good general answers and also present better symbolic reasoning.

The Neurosymbolic AI is a field of artificial intelligence that combines the strengths of symbolic and connectionist AI. Symbolic AI represents knowledge in a structured, human-readable form and uses reasoning and rule-based systems to perform tasks. Connectionist AI, on the other hand, represents knowledge as patterns in a network of simple processing units to learn from data. The neurosymbolic models aim to merge the two approaches by incorporating symbolic reasoning and/or representation with the learning and generalization capabilities of neural networks. The following quote from the book Garcez, Lamb and Gabbay (2009) summarizes the idea:

Figure 3.3: An example of a lambda term evaluated wrongly by a state-of-the-art chatbot, the ChatGPT. The right answer should be “lambda a. lambda b. a”.

Source: Image generated by the authors, at <https://openai.com/blog/chatgpt/>, on 05/01/2023

> The aim of neural-symbolic computation is to explore the advantages that each paradigm presents. Among the advantages of artificial neural networks are massive parallelism, fault tolerance (robust learning), efficient inductive learning, and effective generalization capabilities. On the other hand, symbolic systems provide descriptions (as opposed to only discriminations); can explain their inference process, for example through automatic theorem proving; and use powerful declarative languages for knowledge representation and reasoning.

In Kautz (2022), six different forms of neurosymbolic AI are presented, accordingly to how and where the two different approaches are combined. In the present work, we use the *Neuro: Symbolic* $\rightarrow$ *Neuro* [b] approach, meaning that we take a symbolic domain (the λ-calculus reductions) and apply it to a neural architecture (the Transformer).

[b] Using Kautz (2022) notation.

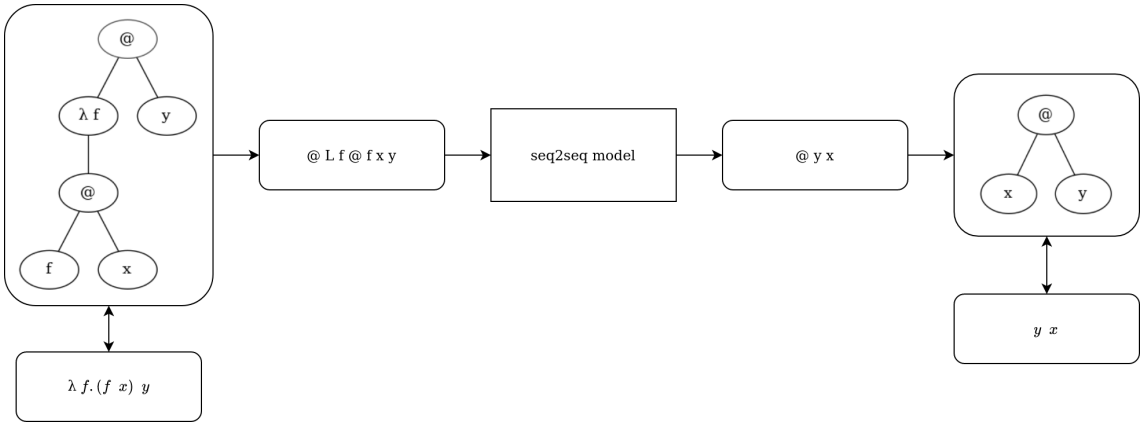
3.4 Sequence-to-sequence

Although neural networks are versatile and effective, they are only suitable for problems where inputs and targets can be represented by fixed-dimensional vector encodings. This is a significant constraint, as many crucial problems are better expressed using sequences of unknown lengths, such as speech recognition and machine translation. It is evident that a versatile method that can learn to translate sequences to sequences without being restricted to a specific domain would be valuable. (SUTSKEVER; VINYALS; LE, 2014).

Sequence-to-sequence (seq2seq) models, emerged from this necessity. They are a type of deep learning model used for tasks that involve mapping an input sequence to an output sequence of variable length. They have been traditionally applied to various natural language processing tasks, such as machine translation, text summarization, and text generation, among others.

Given that we can see the computations in lambda calculus as strings transformations, we can look at the computations we want to learn in this work as machine translation tasks. Also, these strings do not have fixed sizes. So, we have opted to employ the sequence-to-sequence model for our work. Figure 3.4 shows the general layout for our algorithm.

Figure 3.4: General scheme of the seq2seq model applied on the One-Step Beta Reduction task.



Source: The authors.

Among the different architectures for assembling seq2seq models, the most common are the Recurrent Neural Networks (RNN) (LIPTON; BERKOWITZ; ELKAN, 2015), the Long Short-Term Memory (LSTM) (SUTSKEVER; VINYALS; LE, 2014), the Gated Re-

current Unit (GRU) (CHUNG et al., 2014) and the Transformer (VASWANI et al., 2017). In this work, we chose to use the Transformer.

3.5 Transformer

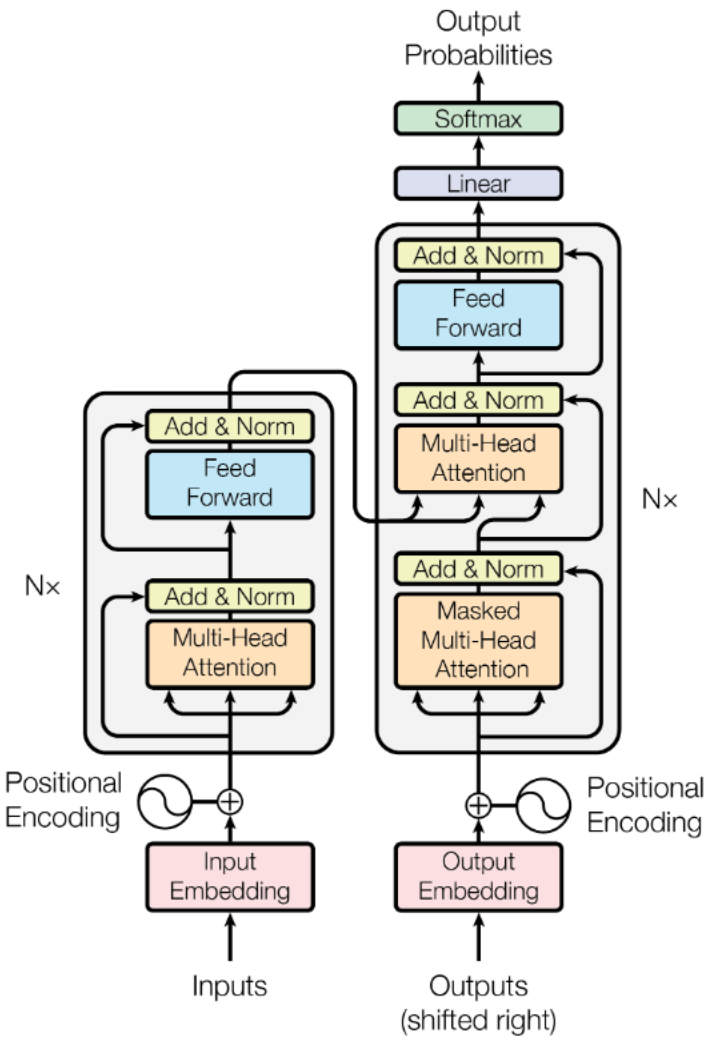
The Transformer model is a type of neural network architecture that was introduced in Vaswani et al. (2017). It is designed to handle sequential data, such as natural language, and has quickly become one of the most popular models for tasks such as natural language processing, machine translation, text classification, and question answering. One of the key innovations of the Transformer model is its use of a self-attention mechanism, which allows the model to dynamically weigh the importance of different parts of the input sequence. This allows the Transformer to capture long-range dependencies in the data, which is particularly useful for processing sequences of variable lengths. Another advantage of the Transformer is its parallelization capacity, which allows it to be trained efficiently on large amounts of data. The Transformer model can be trained in parallel on multiple sequences, which is not possible with other traditional sequence-to-sequence models.

Now a detailed description of the model is presented, including its architecture and internal operation. In terms of architecture, the Transformer consists of an encoder and a decoder, both of which are composed of a series of multi-head self-attention layers and fully connected feed-forward layers, which can be seen in Figure 3.5. The encoder and decoder have slightly different architectures, with the decoder also including an additional masked self-attention mechanism that prevents it from seeing future positions in the sequence.

In the Transformer, each input token is represented as a fixed-length embedding vector. However, these embeddings do not encode any information about the position of the token within the sequence, which can be important for capturing the sequential relationships between different parts of the data. To address this issue, the Transformer uses positional encoding, which adds positional information to the input embeddings.

The encoder works by taking as input a sequence of tokens, such as words or characters, and processing it into a sequence of hidden states that capture the meaning of the input sequence. To achieve this, each self-attention layer computes a weighted sum of the input tokens, with the weights determined by a learned attention mechanism. This allows the model to dynamically weigh the importance of different parts of the

Figure 3.5: Model architecture of the Transformer. The encoder is represented on the left, and the decoder on the right.



Source: Vaswani et al. (2017)

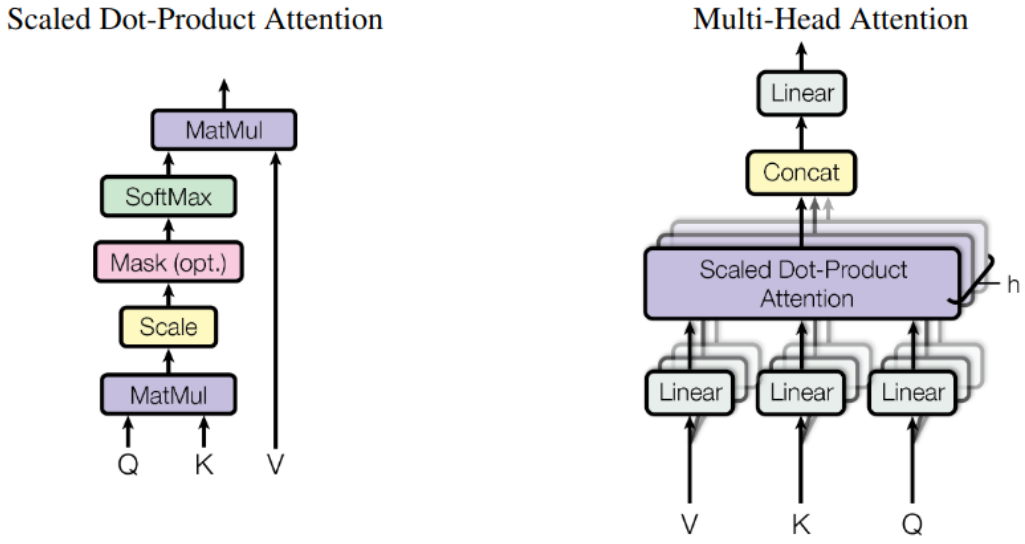
input sequence, allowing it to capture long-range dependencies in the data. The output of each self-attention layer is then passed through a feed-forward layer, which provides a non-linear transformation of the data.

The decoder works by taking the encoded representation of the source sentence and the output from the previous time step as input and generating the output sequence one token at a time. The decoder consists of multiple identical layers, each of which contains three sublayers: Masked Multi-Head Attention, Multi-Head Attention, and Feed-Forward Network. After passing through these three sublayers, the decoder output is passed through a softmax activation function to obtain a probability distribution over the vocabulary. The token with the highest probability is selected as the next output token. This process is repeated for each time step until the end-of-sequence token is

generated or a maximum output length is reached.

The self-attention layer allows the model to capture dependencies between different parts of the input sequence and is shown in Figure 3.6 (left). It takes three inputs: queries, keys, and values. These are linear projections of the input sequence, which are learned during training. The queries, keys, and values are used to compute an attention weight matrix, which determines how much each position in the input sequence attends to every other position. The attention weight matrix is computed by taking the dot product of the queries with the keys and applying a softmax function to the result. This produces a probability distribution over the keys for each query, indicating how much each key attends to the query. The values are then weighted by the attention weights and summed, producing a weighted sum of the values that represent the attended input sequence.

Figure 3.6: Architecture of the self-attention layer in the Transformer architecture.



Source: Vaswani et al. (2017)

The self-attention layer also includes a mechanism called multi-head attention, which allows the model to attend to multiple aspects of the input sequence, which can be seen in Figure 3.6 (right). This is achieved by splitting the queries, keys, and values into multiple smaller sets, and computing the attention weight matrix for each set separately. The results of each set are then concatenated and projected back to the original dimensionality, allowing the model to attend to different aspects of the input sequence in parallel.

The Transformer architecture also includes feed-forward layers, which are basic

types of neural networks, in both the encoder and decoder. The feed-forward layers are important for the Transformer because they allow the model to capture complex relationships between the input and output. The self-attention layers are powerful tools for attending to different parts of the input sequence, but they are limited in their ability to learn complex non-linear transformations. In other words, the feed-forward layers provide a way for the model to learn more complex relationships between the input and output.

The Transformer was the model chosen for this work for several reasons. First, it has parallelization features, which significantly speeds up the training time. Also, it presents better performance than all other seq2seq models on a variety of natural language processing. But, besides the better technical features, the main reason we chose the model is for its self-attention mechanism. To perform the β-reduction over lambda terms, it is necessary to substitute every occurrence of the variable in question with the term. So, we think that the self-attention can be used to "pay attention" to every occurrence of the variable in question on the λ-term when performing the task.

3.6 Hyperparameters

Our model, like most NN models, can be seen as a function that takes a dataset as input and produces a function (prediction function) as the output. In practice, however, our model also has some parameters besides the dataset. We call these parameters Hyperparameters. Bengio (2012) defines a hyperparameter for a learning algorithm A as:

> (...) a variable to be set prior to the actual application of A to the data, one that is not directly selected by the learning algorithm itself. It is basically an outside control knob.

These control knobs are not easy to set, and adjusting them can be a cumbersome task. The process of setting those values requires experience and trial and error, being based more on luck than on science (SMITH, 2018).

