TOKEN INTERCHANGEABILITY AND ALPHA-EQUIVALENCE: ENHANCING THE GENERALIZATION CAPACITY OF LANGUAGE MODELS FOR FORMAL LOGIC

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

İLKER IŞIK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JULY 2025

Approval of the thesis:

TOKEN INTERCHANGEABILITY AND ALPHA-EQUIVALENCE: ENHANCING THE GENERALIZATION CAPACITY OF LANGUAGE MODELS FOR FORMAL LOGIC

submitted by **İLKER IŞIK** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Naci Emre Altun
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Assoc. Prof. Dr. Ramazan Gökberk Cinbiş
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Assoc. Prof. Dr. Uluç Saranlı
Computer Engineering, METU _____

Assoc. Prof. Dr. Ramazan Gökberk Cinbiş
Computer Engineering, METU _____

Assist. Prof. Dr. Özgür Salih Öğüz
Computer Engineering, Bilkent University _____

Date: 18.07.2025

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: İlker Işık

Signature :

ABSTRACT

TOKEN INTERCHANGEABILITY AND ALPHA-EQUIVALENCE: ENHANCING THE GENERALIZATION CAPACITY OF LANGUAGE MODELS FOR FORMAL LOGIC

Işık, İlker
M.S., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Ramazan Gökberk Cinbiş

July 2025, 65 pages

Language models lack the notion of interchangeable tokens: symbols that are semantically equivalent yet distinct, such as bound variables in formal logic. This limitation prevents generalization to larger vocabularies and hinders the model's ability to recognize alpha-equivalence, where renaming bound variables preserves meaning. We formalize this machine learning problem and introduce alpha-covariance, a metric for evaluating robustness to such transformations. To tackle this task, we propose a dual-part token embedding strategy: a shared component ensures semantic consistency, while a randomized component maintains token distinguishability. Compared to a baseline that relies on alpha-renaming for data augmentation, our approach demonstrates improved generalization to unseen tokens in linear temporal logic solving, propositional logic assignment prediction, and copying with an extendable vocabulary, while introducing a favorable inductive bias for alpha-equivalence. Our findings establish a foundation for designing language models that can learn interchangeable token representations, a crucial step toward more flexible and systematic reasoning in formal domains.

Keywords: Machine Learning, Formal Methods, Language Modeling, Linear Temporal Logic

ÖZ

BELİRTEÇ DEĞİŞTİRİLEBİLİRLİĞİ VE ALFA-EŞDEĞERLİLİK: DİL MODELLERİNİN BİÇİMSEL MANTIK İÇİN GENELLEME KABİLİYETİNİN İYİLEŞTİRİLMESİ

Işık, İlker
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Ramazan Gökberk Cinbiş

Temmuz 2025 , 65 sayfa

Dil modelleri, değiştirilebilir belirteç kavramından yoksundur. Bu kavram, biçimsel mantıktaki bağlı değişkenler gibi anlamsal olarak eşdeğer ancak farklı olan sembolleri ifade eder. Bu eksiklik, daha geniş sözcük dağarcıklarına genellemeyi engeller ve modelin alfa eşdeğerliği tanıma yeteneğini engeller. Alfa eşdeğerlik, bağlı değişkenleri yeniden adlandırmanın anlamı korumasıdır. Bu çalışmada, bu makine öğrenimi sorunu formüle edildi ve bu tür dönüşümlere karşı sağlamlığı değerlendirmek için bir ölçüt olan alfa kovaryansı sunuldu. Bu görevi ele almak için, çift parçalı bir belirteç yerleştirme stratejisi öneriyoruz: paylaşılan bir bileşen anlamsal tutarlılığı sağlarken, rastgele bir bileşen belirteç ayırt edilebilirliğini koruyor. Veri artırma için alfa yeniden adlandırmaya dayanan bir yöntem ile karşılaştırıldığında, yaklaşımımız doğrusal zamansal mantık çözümünde, önermesel mantık atama tahmininde ve genişletilebilir bir sözcük dağarcığıyla kopyalamada görülmemiş belirteçlere yönelik genelleme gösterirken, alfa eşdeğerliği için olumlu bir tümevarımsal önyargı sunuyor. Bulgularımız, biçimsel (formal) alanlarda daha esnek ve sistematik akıl yürütmeye doğru önemli bir

adım olan, değiştirilebilir belirteç gösterimlerini öğrenebilen dil modelleri tasarlamak için bir temel oluşturuyor.