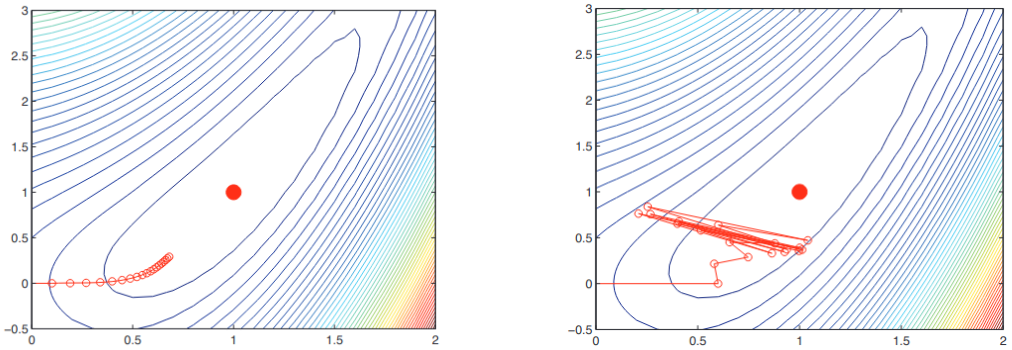
Traditional Neural Networks have some hyperparameters that we can pass, such as the number of layers, the size of each layer, the activation function, the learning rate, etc. The model we use - the Transformer - is more complex, and therefore, has even more hyperparameters, such as the number of attention heads and the number

of layers in the encoder and decoder. Preliminary results showed that using the same hyperparameters that Lample and Charton (2019) used for its symbolic mathematics training is a suitable option for our purposes. Since the goal of this work is not to find the optimal hyperparameters, we settled for this choice. However, we needed to adjust one hyperparameter during the training - the learning rate.

3.6.1 Learning Rate

When the model is learning, at each step, it tries to get closer to the answer. The amount by which it tries to get closer depends on the steepness of the function on that point and on a scaling factor, which we call Learning Rate (LR) (MURPHY, 2012). Choosing this hyperparameter is difficult because there is a trade-off. If we pick an LR too small, the training takes too long. But if we pick an LR too big, the training can start to diverge from the desired output. Figure 3.7 shows a real example of a gradient descent algorithm over a simple function, with two different LRs, one smaller and one bigger. We can see that with the smaller LR, the function converges to the optimal answer. However, with the bigger LR, the function starts to diverge and probably never finds the optimal solution. Thus, in our work, we had to adjust this hyperparameter for each training, so they do not start to diverge.

Figure 3.7: Example of a gradient descent algorithm in a simple function, showing the difference the learning rate can make on the convergence to the optimal answer.



(a) Small LR, converging to the optimal answer. (b) Big LR, diverging from the optimal answer.

Source: Murphy (2012)

4 METHODOLOGY

In this chapter, we describe the methodology we have devised to test and validate our hypothesis and answer the research question we have proposed. This methodology guided the research process, and we believe that by following this methodology, we have conducted a rigorous and replicable research process.

We start by outlining the research question and hypothesis the study seeks to address, providing an explanation of how the hypothesis relates to the tasks we are proposing. Following that, we provide details on the datasets used to achieve the research goals, including details such as the generation of the data and the types and the number of datasets used. Next, we provide information about the specific AI techniques and tools we used in our study, including the machine learning model we chose, the hyperparameters we set, and any other technical details that are relevant. In addition, we discuss the technical aspects of our work, such as the configuration of the machine where the model was trained and the pieces of code that were used. This information is important for reproducibility and for other researchers who may want to build upon our work. Finally, we describe how we present and evaluate the results of our study, including the metrics used to evaluate the model, as well as how we present them.

4.1 Tasks

Our research question is: **Can a Machine Learning model learn to perform computations?** In other words, we want to investigate whether machine learning algorithms can be used to perform computational tasks and whether they are capable of learning to perform these tasks through training. To answer this question, two hypotheses were proposed:

- **H1: The Transformer model can learn to perform a one-step computation on Lambda Calculus.**
- **H2: The Transformer model can learn to perform a full computation on Lambda Calculus.**

For each of these hypotheses, we propose a different task for our model to learn. The hypothesis **H1** is related to the task of performing the One-Step Beta Reduction (OBR). This hypothesis claims that the model is able to perform a single reduction step

in λ-Calculus, taking a λ-term and transforming it according to the β-reduction rules.

The hypothesis **H2** is related to the task of Multi-Step Beta Reduction (MBR). This hypothesis suggests that the model is able to perform multiple reduction steps in lambda calculus, taking a normalizable λ-term, i.e., a λ-term that has a normal form, and transforming it into its normal form through multiple beta reduction steps.

The primary focus of our research question is aligned with the second hypothesis. However, we chose to begin with an easier hypothesis as a starting point. The first task is considered easier because it requires the execution of a single computational step, which is less complex than performing a full computation. This approach enables us to gradually build up our understanding and confidence before moving on to the more challenging second hypothesis.

The Lambda Calculus was chosen as the underlying formalism because it is very simple but still Turing-Complete, i.e., it is powerful enough to express any computation that can be expressed in any other algorithmic system, according to the Church-Turing thesis (SIPSER, 1996). Using modern imperative programming languages, the model would have to learn several abstract concepts to grasp the notion of computation, such as variable assignment, loops, conditional statements, arithmetic operations, etc. In contrast, with λ-Calculus, the model can, instead, focus solely on learning the β-reduction.

To support these hypotheses, we plan to generate several datasets for each of the tasks and use these datasets to train machine learning models. By training the models on these datasets, we aim to determine if the models are able to learn and perform the tasks associated with each hypothesis.

4.2 Lambda Sets and Datasets

Since, to the best of our knowledge, there are no existing references on generating lambda terms in the literature, we need to develop the generation process from scratch. To generate the datasets that the models are going to train on, we first generate *Lambda Sets* (LSs) containing only lambda terms. From these LSs, we generate the datasets needed for the trainings. Thus, we generate three LSs:

- **Random Lambda Set (RLS)**: This LS is generated as random and unbiased as possible, using the techniques from Section 5.2. Thus, this LS can have terms that

do not have a normal form. With the datasets generated from this LS, we want to assert that the model can learn the β-reduction, regardless if the input terms represent meaningful computations or if they have a normal form.

- **Closed Bool Lambda Set (CBLS)**: This LS has its terms representing closed boolean expressions and is generated using the methods from Section 5.3. Thus, all terms in this LS have a normal form. With the datasets generated from this LS, we want to assert that the model can learn the β-reductions from meaningful computations. Also, these datasets are useful to validate the models trained with the datasets from the RLS, i.e., to validate if the model that learned from random terms can perform meaningful computations.
- **Open Bool Lambda Set (OBLS)**: This LS have its terms representing open boolean expression, with free variables in them, and is also generated using the methods from Section 5.3. With the datasets generated from this LS, we want to assert that the model can learn the β-reductions from meaningful computations, even if the terms have free variables. These datasets are also useful to validate the models trained with other datasets.

For the One-Step Beta Reduction task, we generate datasets based on the three LSs proposed. However, for the Multi-Step Beta Reduction task, we do not utilize the Random LS to generate datasets, as the terms produced randomly may not have a normal form and result in an infinite loop during the multi-step β-reduction. The other two LSs are guaranteed to have terms that have a normal form since they are built using only boolean expressions.

In addition to the LS mentioned above, for each task, we create an extra LS, which we refer to as a "Mixed Lambda Set" (MLS). These LSs are a combination of the other LSs used on each task. Thus, for the OBR, the MLS is composed of terms coming from the RLS, CBLS, and OBLS, in the same proportion. For the MBR, the MLS is formed by terms coming from the CBLS and OBLS, in the same proportion. With these LSs we want to assert that the model can learn from a domain that contemplates several kinds of terms.

Since we are interest in comparing the training with different notations, instead of generating only one dataset from each lambda set, we generate three with different variable naming conventions:

- **De Bruijn**: This convention uses the prefix de Bruijn notation, presented in Section 2.10. We utilize this convention because it uses a shorter notation and

presents a way of representing λ-terms without the need of naming the variables, which can be beneficial for our model.

- **Traditional**: This convention uses the prefix traditional notation, presented in Section 2.3 To achieve this version, we take the de Bruijn version and give names to the variables, using the algorithm from Section 2.10.1. The order for variable naming used for this convention is the alphabetical order. We utilize this convention because we want to compare the results of the learning process using the de Bruijn notation with the traditional notation.
- **Random Vars**: This convention also uses the prefix traditional notation. However, for this version, we take the de Bruijn version and give names to the variables using the same algorithm as above, but now using a random order for the variables. We utilize this notation because we want to check if the way we name the variables matter for the performance of the models.

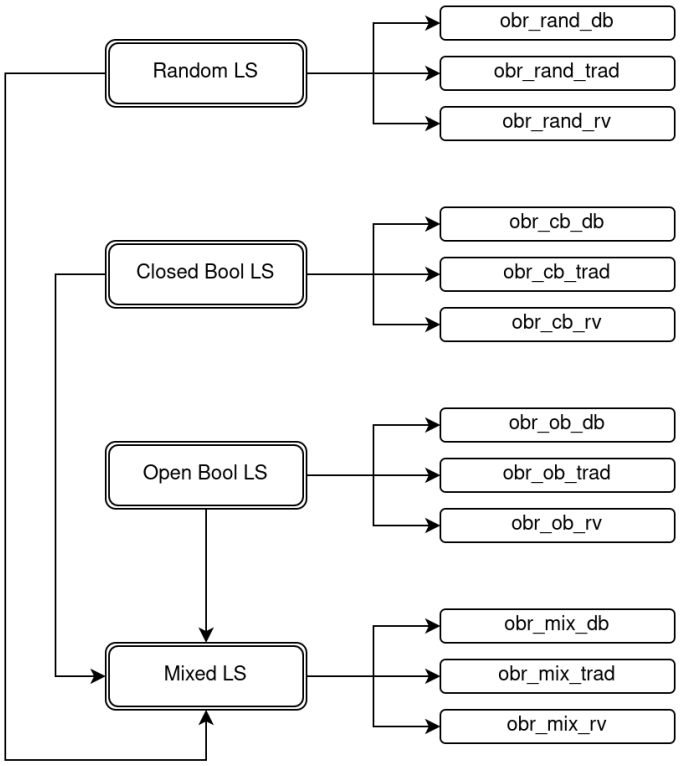
Therefore, for the OBR task, we ultimately have a total of 12 datasets, as illustrated in Figure 4.1. Regarding the MBR task, we have a total of 9 datasets in the end, as depicted in Figure 4.2 [a]. It is noteworthy that the LS utilized for both tasks are identical, meaning that the initial λ-terms for the datasets that employ the same LS are consistent across both tasks. These 21 datasets provide us with a broad set of test cases to evaluate the performance of our models. Chapter 5 explains thoroughly how the lambda sets and the datasets are generated.

Each dataset contains around one million examples, and we further divide each dataset into train, validation, and test sets. Adhering to the methodology outlined by Lample and Charton (2019), we allocate a total of about 10 thousand examples for both the validation and test sets. This division of the datasets into train, validation, and test sets allows us to effectively train our models and evaluate their performance on independent data. By utilizing a large number of examples in each dataset and following established best practices, we want to ensure that our results are robust and representative of the underlying task.

After generating the datasets, we clean them, deleting pairs that: (i) have undergone α-reductions, since we are following the Barendregt Convention; (ii) the first element is already in normal form because we want all pairs to represent β-reductions; (iii) appears repeated on the dataset.

[a] All the 21 datasets generated are available at <https://bit.ly/lambda_datasets>.

Figure 4.1: Scheme of how all the datasets for the OBR tasks are generated. It starts with the three Lambda Sets (RLS, CBLS, and OBLS), and ends with all 12 datasets that are available for the OBR task.



Source: The authors.

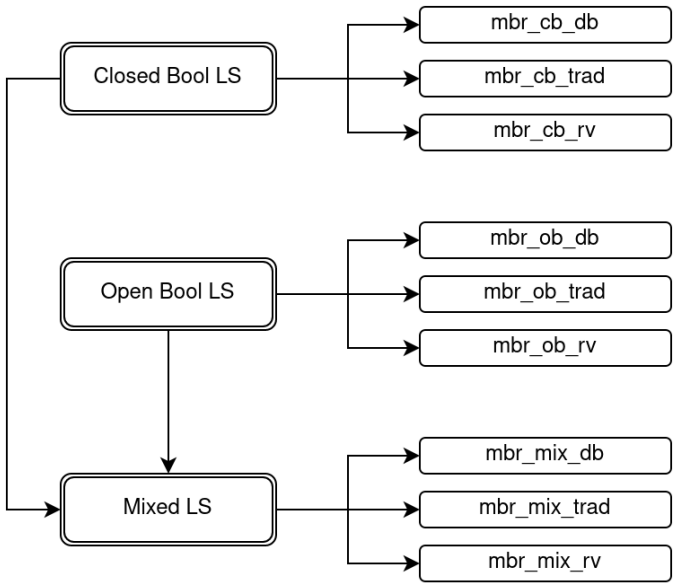
4.3 Training

To learn the tasks mentioned before, we use a neural model. Since the λ-terms we are using do not have a fixed size, we need our model to accept inputs of varying lengths and generate outputs accordingly. To achieve this, we use a sequence-to-sequence model (seq2seq), which allows for inputs and outputs of different sizes. Specifically, we use the Transformer model, proposed by Vaswani et al. (2017). This model has been widely used for different applications, including symbolic ones as demonstrated by Lample and Charton (2019).

For the hyperparameters, preliminary tests showed us that the parameters used by Lample and Charton (2019) were acceptable for our tasks. If needed, they can be adjusted during the training process. So, the initial hyperparameters are the following:

- Number of encoding layers - 6
- Number of decoding layers - 6

Figure 4.2: Scheme of how all the datasets for the MBR tasks are generated. It starts with the two Lambda Sets (CBLS and OBLS), and ends with all 9 datasets that are available for the MBR task.



Source: The authors.

- Embedding layer dimension - 1024
- Number of attention heads - 8
- Optimizer - Adam (KINGMA; BA, 2014)
- Learning rate - 1×10^{-4}
- Loss function - Cross Entropy

4.3.1 Configurations

The experiments were conducted on a server located in our Machine Learning laboratory, situated at the Informatics Institute - UFRGS. The server has the following configurations:

- CPU: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
- RAM: 32 GB (2 x 16 Gb) DDR4 @ 2667 MHz
- GPU: Quadro P6000 with 24 Gb
- OS: Ubuntu 18.04.5 LTS

Our initial aim was to have each training session run for a duration of 12 to 24

hours. Preliminary results showed that each training consisting of 50 epochs with an epoch size of 50000 would take between 12h to 30h to complete. So, we chose this arrangement. This configuration allows the model to process a total of 2.5×10^6 equations, which is 2.5 times the size of the dataset.

With this machine, model, and configuration we are using, we can safely have inputs with up to 250 tokens. With more than that, we end up with memory shortage.

4.4 Code and Implementation

In this work, we utilized two distinct pieces of code. The first piece, adapted from Lample and Charton (2019) [b], contributed for two main purposes: (i) it assisted in the generation of the lambda sets described above, by generating intermediate λ-terms and (ii) it handled the learning process. The second piece of code is our own implementation, which deals with the specifics of the λ-Calculus [c]. Our implementation, coded in *Python*, is a simulator of the Lambda Calculus, supporting both traditional notation and De Bruijn notation, as well as both notations in prefix and suffix manner. It is entirely based on the substitution operation, which is the fundamental operation of Lambda Calculus.

The lambda calculus simulator is used to generate the λ-terms for the lambda sets from the intermediate terms generated by the code adapted from Lample and Charton (2019). Also, it is used to generate the datasets from the lambda sets, generating the respective reductions from the terms in the LSs. It achieves all this by doing the lambda calculus specifics, such as parsing terms, computing reductions, converting terms between the different notations, etc. This code can be used by anyone that needs a Lambda Calculus parser and/or simulator. It also computes metrics for the datasets and has drawing functions to generate images of the terms as trees.

4.5 Experiments and Results

Our idea was to use the cross-validation technique to assess the statistical significance of our results. However, given that each training took approximately 1 day, and we had 21 datasets to train, it was not possible to apply the technique. So, we use just

[b] This code is available at <https://github.com/jmflach/SymbolicLambda>
[c] This code is available at <https://github.com/jmflach/lambda-calculus>

one cross-validation iteration for each training.

To evaluate the performance of the model during the training, the accuracy of the model to predict the data on the test dataset is calculated and recorded after each epoch. This accuracy metric is a measure of how well the model is able to make correct predictions on the data it has seen during training. For each of the models trained, we display a graph showing the evolution of the accuracy of the model (y-axis) over the epochs (x-axis).

The model's accuracy determines whether the predicted string matches the expected output. However, accuracy may not be the only relevant metric for evaluating the performance of a model in text generation or other similar tasks. In some cases, it may be useful to measure the similarity between the predicted string and the expected one, even if they are not identical. So, additionally, we used a string similarity metric to compare how close the predicted string is to the expected one. For this, we used a common string similarity metric, the Levenshtein distance, which measures the number of changes (insertions, deletions, or substitutions) needed to transform one string into the other (LEVENSHTEIN et al., 1966). This metric actually provides the absolute difference between the two strings, so we divide this distance by the maximum distance possible between the two strings (which is the length of the longer string) to generate a percentage of dissimilarity. Then, we just subtract this value from 1 to get a percentage of similarity between the two strings. So, for each trained model, we also provide a string similarity value. The formula used is as follows:

$$str\_sim(s1, s2) = 1 - \frac{lev\_dist(s1, s2)}{max(len(s1), len(s2))}$$

As part of our analysis, we also assess the capacity of the models trained with each dataset to evaluate the other datasets. We achieve this by performing additional evaluations with each of the already trained models. For this, we use a model trained with one dataset to evaluate the other datasets that use the same notation and are designed for the same task. For example, we take the model that trained on the *obr_rand_trad* dataset and evaluate how it performs on the *obr_cb_trad*, *obr_ob_trad* and *obr_mix_trad* datasets.

By evaluating a model on the other datasets, which it did not train on, we can better understand the model's strengths and weaknesses, as well as its ability to generalize to new data. If the model performs well on other datasets of the same type, it

may be a good sign that the model has learned meaningful patterns in the data and can be applied to new, unseen data. If the model performs poorly on other datasets, it may indicate that the model has, for instance, overfit to the original training data or the data was not adequate.

5 GENERATING LAMBDA DATASETS

In Chapter 3 we made clear that the learning model we are using in this work is a type of supervised learning. Thus, data is a key component in the learning process. The quality and relevance of the data that is provided to the model determine its ability to solve the proposed tasks and make accurate predictions effectively. This happens because the model uses the data in the training set to form a general understanding of the relationship between input and output and to extrapolate that understanding to new examples. In this work, we are interested in two main tasks: the task of learning how to perform a single beta reduction given a lambda term, which we call One-Step Beta Reduction (OBR), and the task of learning how to reach the normal form of lambda terms by performing a sequence of one step beta reductions, which we call Multi-Step Beta Reduction (MBR). When the distinction is not needed, we use the word "reduction" to refer to either of the tasks.

The examples on the datasets represent lambda terms and their respective reductions. Given a literature review, there are no datasets available for the tasks we are interested in. Also, there are no datasets composed solely of lambda terms. So, in this chapter, we discuss different methods on how we can generate the datasets we are going to be used to achieve our goals. To achieve this, first, we explain how to generate sets of random lambda terms, that we refer to as Lambda Sets (LSs), which we can later use to generate the datasets for the specific tasks.

We start explaining how to generate the terms completely randomly, inserting as little bias as possible. Then, we discuss some issues with the data generated by this technique and show how to avoid those issues, limiting the scope for the terms generated. Finally, we explain how to use the LSs generated to produce the final datasets for the desired tasks.

Having the lambda terms dataset, applying the reductions is straightforward. So, generating the LSs is the most challenging step of the dataset generation. To accomplish this, we use an existing algorithm to generate intermediate term trees and then transform those trees into lambda terms.

Also, as mentioned before, we want to create versions of the datasets in De Bruijn (DB) notation to compare the difference in the ability of the models to learn both representations.

5.1 Format

The lambda terms on the datasets are represented as lambda terms in prefix notation. So, each dataset consists of a simple text file with each line representing an example of a reduction, with the following style:

BETA <M> <N>

<M> and <N> are lambda terms in prefix notation. For the OBR task, <N> is the β-reduction of <M>, using the lazy evaluation strategy. For the MBR task, <N> is the normal form of <M>. The word BETA and <M> are separated by a blank space. The terms <M> and <N> are separated by a `tab` character.

For the strings that represent the lambda terms, we use the uppercase letter "L" to represent the λ symbol and "@" for the application symbol. This convention is important only for making the terms more human-readable because, in the learning stage, the model translates all tokens into numbers. Below, there are some examples of how the dataset should look at the end of the generation for both the OBR and MBR tasks, in infix notation to be more human-readable, and prefix notation, which is exactly how it appears on the training datasets. It is important to note that the final datasets that are used in this work contain terms that are larger in size than the examples shown here. These examples are provided only as a demonstration and are not representative of the complexity and size of the actual terms that are present in the final datasets.

OBR task with infix notation:

BETA $(\lambda b.\ ((\lambda p.\ \lambda a.\ \lambda b.\ (p\ b\ a))\ (\lambda a.\ \lambda b.\ b)))\ (\lambda a.\ \lambda b.\ a)$ $(\lambda p.\ \lambda a.\ \lambda b.\ (p\ b\ a))\ (\lambda a.\ \lambda b.\ b)$
BETA $(\lambda p.\ \lambda a.\ \lambda b.\ (p\ b\ a))\ (\lambda a.\ \lambda b.\ b)$ $\lambda a.\ \lambda b.\ ((\lambda a.\ \lambda b.\ b)\ b\ a)$
BETA $\lambda a.\ \lambda b.\ ((\lambda a.\ \lambda b.\ b)\ b\ a)$ $\lambda a.\ \lambda b.\ ((\lambda b.\ b)\ a)$
BETA $\lambda a.\ \lambda b.\ ((\lambda b.\ b)\ a)$ $\lambda a.\ \lambda b.\ a$

OBR task with prefix notation, with the same terms as the infix notation example:

BETA @ L b @ L p L a L b @ @ p b a L a L b b L a L b a @ L p L a L b @ @ p b a L a L b b
BETA @ L p L a L b @ @ p b a L a L b b L a L b @ @ L a L b b b a
BETA L a L b @ @ L a L b b b a L a L b @ L b b a
BETA L a L b @ L b b a L a L b a

NBR task with infix notation:

BETA $(\lambda b.\ ((\lambda p.\ \lambda a.\ \lambda b.\ (p\ b\ a))\ (\lambda a.\ \lambda b.\ b)))\ (\lambda a.\ \lambda b.\ a)$ $\lambda a.\ \lambda b.\ a$
BETA $(\lambda p.\ \lambda a.\ \lambda b.\ (p\ b\ a))\ (\lambda a.\ \lambda b.\ b)$ $\lambda a.\ \lambda b.\ a$
BETA $\lambda a.\ \lambda b.\ ((\lambda a.\ \lambda b.\ b)\ b\ a)$ $\lambda a.\ \lambda b.\ a$
BETA $\lambda a.\ \lambda b.\ ((\lambda b.\ b)\ a)$ $\lambda a.\ \lambda b.\ a$

NBR task with prefix notation, with the same terms as the infix notation example:

```
BETA @ L b @ L p L a L b @ @ p b a L a L b b L a L b a     L a L b a
BETA @ L p L a L b @ @ p b a L a L b b     L a L b a
BETA L a L b @ @ L a L b b b a     L a L b a
BETA L a L b @ L b b a     L a L b a
```

5.2 Generating Lambda Sets

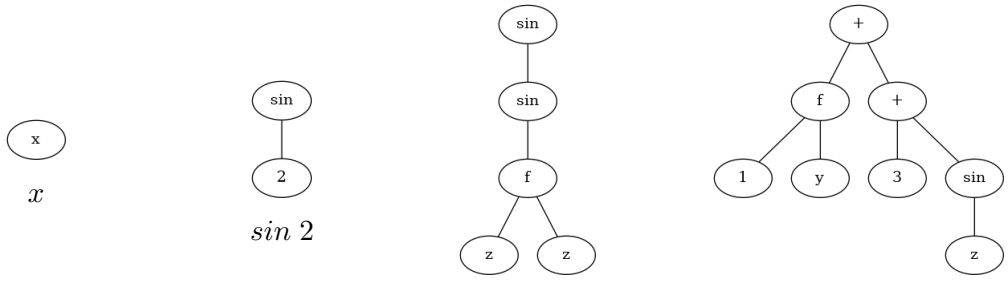
Our goal in this section is to produce a set of randomly generated lambda terms as trees. Since, to the best of our knowledge, there are no references for the generation of lambda terms in the literature, we have to formulate the generation from the beginning. Also, the task of generating uniformly random trees is not a trivial task (LAMPLE; CHARTON, 2019). So, we adapt an existing algorithm that generates random mathematical expression trees. With this algorithm, we first sample a set of intermediate trees, that represents the structure of the lambda term trees. Then, we define a pipeline for dealing with the lambda calculus specifics of the generation, taking these intermediate trees and generating the lambda terms.

5.2.1 Generating Intermediate Trees

We start by generating random intermediate trees that represent the shape of the final lambda trees. However, the task of generating uniformly random trees is not a trivial task. The naive approach tends to generate deeper trees over broader ones (LAMPLE; CHARTON, 2019). To generate truly random trees, we use the algorithm proposed by Lample and Charton (2019) to generate intermediate trees.

In this algorithm, the internal nodes of the trees are operators, and the leaves are variables or numbers. The sets of possible operators and leaves are parameters of the algorithm. The algorithm then generates random trees representing mathematical expressions using those arguments. Besides that, one thing that is important is the arity of each possible operation, because it represents the number of children that the node has. Figure 5.1 shows some expression trees the algorithm could generate with $[(add, 2), (sin, 1), (f, 2)]$ as possible internal nodes and $[1, 2, 3, x, y, z]$ as possible leaves.

Figure 5.1: Examples of mathematical expression trees, as well as their string representations, generated by the algorithm used to generate intermediate trees



x $sin\ 2$ $sin\ sin\ f\ z\ z$ $add\ f\ 1\ y\ add\ 3\ sin\ z$

Source: The authors.

Figure 5.2: Difference between implicitly putting the variable of the abstraction in the node, and explicitly putting the variable as a leaf, as implemented in this work.

L v M L v M

Source: The authors.

The problem is that this algorithm was made to generate trees that represent mathematical expressions. To use this algorithm for our purposes, we need to adapt it from generating mathematical expressions to generating lambda terms. For this, we have to determine the set of possible internal nodes and leaves to provide to the algorithm.

In Section 2.3.1, we have demonstrated how lambda terms can be represented as trees. Table 2.1 shows that the internal nodes of a term are the lambda abstractions and applications. The leaves, on the other hand, are the lambda variables, and they can be names, such as *x, y, z*, or numbers if we are using the de Bruijn Notation.

So, it seems straightforward to use two "operators": abstraction (*abs*) and application (*app*). But we also have to provide the arity of those operations. For the *app* it is easy since we can see it as an operator with two "arguments", so, its arity would be 2. But for the *abs*, it is not so simple. When the lambda trees were presented, the variable name of the abstraction was implicitly put in the abstraction node. But in the string we want to generate, each token is a node by itself, including the variable of the abstraction. Figure 5.2 shows how the lambda tree is actually built.

So, if we look at the abstraction as an operator, we have to decide its arity. It

has two “arguments”: a variable name - the binding of the abstraction - and a body. But they are not both arbitrary lambda terms, as it is in the application. So, the arity is not 2, because it can not have any type of lambda term as the first argument. But it is also not 1, because it needs 2 arguments. So, the arity kind of sits between 1 and 2.

To overcome this, we actually generate the terms in de Bruijn notation. With this notation, we do not need to specify the name of the variable binding, and then we can simply say that the arity for the lambda abstraction is 1.

For the leaves, that represent the lambda variables, generating the terms using the DB representation also gives an advantage. We want to separate the generation of the “shape” of the trees from the lambda term specifics. So, we simply generate the number 1 for every index at this stage, as a placeholder for the final DB indexes. This also allows us to generate more than one tree for every tree generated in this step because we can sample as many trees as we want from one intermediate tree. With this, we are able to use their algorithm as a black box, without any modification. But the original algorithm from Lample and Charton (2019) also receives a sample probability for every operation in the set of the operations desired. For example, if we pass 0.7 for the app and 0.3 for the abs, the chance of a node being an app is 70% and 30% for the abs. The problem with this is that all trees generated have a similar proportion of applications and abstractions. But there are terms that have way more abstractions than applications and vice-versa. We want a dataset that has terms with very distinct forms. So, since we wanted the proportion of apps and abs to be variable, we actually modified their algorithm in such a way that it randomly change these probabilities between a range of values.

Another difference is that their algorithm not only generates the expressions but already generates pairs of mathematical expressions for the final dataset - for instance, a function and its integral. But since the lambda terms are not yet completed we do not do this here. We perform this step later, once the terms have been completed. For now, we are only generating pairs with the following format:

```
BETA <intermediate_tree>   null
```

With this, we have a set of randomly generated intermediate trees that we can use to sample lambda terms.

5.2.2 Finishing the Trees

The terms generated in the previous section are in de Bruijn notation and with all variable indexes set to one. Now, our algorithm goes through every tree generated and makes the lambda specifics adjustments. First, we set the DB indexes to their final values. Once this step is completed, we obtain a set of λ-terms in de Bruijn notation, which already is a *Lambda Set*. But since we are interested in generating datasets in de Bruijn, traditional and random vars conventions, in the next step we explain how we convert this DB LS to the other two conventions, transforming the terms from the DB notation to the traditional notation, i.e., give names to the lambda variables, using two different naming techniques.

To execute the first step, our idea is to first compute the maximum value that the index can assume to be binded by some abstraction, i.e., to not represent a free variable. After, we generate a random number between 1 and the maximum index computed, following a distribution. To generate terms that have free variables, we can change the range for this generation. As mentioned before, it is possible to apply this step several times for the same intermediate tree to generate different λ-terms.

For the second step, we need to convert from the De Bruijn notation to the traditional notation, with names instead of numeric indexes to represent variables. This step is not trivial because there is no one-to-one correspondence between terms in DB notation and terms in traditional notation. One DB term can have an infinite number of counterparts in traditional notation, all alpha-equivalents. To achieve this, we use the algorithm proposed in Section 2.10.1. As seen in that Section, we need to define the alphabet and the order for the variable naming. For the alphabet, we use the English alphabet. For the order, we use two different approaches: (i) the alphabetical order, and (ii) the random order. With the alphabetical order, we want to make the terms more predictable to the model. With the random order, we want to reduce the predictability of the terms, training a model that is more agnostic to the naming of the variables.

5.2.3 Problems with this Generation

The dataset generated with this approach has some problems. First, the majority of the terms generated by this approach are not meaningful, i.e., they do not represent anything based on the encodings that we use. Although this can be seen as a major

problem, we still use this dataset in our training because our goal is to learn the β-reduction, regardless of the meaning of the terms.

So, for instance, the chance of this algorithm generating the number 10 - $\lambda f.\lambda x.(f(f(f(f(f(f(f(f(f(fx)))))))))$ - , a term with 10 abstractions in a row, is virtually zero. it is unrealistic to expect the model to learn how to handle numbers if the dataset probably does not contain numbers.

Also, since these terms do not represent any computations that have a practical or meaningful application, they may not have a normal form, i.e., they may enter a loop when performing a sequence of β-reductions.

5.3 Generating Lambda Terms with Reduced Scope

As seen in the previous section, when we try to generate lambda terms sampling random trees directly from lambda terms nodes, we end up with terms that are, in fact, lambda terms, but probably do not correspond to known encodings [a], such as those presented in Section 2.8.

A solution for this issue would be to generate terms that actually represent meaningful computations, such as the sum of two numbers, or a binary search tree converter. To achieve this, we could change the level of abstraction we are generating the expressions. Instead of directly generating lambda trees, we could, for instance, generate arithmetic expression trees and later, using known lambda encodings and algorithms for numbers and arithmetic operations, translate the generated expressions to lambda terms. This way the generation would still be random, but we end up with terms that represent something on our encoding. Of course, by doing this, we are limiting the scope of the computations our dataset is representing, but since one can argue that our last dataset is not representing significant computations, this could be an improvement.

The newly introduced method has the capability to produce terms for any domain of interest. However, for this study, we must designate a particular domain in order to create the data set and train the model. One common domain that researchers choose is arithmetic (ZAREMBA; SUTSKEVER, 2014; KAISER; SUTSKEVER, 2015; TRASK et al., 2018). But arithmetic in lambda calculus has two main problems for this approach. First, since the definition of numbers is basically a unary definition, the terms are lengthy. So,

[a] One can argue that, since we can create any encoding we want, for any term, there exists an encoding that that term has a meaning

Table 5.1: Table showing the lambda encoding for the boolean values true and false, as well as some algorithms for common boolean operators.

Boolean	**λ-term**	**DB prefix λ-term**
true	$\lambda a.\lambda b.a$	L L 2
false	$\lambda a.\lambda b.b$	L L 1
and	$\lambda p.\lambda q.p\ q\ p$	L L @ @ 2 1 2
or	$\lambda p.\lambda q.p\ p\ q$	L L @ @ 2 2 1
not	$\lambda p.\lambda a.\lambda b.p\ b\ a$	L L L @ @ 3 1 2

Source: The authors.

performing operations with big operands would generate terms that our configuration would not support. Second, some algorithms are counter-intuitively complex, such as the subtraction using the Church encoding (BARENDREGT, 1997).

Thus, we move to another domain: boolean arithmetic. In this regard, the lambda calculus is an excellent tool. The terms are short and the encodings are very simple. Table 5.1 shows the encoding for the values (True and False), as well as some algorithms for common boolean operations.

Looking at the table, we can see that the traditional boolean operators have a simple algorithm. For our work, we are interested in the operators *and*, *or*, and *not* since they are enough to produce any other boolean operator (HARRIS; HARRIS, 2010). For the leaves, we can choose if we want to generate closed or open boolean expressions, i.e., if we want the expressions to have only concrete values (true and false), or variables too.

So, to generate this type of dataset, we use the algorithm we used in section 5.2.1. Instead of the lambda-specific nodes, we can use nodes of the expressions we want to generate. In our case, we use $[and, or, not]$ as the internal nodes and $[true, false]$ - or variables for open expressions - as leaves.

After generating the boolean expressions, we can translate them to Lambda Calculus, using the encoding/algorithms presented in table 5.1. To follow the same approach as the previous datasets, we first generate the DB version of it and then use the same steps of the previous section to generate the dataset in the traditional notation. This dataset also has the advantage of having terms that behave more predictably since all of them have a normal form.

5.4 Generating the Final Datasets

Now that we know how to generate *Lambda Sets*, both randomly and domain-specific, we can use them to generate the datasets the network takes as input. This generation depends on the tasks we want to perform, which are two: One-Step Beta Reduction and Multi-Step Beta Reduction. They are similar but have some differences, and we are going to see how to generate them in this section.

5.4.1 One-Step Beta Reduction

For this task, we generate pairs of the type:

BETA <M> <N>

where <M> and <N> are lambda terms and <N> is the One-Step Beta Reduction of <M>. In Sections 2.7 and 2.9, we saw that we can apply the beta reduction on a term iteratively until it reaches its normal form (or enters a loop). So, for example, the term $and\ true\ false$ in lambda generates the following reductions:

$$
\begin{aligned}
& @\ @\ \lambda\ p\ \lambda\ q\ @\ @\ p\ q\ p\ \lambda\ a\ \lambda\ b\ a\ \lambda\ a\ \lambda\ b\ b \\
\rightarrow_\beta \quad & @\ \lambda\ q\ @\ @\ \lambda\ a\ \lambda\ b\ a\ q\ \lambda\ a\ \lambda\ b\ a\ \lambda\ a\ \lambda\ b\ b \\
\rightarrow_\beta \quad & @\ @\ \lambda\ a\ \lambda\ b\ a\ \lambda\ a\ \lambda\ b\ b\ \lambda\ a\ \lambda\ b\ a \\
\rightarrow_\beta \quad & @\ \lambda\ b\ \lambda\ a\ \lambda\ b\ b\ \lambda\ a\ \lambda\ b\ a \\
\rightarrow_\beta \quad & \lambda\ a\ \lambda\ b\ b
\end{aligned}
$$

So, instead of just computing one beta reduction for each term we have and generating only one pair to the final dataset, we can generate several pairs, one for each reduction we were able to compute. The previous example can generate the following pairs:

```
BETA @ @ L p L q @ @ p q p L a L b a L a L b b     @ L q @ @ L a L b a q L a L b a L a L b b
BETA @ L q @ @ L a L b a q L a L b a L a L b b     @ @ L a L b a L a L b b L a L b a
BETA @ @ L a L b a L a L b b L a L b a     @ L b L a L b b L a L b a
BETA @ L b L a L b b L a L b a     L a L b b
```

We just have to be careful because, as the terms generated randomly can go into a loop, we must define a limit on the maximum number of reductions we apply in a term. Also, the beta-reductions of a term can produce a bigger term than the original one, so we have to remove the pairs that contain terms that exceed the limit our configuration has.

So, our main algorithm here is: given a set of terms, we go through every term and perform the beta reduction with the lazy strategy until the term reaches its normal form or the number of reductions reaches the limit defined. For every reduction made, we generate a pair on the dataset. Since we want the terms on the datasets to represent beta-reductions, if a term does not have a redex, i.e., it is already in a normal form, we leave it out of the dataset. Also, sometimes a term has to undergo an alpha-reduction to achieve its beta-reduction. We leave the corresponding pair out of the dataset. This does not reduce the power of the model, because, as mentioned in Section 2.6, we follow the Barendregt convention and so we can assume we do not need to perform alpha reductions. For simplicity, we just remove those terms from the dataset. Finally, we also remove from the dataset pairs that appear more than once on the dataset.

5.4.2 Multi-Step Beta Reduction

For this task, we generate pairs of the type:

BETA <M> <N>

where <M> and <N> are lambda terms and <N> is the normal form of <M>. To achieve the normal form of a term, we must iteratively compute the beta reduction on the term until it does not have any possible reduction. So, we must go through every term and compute its beta reduction until it reaches its normal form. We already did it in the previous section. However, since the term achieving the normal form is crucial for this task, it is not desirable to use a set of terms generated using the random method from Section 5.2. The reason for this is that since the terms are completely random, they are unpredictable and may not have a normal form, i.e., they may get stuck in a loop when reductions are applied.

Using the example 5.4.1 of the previous subsection, we can simply take the first and last element from the list of reductions and make a pair to the dataset:

BETA @ @ L p L q @ @ p q p L a L b a L a L b b L a L b b

So, for every term, we generate a pair on the dataset. But we can see that every term on the list generated has the same normal form. So, another possibility is to take advantage of this fact and, for every term in the list of reductions, generate a pair with

the last term for the dataset. For the example 5.4.1:

```
BETA @ @ L p L q @ @ p q p L a L b a L a L b b     L a L b b
BETA @ L q @ @ L a L b a q L a L b a L a L b b     L a L b b
BETA @ @ L a L b a L a L b b L a L b a     L a L b b
BETA @ L b L a L b b L a L b a     L a L b b
```

The first approach generates a dataset in which every pair represents a boolean expression and its result after evaluation. The second loses this semantics since the intermediate terms no longer hold the meaning of being a boolean expression written in lambda calculus. But it has the advantage of being more agnostic to the semantics of the terms on the dataset.

5.5 Term Sizes

As mentioned in Section 4.3.1, the maximum number of tokens that our configuration allows is 250. So, our generation must respect this limit. For the random dataset, establishing this limit was easy, since there is a parameter in Lample and Charton (2019) algorithm called *max_len* that allows us to generate only terms that have a size below this number. However, for the boolean datasets, it was not so simple, since we must take the generated boolean term and convert it to a lambda term. So, we can not set the *max_len* for this generation. Also, when performing beta reductions, the terms can grow bigger before they start to reduce their size. After conducting several tests, we determined that the parameter for the maximum number of internal nodes to provide to the algorithm should be set to 5. With this configuration, we were able to generate the terms sizes listed in Table 5.2. While we calculated these sizes using the traditional convention datasets, we expect similar results for the other datasets (with the exception of the de Bruijn convention, which should produce smaller term sizes).

5.6 Number of Reductions

During this chapter, it was discussed that for certain Lambda Sets, we iteratively generate the reductions for each term until it reaches its normal form. Thus, another important metric we consider is the number of reductions that each term had to undergo to reach its normal form. This number can be seen as how many computational steps are

Table 5.2: Table showing the minimum, maximum, and average sizes of the input λ-terms for each dataset. The datasets considered were the ones that use the traditional convention.

Task	Dataset	min	max	avg
OBR	random	5	249	127.2 ± 64.99
	closed bool	9	193	97.6 ± 26.76
	open bool	5	181	66.46 ± 21.73
	mixed	5	249	86.93 ± 46.56
MBR	closed bool	9	193	97.55 ± 26.75
	open bool	5	181	66.46 ± 21.72
	mixed	5	181	77.96 ± 28.02

Source: The authors.

necessary for evaluating a given term. Table 5.3 shows the average number of reductions that the terms of each Lambda Set have undergone to generate their respective datasets - for both the OBR and MBR tasks.

Table 5.3: Table showing the minimum, maximum, and average number of reductions generated by each Lambda Set. The mixed dataset considered here is the one with terms coming only from the closed bool and open bool Lambda Sets.

Lambda Set	min	max	avg
closed bool	3	100	18.8 ± 12.22
open bool	1	100	18.88 ± 10.42
mixed	2	100	18.82 ± 11.32

Source: The authors.

6 RESULTS

This chapter presents the results obtained from the implementation of the proposed AI model, the Transformer, to solve the tasks of One-Step Beta Reduction (OBR) and Multi-Step Beta Reduction (MBR), over the datasets generated. Here, we provide a detailed account of the findings from the experiments conducted and to show the effectiveness of the algorithm in solving the targeted problem. We aim to present the findings in a clear and organized manner, to facilitate understanding and interpretation. The results include relevant tables and graphs to support the claims made in the study.

We first present the results from the training, for both the One-Step Beta Reduction and Multi-Step Beta Reduction tasks. Next, we present the results from the evaluations across the datasets.

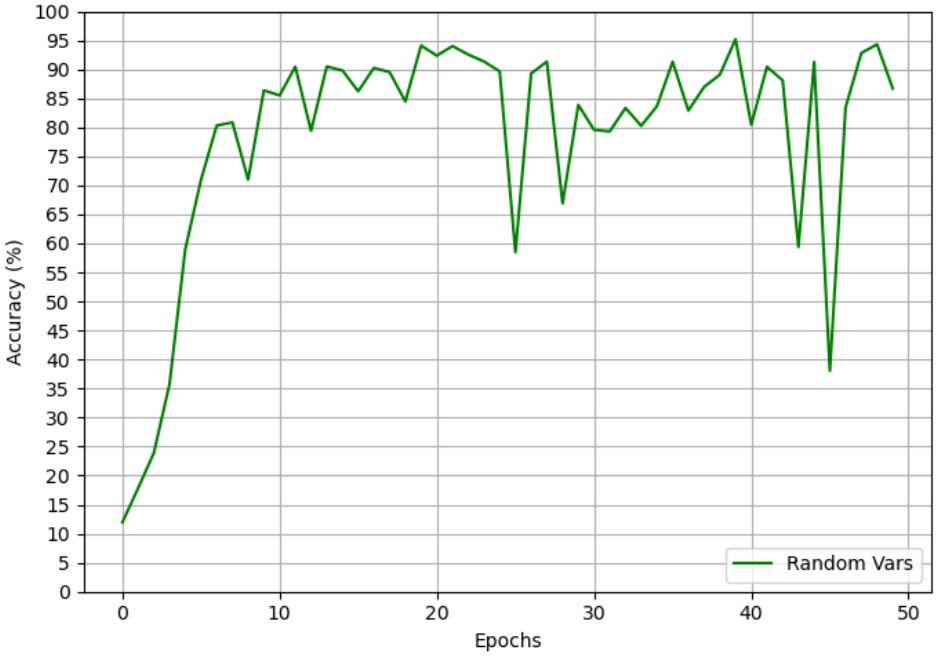
6.1 Training Results

This section showcases the results of the training for both the OBR and MBR tasks. Initially, we highlight an issue encountered during the trainings, concerning one of the hyperparameters - the learning rate. Then, we display the training graphs for each of the tasks and each of the evaluated datasets. After, we present a table showing the final accuracies for all models, as well as the string similarity metric.

6.1.1 Learning Rate

Some training experiments displayed oscillation in the accuracy, indicating that the initial learning rate (1×10^{-4}) was too high. Figure 6.1 shows an example of training where the accuracy oscillates at unacceptable rates. So, we had to adjust the learning rate for these trainings. We initially used the same value for all trainings, but decreased it based on the degree of oscillation. Table 6.1 shows the final learning rates for each training executed. Although the learning rate was adjusted for different trainings, we kept it consistent for the three conventions in each lambda set, for comparison purposes. It is important to note that we did not perform a thorough search for the optimal learning rate, instead selecting a parameter that resulted in a satisfactory and converging accuracy.

Figure 6.1: Graph displaying the progression for the training of the One-Step Beta Reduction task, for the closed bool dataset, with the random vars convention, with a learning rate of 1×10^{-4}. After epoch 20, the accuracy started to oscillate at unacceptable rates.



Source: The authors.

Table 6.1: Values for the learning rate hyperparameter chosen for each of the tasks and lambda sets trained. The value started with 1×10^{-4} and it was lowered as the trained showed an unacceptable oscillation, indicating the learning would not converge.

Task	Lambda Set	Learning Rate
One-Step Beta Reduction	random	1×10^{-4}
	closed bool	6×10^{-5}
	open bool	8×10^{-5}
	mixed	1×10^{-4}
Multi-Step Beta Reduction	closed bool	3×10^{-5}
	open bool	5×10^{-5}
	mixed	5×10^{-5}

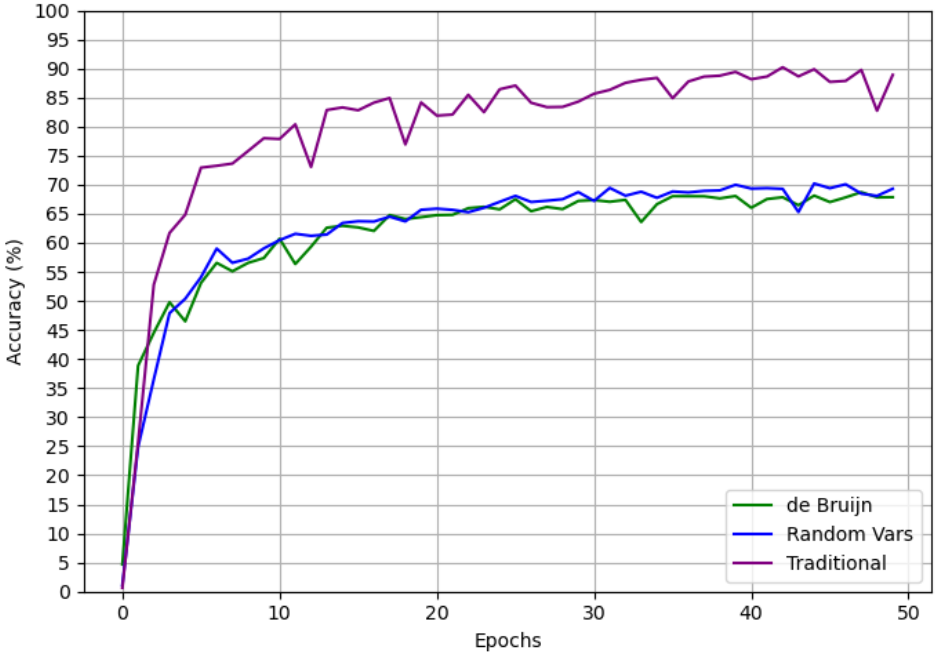
Source: The authors.

6.1.2 Results

This section presents graphs and tables that illustrate the training outcomes for every model trained. The graphs display the model's accuracy for the test dataset of the respective training dataset as it evolves over the training epochs. Each graph showcases the results for all three conventions utilized in this study: the traditional convention, the random vars convention, and the de Bruijn convention. In an effort to avoid an excessive amount of information on the graphs, we chose to display only the model's accuracy and omit the model's loss, given that each graph already represents three results. It would also be repetitive, considering the inverse correlation between the loss and the accuracy seen in the training outcomes.

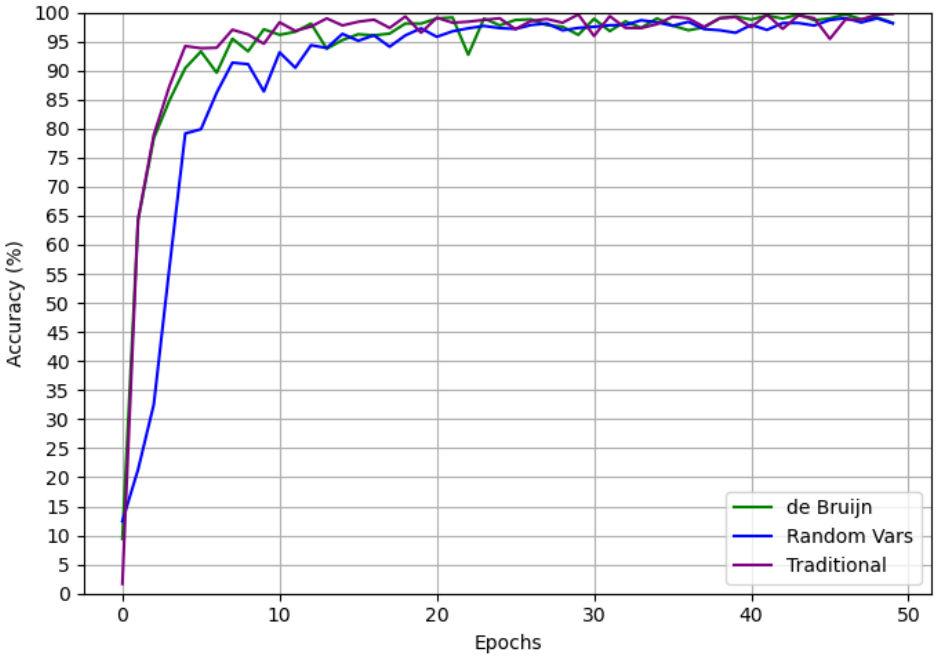
For the OBR task, the training for the random datasets can be seen in Figure 6.2, the closed bool in Figure 6.3, the open bool in Figure 6.4, and the mixed in Figure 6.5. For the MBR task, the training for the closed bool datasets can be seen in Figure 6.6, the open bool in Figure 6.7, and the mixed in Figure 6.8. Besides the graphs, Table 6.2 shows the final accuracies, i.e., the accuracy of the last epoch for all the models trained, for both OBR and MBR tasks. The table also depicts the average string similarity percentage, calculated using the Levenshtein distance shown in Section 4.5.

Figure 6.2: Graph displaying the progression for the training of the One-Step Beta Reduction task, for the random datasets, over the three different conventions.



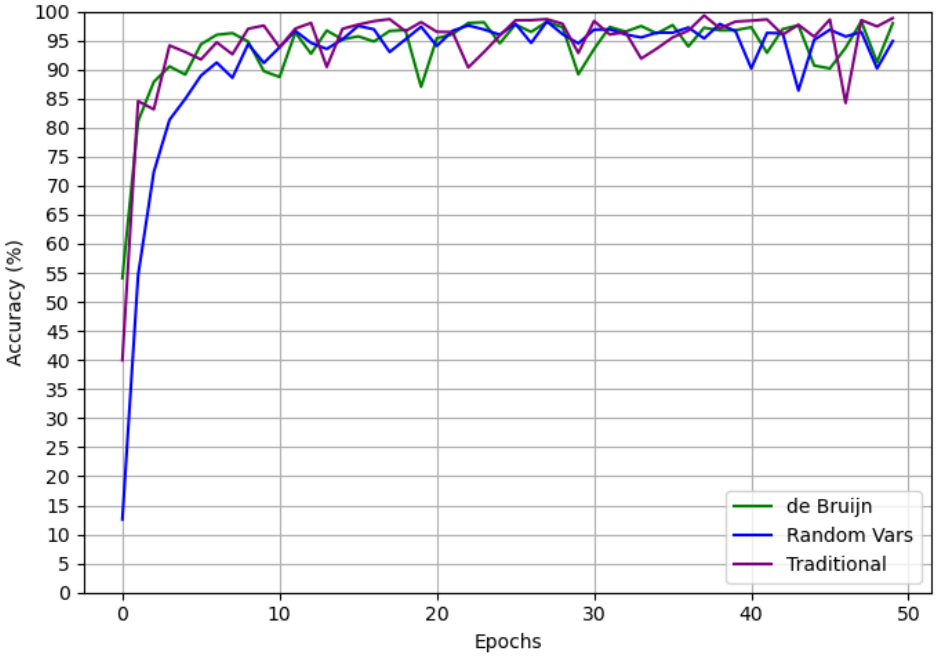
Source: The authors.

Figure 6.3: Graph displaying the progression for the training of the One-Step Beta Reduction task, for the closed bool datasets, over the three different conventions.



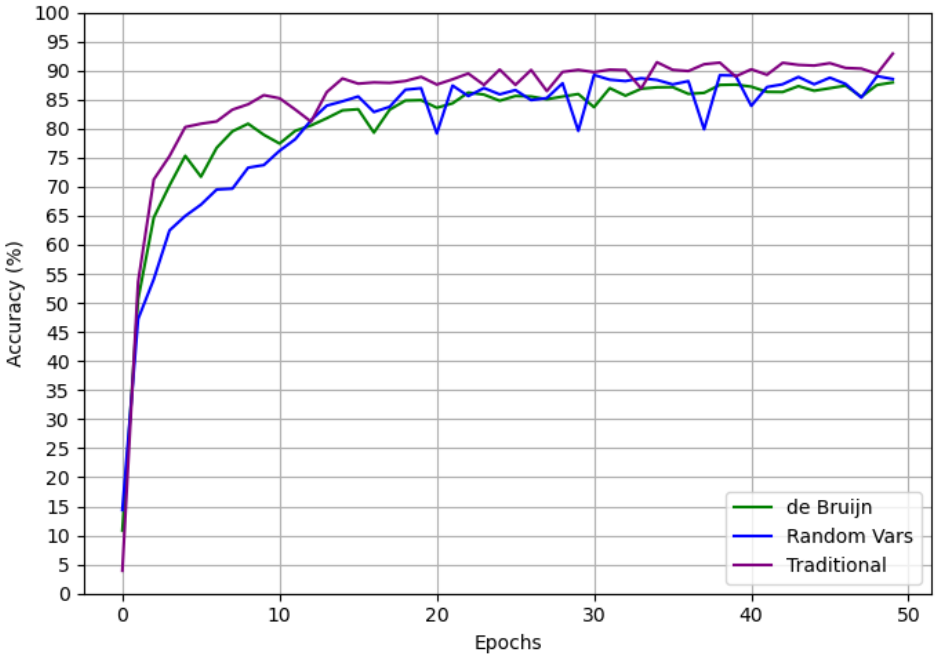
Source: The authors.

Figure 6.4: Graph displaying the progression for the training of the One-Step Beta Reduction task, for the open bool datasets, over the three different conventions.



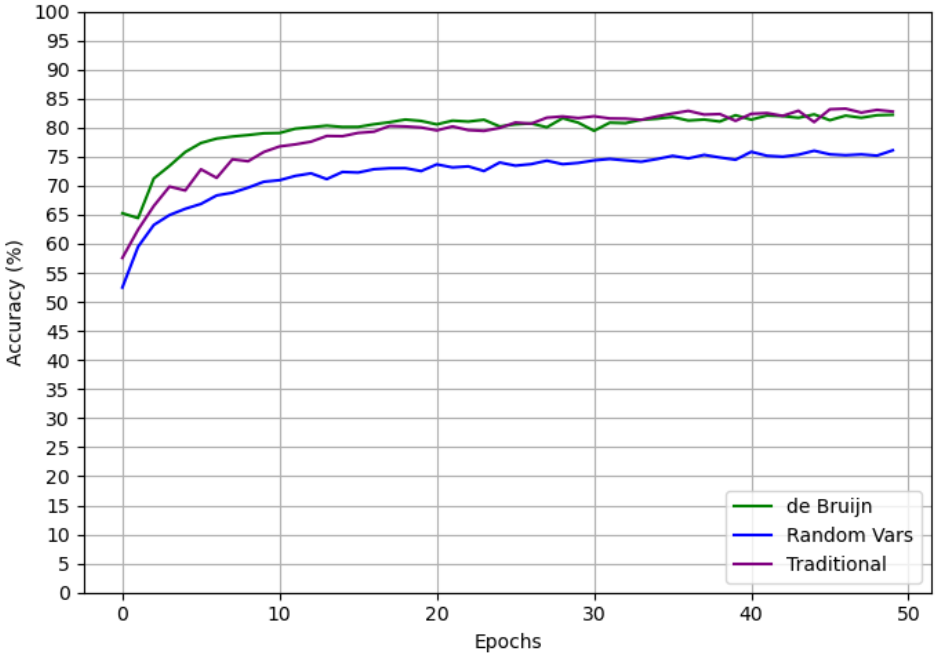
Source: The authors.

Figure 6.5: Graph displaying the progression for the training of the One-Step Beta Reduction task, for the mixed datasets, over the three different conventions.



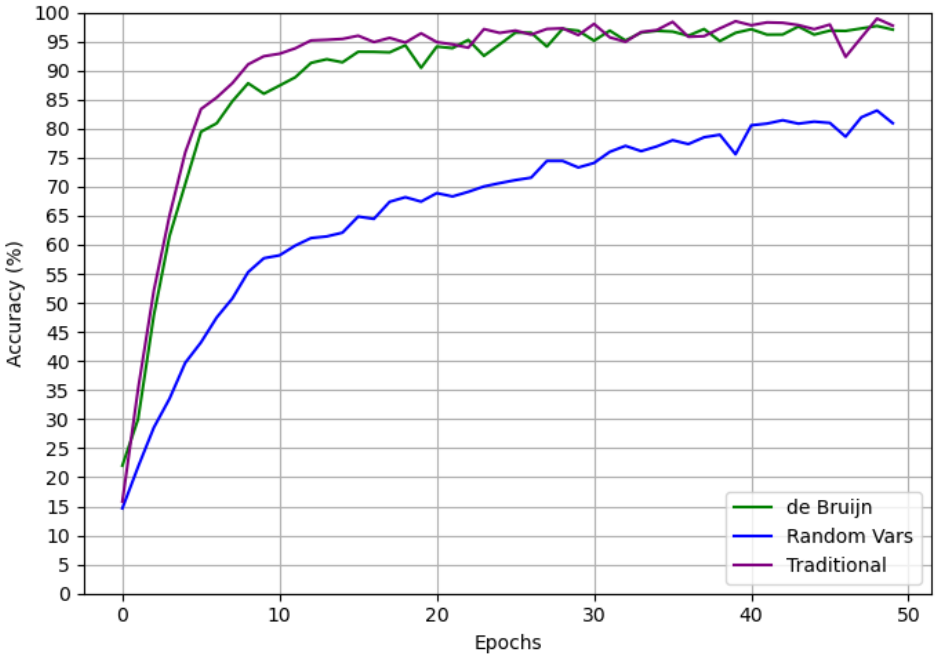
Source: The authors.

Figure 6.6: Graph displaying the progression for the training of the Multi-Step Beta Reduction task, for the closed bool datasets, over the three different conventions.



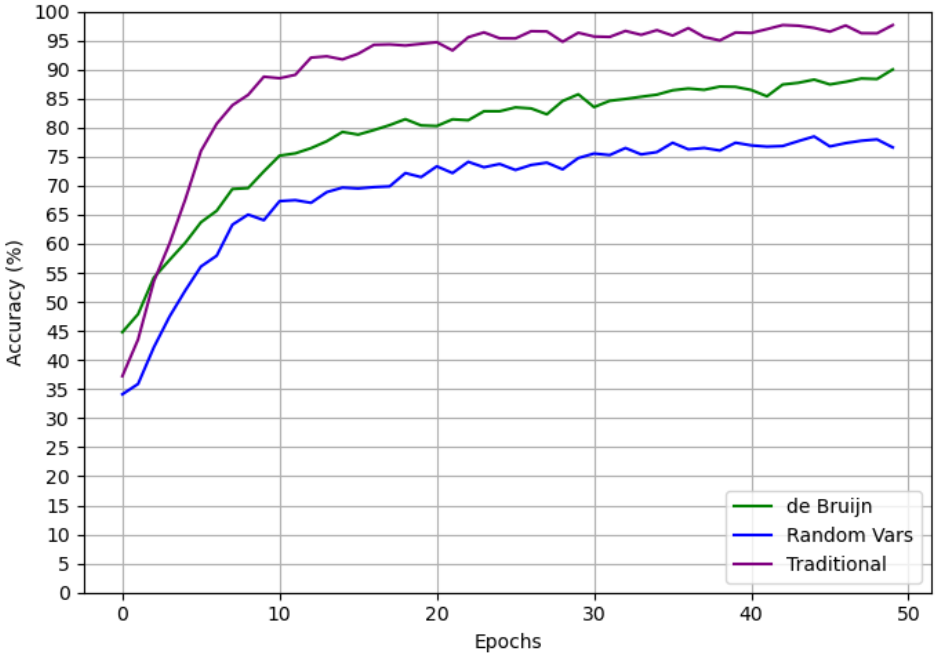
Source: The authors.

Figure 6.7: Graph displaying the progression for the training of the Multi-Step Beta Reduction task, for the open bool datasets, over the three different conventions.



Source: The authors.

Figure 6.8: Graph displaying the progression for the training of the Multi-Step Beta Reduction task, for the mixed datasets, over the three different conventions.



Source: The authors.

Table 6.2: Accuracy and the average string similarity for the evaluation of the models trained. * Rounded from 0.998....

Task	Lambda Set	Convention	ACC (%)	STR SIM (%)
OBR	random	trad	88.89	99.83
		random vars	69.30	99.51
		de Bruijn	67.84	99.34
	closed bool	trad	99.73	100.00 *
		random vars	98.10	99.98
		de Bruijn	98.16	99.99
	open bool	trad	98.82	99.99
		random vars	94.88	99.95
		de Bruijn	97.94	99.97
	mixed	trad	92.88	99.89
		random vars	88.52	99.77
		de Bruijn	87.93	99.73
MBR	closed bool	trad	82.75	97.97
		random vars	76.08	97.06
		de Bruijn	82.20	96.49
	open bool	trad	97.70	99.92
		random vars	80.92	98.19
		de Bruijn	97.02	99.77
	mixed	trad	97.63	99.89
		random vars	76.58	98.15
		de Bruijn	89.99	98.64

Source: The authors.

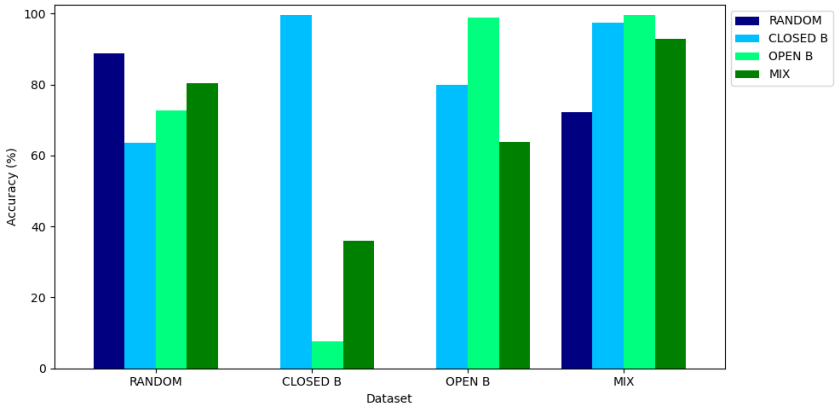
6.2 Evaluation Across Datasets

In this section, we show the results obtained by some additional evaluations with the already trained models, for both the OBR and MBR tasks. We use a model trained with one dataset to evaluate the other datasets that use the same convention and are designed for the same task. For example, we take the model that trained on the *obr_cb_db* dataset and evaluate how it performs on the *obr_rand_db*, *obr_ob_db* and *obr_mix_db* datasets. We use the test datasets to perform these evaluations.

For the OBR task, the evaluation results for the models trained using the traditional convention can be found in Figure 6.9, the random vars convention in Figure 6.10, and the de Bruijn convention in Figure 6.11. For the MBR task, the evaluation results for the models trained using the traditional convention can be found in Figure 6.12, the random vars convention in Figure 6.13 and the de Bruijn convention in Figure 6.14.

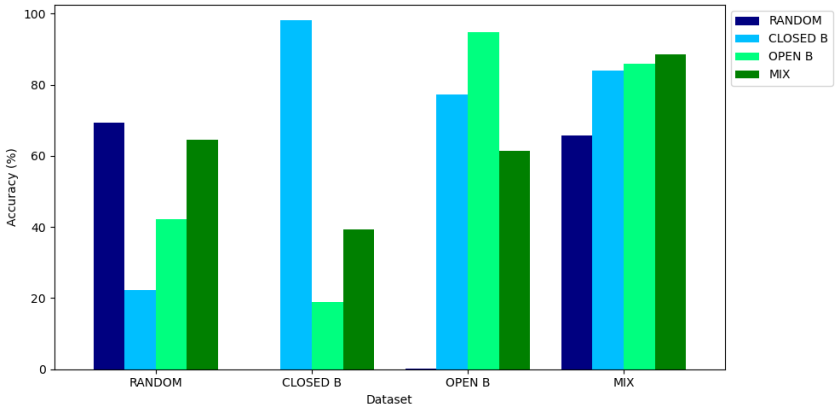
Besides the graphs, tables 6.3 and 6.4 shows the values found for the evaluations with the trained models, for the OBR and MBR tasks, respectively.

Figure 6.9: Graph showing the evaluation accuracy (%) for each model, represented by the grouped bars, and each dataset, represented by the different colors, for the OBR task using the traditional convention.



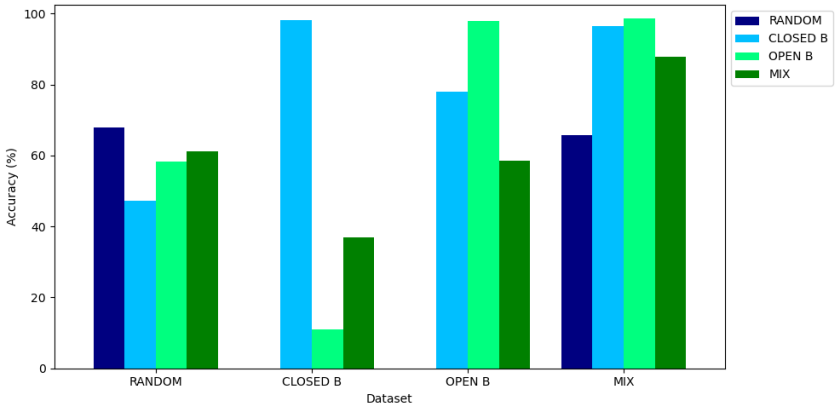
Source: The authors.

Figure 6.10: Graph showing the evaluation accuracy (%) for each model, represented by the grouped bars, and each dataset, represented by the different colors, for the OBR task using the random vars convention.



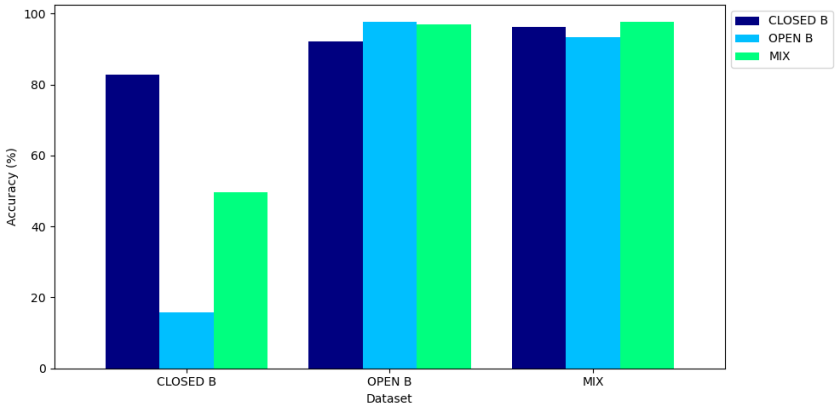
Source: The authors.

Figure 6.11: Graph showing the evaluation accuracy (%) for each model, represented by the grouped bars, and each dataset, represented by the different colors, for the OBR task using the de Bruijn convention.



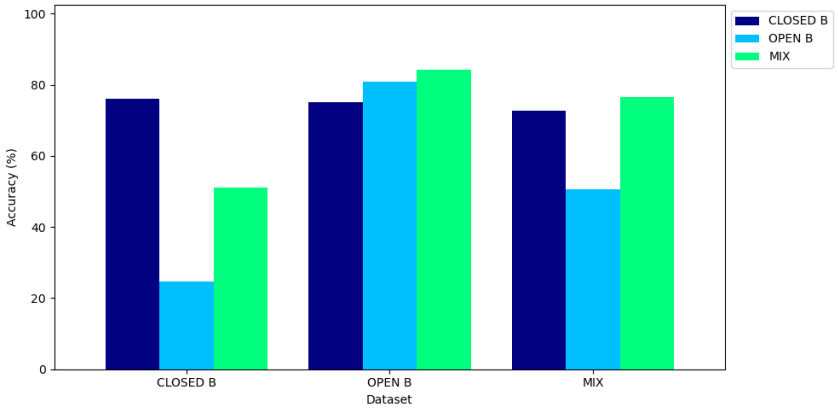
Source: The authors.

Figure 6.12: Graph showing the evaluation accuracy (%) for each model, represented by the grouped bars, and each dataset, represented by the different colors, for the MBR task using the traditional convention.



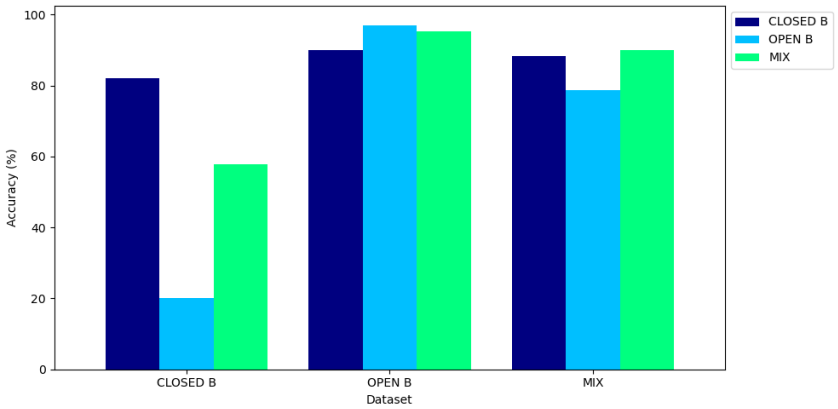
Source: The authors.

Figure 6.13: Graph showing the evaluation accuracy (%) for each model, represented by the grouped bars, and each dataset, represented by the different colors, for the MBR task using the random vars convention.



Source: The authors.

Figure 6.14: Graph showing the evaluation accuracy (%) for each model, represented by the grouped bars, and each dataset, represented by the different colors, for the MBR task using the de Bruijn convention.



Source: The authors.

Table 6.3: Accuracy (%) for the evaluation of the models over different datasets, for the OBR task. For each of the three different conventions (trad, random vars, and De Bruijn), the model trained with each dataset (rows) was evaluated with each dataset (columns). The last column indicates the average accuracy of the model over the different datasets.

Convention	Lambda Set	random	closed bool	open bool	mixed	AVERAGE
traditional	random	88.89	63.69	72.66	80.47	76.43
	closed bool	0.00	99.73	7.73	35.92	35.85
	open bool	0.04	80.01	98.82	63.79	60.67
	mixed	72.26	97.42	99.62	92.88	90.55
random vars	random	69.30	22.17	42.27	64.65	49.60
	closed bool	0.05	98.10	18.82	39.26	39.06
	open bool	0.17	77.24	94.88	61.53	58.46
	mixed	65.77	83.90	85.92	88.52	81.03
de Bruijn	random	67.84	47.15	58.35	61.14	58.62
	closed bool	0.00	98.16	10.96	36.87	36.50
	open bool	0.01	77.93	97.94	58.59	58.62
	mixed	65.70	96.39	98.71	87.93	87.18

Source: The authors.

Table 6.4: Accuracy (%) for the evaluation of the models over different datasets, for the MBR task. For each of the three different conventions (trad, random vars, and De Bruijn), the model trained with each dataset (rows) was evaluated with each dataset (columns). The last column indicates the average accuracy of the model over the different datasets.

Convention	Lambda Set	closed bool	open bool	mixed	AVERAGE
traditional	closed bool	82.75	15.85	49.66	49.42
	open bool	92.21	97.70	96.86	95.59
	mixed	96.20	93.28	97.63	95.70
random vars	closed bool	76.08	24.79	50.98	50.62
	open bool	75.23	80.92	84.25	80.13
	mixed	72.72	50.68	76.58	66.66
de Bruijn	closed bool	82.20	20.20	57.83	53.41
	open bool	90.02	97.02	95.37	94.14
	mixed	88.43	78.64	89.99	85.69

Source: The authors.

7 DISCUSSION

In this chapter, we provide an in-depth discussion of the results, emphasizing certain critical aspects of the findings. This discussion aims to provide an analysis of the data obtained and offer insights into the possible reasons behind the observed outcomes. The discussion is made taking into account the research question proposed **"Can a Machine Learning model learn to perform computations?"** and the hypothesis:

- **H1: The Transformer model can learn to perform a one-step computation on Lambda Calculus.**
- **H2: The Transformer model can learn to perform a full computation on Lambda Calculus.**

For this, we first analyze the results from the trainings, for both the One-Step Beta Reduction and Multi-Step Beta Reduction tasks. Next, we analyze the results from the evaluations across the datasets.

7.1 Training Results

In this section, we are going to discuss the results of the trainings presented in Section 6.1. With these results, we are able to determine which datasets and conventions performed better and try to conjecture some hypotheses about what happened in the trainings.

7.1.1 One-Step Beta Reduction

In the training of the models for the OBR task, it was found that each model achieved an accuracy of at least 67.84%. However, when only the best conventions were considered, each model achieved a minimum accuracy of 88.89%. Additionally, one model achieved a remarkable 99.73% accuracy. These findings can be seen in Table 6.2, and they highlight the high level of accuracy and effectiveness of the models, particularly when utilizing the optimal conventions, which supports hypothesis H1.

The metric for string similarity can also be seen in Table 6.2 and indicates that, despite incorrectly predicting some terms, the model was able to accurately predict a

significant portion of those terms, with all similarities being at least 99.34%. If we take only the best conventions, this number goes up to 99.83%. Also, some models achieved an outstanding performance of over 99.99% for this metric. Additionally, for example, the model trained with the random dataset with the de Bruijn convention got a final accuracy of 67.84%. However, the string similarity metric for the same training was 99.34%. This illustrates how much the model got the incorrect answers close to the correct ones.

Upon analyzing the performance of the models on different datasets, it is evident that the closed bool and open bool datasets were easier to learn compared to other datasets, as we can see in Figures 6.2, 6.3, 6.4 and 6.5. The boolean datasets achieved good accuracies in a shorter span of time and presented similar results between themselves. The random dataset was the hardest to learn, as shown in figure 6.2. We think this is due to the absence of more defined patterns among the terms. The mixed dataset, as expected, fell between the random and the boolean datasets in terms of difficulty to learn. However, its accuracy surprised us, since it was the most diverse dataset, meaning that it learned to perform the OBR task both for random and boolean terms, with high accuracies (92.88% for the optimal convention, as shown in Table 6.2).

It is worth noting that, although the accuracies for the random and the mixed datasets were comparatively lower than those of the boolean datasets, the graphs illustrating their performance present a growing pattern, as illustrated in Figures 6.2 and 6.5. This indicates that further training with more epochs could yield higher accuracies for these datasets.

Analyzing the performance of the models that use different conventions in Table 6.2, it can be observed that the traditional convention consistently outperformed the other two conventions, which exhibited similar levels of performance. However, the only training that the convention really made a difference was on the random dataset, which we saw, was the hardest one to learn. We suppose that, for the other datasets, the difference of the convention did not matter because it was so easy for the model to learn that even the "harder" conventions were not a problem. We also think that the traditional convention performed overall better than the other two conventions because, despite being based on a simpler convention, the beta reduction is more intricate for the de Bruijn convention, and consequently, is harder to learn. Furthermore, when compared to the random vars convention, the traditional convention, with its ordered naming rule, tends to provide the model with more predictable outcomes.

7.1.2 Multi-Step Beta Reduction

For the training of the models for the MBR task, every model attained a minimum accuracy of 76.08%. Nevertheless, when considering only the best conventions, each model exhibited an accuracy of at least 82.75%. Furthermore, one model achieved an exceptional 97.70% accuracy. These outcomes, which can be seen in Table 6.2, emphasize the effectiveness and high accuracy of the models, especially when using the optimal conventions, which supports hypothesis H2.

The metric for string similarity, found at Table 6.2, indicates again that, even though the model made incorrect predictions for some terms, it accurately predicted a significant portion of those terms, with all similarities being no less than 96.49%. Considering only the best conventions, this number increases to 97.97%. Additionally, some models performed exceptionally well, achieving up to 99.92% for this metric. Again, the models that did not obtain a good accuracy got an outstanding performance on this metric. For instance, the model trained with the mixed dataset, using the random vars convention, obtained an accuracy of 76.58%. Nevertheless, the string similarity metric for the same training was 98.15%. This shows that for this task, the models also got the wrong predictions very close to the correct ones.

For the closed bool dataset in the MBR task, it is important to notice that the set of possible terms that the model should predict is small (namely, *true* and *false*). For the traditional convention and, especially, for the random vars convention, the *true* and *false* terms are not always the same term since there are many alpha-equivalent terms for *true* and *false* using the English alphabet. But in the DB case, there are only two distinct terms for the *true* and *false* ("*L L 2*" and "*L L 1*", respectively). Thus, one might expect that the closed bool dataset would be easier to learn since there are only a few possible terms for the model to predict (only two in the DB case), while the open bool, on the other hand, had output terms that differ dramatically from one another.

But the opposite was actually observed, as we can see in Figures 6.6 and 6.7. The closed bool dataset was found to be harder to learn than the open bool dataset, with the model that trained on it having significantly lower accuracies than the open bool model, which seems counter-intuitive. Our hypothesis is that, precisely because the terms were so similar in the closed bool dataset, the model resorted to guessing the output term from a limited set of possibilities, based on some features of the inputs, instead of learning to perform the reductions. But, since this was not possible for the open bool dataset, the

model was forced to actually learn to perform the multi-step beta reduction on the input terms. The fact that the closed bool model already starts the training with around 55% accuracy also corroborates our hypothesis that the model is learning to guess from a limited set instead of learning the reductions.

The model trained with the mixed dataset seems to have overcome this issue, as we can see in Figure 6.8. Considering the traditional convention, the accuracy of the model was similar to the accuracy of the model trained on the open bool dataset, even with half of its terms having come from the closed bool dataset. This actually supports our previous hypothesis since we think that having more variability on the terms forced the model to learn the reductions instead of only guessing between a small set of possible outcomes.

For the trainings on different conventions, Table 6.2 shows that the random vars convention had the worst accuracies for the three datasets. However, only the models trained on the open bool and on the mixed datasets presented a large gap between different conventions. We suppose that the naming convention did not change the guessing factor on the learning process for the models that trained on the closed bool. What is interesting is that the de Bruijn convention led to accuracies as good as the traditional convention and significantly better than the random vars convention for the models trained on the open bool and closed bool datasets. This was unexpected since the β-reduction on the de Bruijn notation is more intricate than on the traditional notation, which the other two conventions use. This shows how important the order for the naming of the variables is for this task. For the model trained on the mixed dataset, the order of the different conventions was more aligned with the expected, with the traditional being the best convention, followed by the other two. But this result, although expected, was unusual since the other models did not follow this order.

7.2 Evaluations Across Datasets

In this section, we are going to discuss the results of the evaluations across the datasets presented in Section 6.2. These evaluations can give us some insights as to whether the model really learned the reductions or if it just learned the reductions for that specific set of terms.

7.2.1 One-Step Beta Reduction

The evaluations for this task have yielded promising results, especially for the models trained with the mixed dataset. These models produced the best average accuracies for all models in the OBR tasks, as seen in Table 6.3. This shows that, as expected, the model trained with the mixed dataset were able to better capture the diversity of terms present on the different datasets.

The models trained with both boolean datasets performed poorly for the evaluation with the random dataset, with accuracies close to 0%, as we can see in Table 6.3. We suppose that this happened because the terms in the random dataset are very distinct from the terms from the boolean datasets. Also, since the opposite did not happen, we think that the random dataset contains terms that are actually harder to learn, as we presumed in the previous section.

As expected, almost every model had a better accuracy on the evaluation with the dataset it was trained on rather than with the others, as shown in Table 6.3. But one result that may be seen as counter-intuitive is the evaluation of the model trained with the mixed dataset. It had better accuracy for the open bool and closed bool datasets for the traditional and de Bruijn conventions, which can be seen in Figures 6.9 and 6.11. We, again, suppose that this happened because the random dataset is the hardest to learn. Thus, since the mixed dataset has 1/3 of its terms from the random dataset, it turns out to be harder to learn than the closed bool and open bool datasets. So, it ends up giving a better evaluation for those two datasets than the dataset it was trained with, which contains terms from the random dataset.

Another result that is interesting is that the model trained on the open bool dataset was able to extrapolate and get good accuracies for the evaluation of the closed bool dataset, with a minimum accuracy of 77.24%. But the opposite did not happen, with accuracies as low as 7.73%, as we can see in Table 6.3. This shows that the model trained with the open bool dataset had a better extrapolation quality, probably because their terms are more diverse than the closed bool dataset.

Aside from what was mentioned, the different conventions did not present a significant difference between the evaluations, as shown in Figures 6.9, 6.10 and 6.11.

7.2.2 Multi-Step Beta Reduction

The evaluations for this task have, again, yielded good results, particularly for the models trained with the open bool dataset. As shown in Table 6.4, the use of the open bool dataset led to better average accuracies for almost all models in the MBR task.

Table 6.4 demonstrates that, as anticipated, the majority of models performed better in the evaluations using the dataset they were trained on, as opposed to the other datasets. However, the models trained with the open bool dataset had accuracies quite close to one another for the three datasets evaluated, as seen in Figures 6.12, 6.13 and 6.14. In fact, as seen in figure 6.10, it presented a better accuracy for the mixed dataset rather than the dataset it was trained on. We presume that this happened because the model trained with the open bool dataset was able to generalize better than the others.

Again, the models trained with the closed bool did not extrapolate and got good accuracies for the other datasets, especially the closed bool, with accuracies as low as 15.85%, as seen in Table 6.4. But the opposite happened, with the open bool models getting a minimum of 75.23% of accuracy for the closed bool dataset. We think that this happened for the same reason discussed on Section 7.1.2, which is that the model trained on the closed bool dataset just learned to guess the output from a limited set of possible terms, not actually learning the β-reduction.

Once more, apart from what was stated, the different conventions did not present any main differences between the evaluations, as shown in Figures 6.12, 6.13 and 6.14.

8 CONCLUSION AND FUTURE WORK

In this research, the goal was to explore the following research question: **"Can a Machine Learning model learn to perform computations?"**. To investigate this, we proposed to use a machine learning model, the Transformer, to learn to perform computations using the λ-Calculus as the underlying formalism. To accomplish this, the study proposed to teach the model both One-Step Beta Reduction, which represents a one-step computation, and Multi-Step Beta Reduction, which represents a full computation. Thus, two hypotheses were formulated:

- **H1: The Transformer model can learn to perform a one-step computation on Lambda Calculus.**
- **H2: The Transformer model can learn to perform a full computation on Lambda Calculus.**

We started presenting a theoretical background about the syntax, semantics, and essential concepts of the λ-Calculus. Subsequently, the focus shifted to the field of artificial intelligence and its sub-field, machine learning, specifically the neural network model. The challenges posed by traditional neural networks for this task were discussed, followed by the introduction of seq2seq models, which can handle inputs and outputs of any length. Finally, the study presented the Transformer model as the most suitable solution due to its self-attention mechanism.

Later, we discussed the methodology employed in this study. In this discussion, we provided an overview of the process used to generate the datasets, discussed the specifics of the training process, shared some technical information about the machine and code we used, and elaborated on how we conducted the experiments and analyzed the results. as well as how we generate the datasets used. Then, we examined the specific details of how we generate the datasets, explaining each step of the generation.

Finally, we presented the results obtained in this work, using tables and graphs to display the data. We also presented a discussion about these results, where we highlighted the main findings of this study.

In this chapter, we present the conclusions of our work, highlighting the main contributions we believe this research provided. Lastly, we propose suggestions for future works based on the present study.

8.1 Conclusions

Through comprehensive experimentation and analysis, it was demonstrated that the Transformer model is capable of capturing the syntactic and semantic features of λ-calculus, allowing for accurate and efficient predictions. The results obtained were positive, with overall good accuracies for both tasks at hand. For the One-Step Beta Reduction, we got accuracies up to 99.73%, and string similarity metric of over 99.99%. For the Multi-Step Beta Reduction, we obtained accuracies of up to 97.70%, and string similarity metric exceeding 99.90%. Besides that, the models presented a good generalization performance across different datasets. Due to limitations of hardware and time, our models trained for just 50 epochs. Considering that is a pretty low number compared with substantial trainings of large models, and that we did not do a search for the optimal hyperparameters, we can assure that the accuracy of our models can be even higher than what was presented here.

These results illustrate the effectiveness of the model in learning the desired tasks and support the two hypotheses raised in this study, and, subsequently, the research question proposed. Also, these results showed that the Transformer's self-attention mechanism is well suited for capturing the dependencies between variables and functions in λ-Calculus.

This work sheds light on the potential of deep learning models to learn abstract mathematical concepts and provides a foundation for future work in this area. Furthermore, this research highlights the importance of incorporating structured and symbolic knowledge into deep learning models. By applying the Transformer model to learn λ-Calculus, we believe that this dissertation has contributed to the advancement of the field of machine learning.

8.1.1 Main Contributions

We believe that the methods and results presented in this work have yielded some significant outcomes for future researches. The main contributions that have resulted from this research can be summarized as follows:

- Dataset generation: Since datasets for lambda terms and reductions did not exist, we built the generation for these datasets from scratch. These datasets and gener-

ation methods can be used in future researches in the lambda calculus domain.

- Lambda calculus learning: The outcomes from learning the reductions of lambda calculus are promising and hold potential implications for future researches in the field of AI and computer programs.
- Functional programming learning: The results obtained in this study can be taken into account for shifting the programming paradigm from imperative to functional in future researches in the field of learning to compute.

8.2 Future Work

Since our work changed the paradigm traditionally used for the underlying formalism of the computations intended to be learned, from the imperative to the functional paradigm, we open a new set of possibilities for new studies. Also, our work had limitations, both in time and computational power. Addressing these limitations through increased resources would provide a way for further advancements. Furthermore, the research was limited to a formalism - the λ-Calculus, that can be switched or extended. Thus, for future works, we propose the following suggestions:

- Increased training: The current study trained the model for a limited number of epochs. Further research could aim to train the best notation for more epochs to see if performance can be improved.
- Hyperparameter optimization: The study used a set of predefined hyperparameters for the Transformer model. A thorough search for the optimal hyperparameters could be conducted to find the best set of hyperparameters for learning the Lambda Calculus.
- Improved error analysis: The study provided a preliminary error analysis, but further work could aim to conduct a more in-depth error analysis to better understand the types of mistakes the model is making and to identify areas for improvement.
- Incorporating other formalisms: This study focused on learning Lambda Calculus, but there are other formalisms such as Combinatorial Logic and Turing Machines that could be trained by the model and compared with the current work.
- Solve typing problems: Learn a typed λ-Calculus to solve some typing problems, which can be uncomputable: well-typedness, type assignment, type checking, and type inhabitation.

- Learn more complex versions of the λ-Calculus: Learn the not-pure λ-Calculus, with numbers and arithmetical and boolean operations already embbeded.
- Learn to compute a functional programming language: Learn a functional programming language, that is based on λ-Calculus, such as Haskell or Lisp.
- Learn to detect loops: Use the same methods for the training, but instead of learning to perform the computations, learn to identify if a λ-term does not have a normal form, i.e., if it is going to enter a loop when applying the reductions.

These future work suggestions have the potential to bring further advancements in the application of machine learning models to the field of learning to execute computer programs, especially using functional programming as the base paradigm, and contribute to a deeper understanding of the underlying computational process.

REFERENCES

BARENDREGT, H. **The Lambda Calculus: Its Syntax and Semantics**. North-Holland, 1984. (North-Holland Linguistic Series). ISBN 9780444867483. Available from Internet: <https://books.google.com.br/books?id=eMtTAAAAYAAJ>.

BARENDREGT, H. The impact of the lambda calculus in logic and computer science. **Bulletin of Symbolic Logic**, Cambridge University Press, v. 3, n. 2, p. 181–215, 1997.

BENGIO, Y. Practical recommendations for gradient-based training of deep architectures. **Neural Networks: Tricks of the Trade: Second Edition**, Springer, p. 437–478, 2012.

BISHOP, C. M.; NASRABADI, N. M. **Pattern recognition and machine learning**. [S.l.]: Springer, 2006.

BROWN, T. B. et al. Language models are few-shot learners. In: . [S.l.: s.n.], 2020. v. 2020-December. ISSN 10495258.

BRUIJN, N. G. D. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In: ELSEVIER. **Indagationes Mathematicae (Proceedings)**. [S.l.], 1972. v. 75, n. 5, p. 381–392.

CHUNG, J. et al. Empirical evaluation of gated recurrent neural networks on sequence modeling. **arXiv preprint arXiv:1412.3555**, 2014.

CHURCH, A. A set of postulates for the foundation of logic. **The Annals of Mathematics**, v. 33, 1932. ISSN 0003486X.

CHURCH, A. An unsolvable problem of elementary number theory. **American Journal of Mathematics**, v. 58, 1936. ISSN 00029327.

CHURCH, A. A formulation of the simple theory of types. **Journal of Symbolic Logic**, v. 5, 1940. ISSN 0022-4812.

DAVIS, M. **Engines of Logic: Mathematicians and the Origin of the Computer**. [S.l.]: Norton, 2001. ISBN 9780393322293.

DIVERIO, T. A.; MENEZES, P. **Teoria da Computação - Máquinas Universais e Computabilidade**. [S.l.]: Bookman Editora, 2009.

GARCEZ, A. d.; LAMB, L. C. Neurosymbolic ai: the 3rd wave. **arXiv preprint arXiv:2012.05876**, 2020.

GARCEZ, A. S. d'Avila; LAMB, L. C.; GABBAY, D. **Neural-Symbolic Cognitive Reasoning**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. Available from Internet: <http://link.springer.com/10.1007/978-3-540-73246-4>.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep learning**. [S.l.]: MIT Press, 2016.

GRAVES, A.; WAYNE, G.; DANIHELKA, I. Neural turing machines. **arXiv preprint arXiv:1410.5401**, 2014.

HAMBLIN, C. L. Translation to and from polish notation. **The Computer Journal**, v. 5, 1962. ISSN 0010-4620.

HARRIS, D.; HARRIS, S. L. **Digital design and computer architecture**. [S.l.]: Morgan Kaufmann, 2010.

HINDLEY, J. R.; CARDONE, F. History of lambda-calculus and combinatory logic. **History**, v. 5, 2006. ISSN 03666999.

JUNG, A. A short introduction to the lambda calculus. **Computer**, 2004.

KAISER, Ł.; SUTSKEVER, I. Neural gpus learn algorithms. **arXiv preprint arXiv:1511.08228**, 2015.

KAUTZ, H. The third ai summer: Aaai robert s. engelmore memorial lecture. **AI Magazine**, v. 43, n. 1, p. 105–125, 2022.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. **arXiv preprint arXiv:1412.6980**, 2014.

LAMPLE, G.; CHARTON, F. Deep learning for symbolic mathematics. **arXiv preprint arXiv:1912.01412**, 2019.

LEVENSHTEIN, V. I. et al. Binary codes capable of correcting deletions, insertions, and reversals. In: SOVIET UNION. **Soviet physics doklady**. [S.l.], 1966. v. 10, n. 8, p. 707–710.

LIPTON, Z. C.; BERKOWITZ, J.; ELKAN, C. A critical review of recurrent neural networks for sequence learning. **arXiv preprint arXiv:1506.00019**, 2015.

MACHADO, R. An introduction to lambda calculus and functional programming. In: . [S.l.: s.n.], 2013.

MADDISON, C.; TARLOW, D. Structured generative models of natural source code. In: PMLR. **International Conference on Machine Learning**. [S.l.], 2014. p. 649–657.

MICHAELSON, G. **An Introduction to Functional Programming Through Lambda Calculus**. Dover Publications, 2011. (Dover books on mathematics). ISBN 9780486478838. Available from Internet: <https://books.google.com.br/books?id=gKvwPtvsSjsC>.

MOU, L. et al. Building program vector representations for deep learning. **arXiv preprint arXiv:1409.3358**, 2014.

MURPHY, K. P. **Machine learning: a probabilistic perspective**. [S.l.]: MIT press, 2012.

PIERCE, B. C. **Types and programming languages**. [S.l.]: MIT press, 2002.

ROOSE, K. The brilliance and weirdness of chatgpt. **The New York Times**, 2022. Available from Internet: <https://www.nytimes.com/2022/12/05/technology/chatgpt-ai-twitter.html>.

RUMELHART, D. E.; MCCLELLAND, J. L. **Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations**. [S.l.]: MIT Press, 1986.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence A Modern Approach**. [S.l.: s.n.], 2021. ISSN 1098-6596.

SHUTE, M. Computer architecture: a quantitative approach. **Microelectronics Journal**, v. 24, 1993. ISSN 00262692.

SIPSER, M. Introduction to the theory of computation. **ACM Sigact News**, ACM New York, NY, USA, v. 27, n. 1, p. 27–29, 1996.

SMITH, L. N. : Part 1–learning rate, batch size, momentum, and weight decay. **arXiv preprint arXiv:1803.09820**, 2018.

SUTSKEVER, I.; VINYALS, O.; LE, Q. V. Sequence to sequence learning with neural networks. **Advances in Neural Information Processing Systems**, v. 4, p. 3104–3112, 2014. ISSN 10495258.

TRASK, A. et al. Neural arithmetic logic units. **Advances in neural information processing systems**, v. 31, 2018.

TURING, A. M. On computable numbers, with an application to the entscheidungsproblem. **Proceedings of the London Mathematical Society**, s2-42, 1936. ISSN 1460244X.

TURING, A. M. Computability and λ-definability. **Journal of Symbolic Logic**, v. 2, 1937. ISSN 0022-4812.

VASWANI, A. et al. Attention is all you need. **Advances in Neural Information Processing Systems**, v. 2017-Decem, p. 5999–6009, 2017. ISSN 10495258.

ZAREMBA, W.; SUTSKEVER, I. Learning to execute. **arXiv preprint arXiv:1410.4615**, 2014.

APPENDIX A — RESUMO EXPANDIDO

Tradicionalmente, redes neurais têm sido utilizadas para tratar de problemas estatísticos, como reconhecimento de fala, classificação de imagem, processamento de linguagem natural, entre outros. Por outro lado, tarefas simbólicas, como operações aritméticas, inferência lógica e execução de programas de computador não estavam na mira da IA conexionista. Porém, avanços nesses campos levaram à emergência de modelos que juntam a IA conexionista e IA simbólica, caindo no domínio neurosimbólico. Neste trabalho, estamos interessados em investigar a capacidade de modelos de redes neurais, particularmente o Transformer, em aprender a executar computações, domínio que é visto como complexo demais para redes neurais. Para atingir esse objetivo, utilizamos o cálculo lambda, um formalismo simples, porém poderoso, como base para nossos estudos. O cálculo lambda foi desenvolvido por Alonzo Church na década de 1930, como um meio de resolver o Entscheidungsproblem. É um sistema formal baseado em funções, capturando a noção de definição e aplicação de funções. Além da importância histórica, o Cálculo Lambda é um formalismo compacto e turing-completo, sendo base para várias linguagens de programação funcionais modernas. Em sua essência, o Cálculo Lambda pode ser visto como uma linguagem de programação, sendo constituído de termos que podem sofrer reduções. Os termos podem ser interpretados como programas e as reduções podem ser vistas como as computações executadas sobre os termos. Aplicar uma única redução a um termo representa um passo de computação no formalismo. Aplicar reduções sucessivas até não existir mais reduções a serem aplicadas (caso o termo não entre em loop) representa uma computação total. Neste trabalho, estaremos interessados na execução dessas duas tarefas, as quais chamaremos de OBR (one-step beta reduction) para a computação de um passo e MBR (multi-step beta reduction) para a computação total. Para aprender a executar as computações, utilizaremos um modelo neural. Porém, redes neurais tradicionais lidam somente com entradas e saídas de tamanho fixo. Por isso, utilizaremos um modelo sequência-para-sequência (seq2seq), onde a entrada e a saída podem ter tamanhos variáveis. Especificamente, utilizaremos o Transformer, modelo paralelizável baseado no mecanismo de self-attention, amplamente utilizado para tarefas seq2seq, inclusive simbólicas.

Para guiar nosso trabalho, propomos a seguinte questão de pesquisa: “Um modelo de aprendizado de máquina consegue aprender a executar computações?”. Para tentar responder essa questão, formulamos duas hipóteses: (i) O modelo Transformer consegue

aprender a executar uma computação de um passo no cálculo lambda; (ii) O modelo transformer consegue aprender a executar uma computação total no cálculo lambda. O foco principal da nossa pesquisa está alinhado com a segunda hipótese, porém, escolhemos começar com uma hipótese mais simples como ponto de partida. Como nosso modelo de aprendizado aprende a partir de dados, e não havia referências para a geração de datasets com termos lambda na literatura, nós desenvolvemos o processo de geração dos datasets do zero. Nesse processo, desenvolvemos dois métodos diferentes. O primeiro gera termos lambda aleatoriamente, mesmo que eles não façam parte de nenhuma codificação conhecida. O segundo, gera termos lambda dentro de um domínio específico, seguindo uma codificação. Em nosso trabalho, o domínio escolhido para o segundo método foi a álgebra booleana. Para os resultados, apresentamos a acurácia dos modelos em cada época treinada, bem como uma métrica de similaridade de string, baseada na distância de Levenshtein. Além disso, apresentamos uma avaliação de cada modelo treinado no trabalho com os outros datasets utilizados.

Os resultados obtidos foram satisfatórios para ambas as tarefas e para a avaliação. Para a tarefa de OBR, obtemos acurácias de até 99.73%, e similaridade de strings de mais de 99.99%. Para a tarefa de MBR, obtemos acurácia de até 97.97% e similaridade de strings ultrapassando 99.90%. Além disso, os modelos apresentaram uma boa capacidade de generalização entre os datasets diferentes. Por limitações de hardware e tempo, nossos modelos treinaram por apenas 50 épocas cada, número pequeno comparado à treinos de grandes modelos. Além disso, nós não fizemos uma busca pelos melhores hiperparâmetros. Considerando essas limitações, podemos assegurar que os resultados obtidos podem ser ainda melhores do que os mostrados neste trabalho. Esses resultados mostran a efetividade do modelo em aprender as tarefas propostas e suportam as duas hipóteses levantadas neste estudo. Cremos que os métodos e resultados apresentados neste trabalho renderam alguns resultados significativos para pesquisas futuras. As principais contribuições oriundas deste trabalho são: (i) geração de datasets: métodos para geração de datasets de termos lambda e suas reduções; (ii) aprendizado do cálculo lambda: principal resultado do trabalho, mostra como esse aprendizado é promissor e tem potenciais implicações para trabalhos futuros na área; (iii) aprendizado de linguagens funcionais: os resultados obtidos podem ser levados em consideração para uma mudança de paradigma, de imperativo para funcional, na área de aprendizado de execução de programas de computador.