Anahtar Kelimeler: Makine Öğrenmesi, Biçimsel Yöntemler, Dil Modelleme, Doğrusal Zamansal Mantık

To those who don't like dedications.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my current thesis supervisor, Assoc. Prof. Dr. Ramazan Gökberk Cinbiş, and my former supervisor, Dr. Ebru Aydın Göl, for their continuous guidance and support throughout this work.

I am also grateful to the anonymous reviewers of ICLR 2025 and ICML 2025 for their constructive feedback, which greatly contributed to improving the earlier versions of this research.

I also would like to thank the jury members, namely Assoc. Prof. Dr. Uluç Saranlı and Assist. Prof. Dr. Özgür Salih Öğüz, for their valuable time and insightful evaluations.

The numerical calculations were partially performed at TÜBİTAK TRUBA, MareNostrum5, METU ImageLab, and METU ROMER resources. This project was supported in part by the project METU ADEP-312-2024-11525. Dr. Cinbis is supported by the "Young Scientist Awards Program (BAGEP)'' of Science Academy, Türkiye.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS

2D	2 Dimensional
3D	3 Dimensional
AC	AdaCos Loss Function
AP	Atomic Proposition
LLM	Large Language Model
LTL	Linear Temporal Logic
N.D.	Normal Distribution (Random embedding method)
N.P.	Neighboring Points (Random embedding method)
H.V.	Hypercube Vertices (Random embedding method)
RNN	Recurrent Neural Network

CHAPTER 1

INTRODUCTION

Following the deep learning revolution that affected numerous application areas [18], recent literature shows that deep learning based approaches also perform well in neurosymbolic reasoning tasks, such as theorem proving [29] and mathematical reasoning [50]. The formal reasoning capabilities of these models were once doubted, but Liu et al. [39] demonstrated the ability of Transformer models [63] to learn shortcuts to automata. Of particular interest is the generalization ability of such models to unseen, out-of-distribution data [53], enhancing their appeal for logical reasoning [2].

Another application area is linear-time temporal logic (LTL), which is heavily utilized by the formal verification community [16, 5] for reasoning about how logical propositions change over time [47]. Through the use of temporal operators, LTL formulae can specify, for example, that a proposition p must hold at all time steps ($\mathbf{G}p$), or at least one time step ($\mathbf{F}p$). LTL formulae operate on traces, which describe how the propositions change over time.

Solving a given LTL formula involves finding a satisfying trace, and it proved essential for generating examples for system specifications in the literature. This field was dominated by the methods that use classical algorithms, such as `spot` [19] and `aalta` [38]. However, following the success of Transformer models on end-to-end symbolic integration [35], Hahn et al. [28] attacked the LTL solving problem using the same approach. Their capability to generalize to longer formulae is especially noteworthy, and it was made possible thanks to tree-positional encoding [55].

However, generalization to longer formula lengths is not the only concern. In partic-

ular, each LTL formula features a set of atomic propositions (henceforth APs), and it's desirable for the model to generalize to more APs. But the architecture of the model does not even accept new APs that are not seen during training, despite the fact that all APs represent *semantically equivalent* concepts while being *distinguishable* from each other. This situation arises in many other application areas, such as mathematical expressions and lambda calculus [1], where renaming the bound variables does not change the meaning. This phenomenon is described as *alpha-equivalence*. *Alpha-conversion* (or *alpha-renaming*) refers to the process of creating alpha-equivalent input-output pairs.

In this thesis, we propose a novel approach for representing interchangeable tokens in neural network models. To summarize, our method constructs some part of the token embeddings on-the-fly instead of learning all of them during training. The token embeddings for interchangeable tokens consist of two parts: a learnable part and a randomized part. The learnable part is shared across all interchangeable tokens, and the model must depend on the randomized part to differentiate these tokens. Thanks to the randomized component, our method can generate embeddings for arbitrarily many interchangeable tokens as needed during both training and inference, with the only practical limitation being the exponentially growing sampling set size for discrete random generation methods. We use the weight tying technique [48] to share the same token embeddings with the final projection matrix, which calculates the logits (i.e., next-token probabilities before softmax).

We use our embedding method in a Transformer encoder-decoder model and evaluate it on three tasks: copying with an extendable vocabulary, solving LTL formulae, and predicting assignments for propositional logic. As a baseline, we consider a simpler approach that uses alpha-renaming for data augmentation during training to expose the model to a larger vocabulary, which is also new in the literature to the best of our knowledge. Overall, our method demonstrates generalization capabilities to larger vocabulary sizes, and also combines well with positional encodings that exhibit length generalization. We also experiment with dataset perturbation to show that our method introduces a helpful inductive bias for alpha-equivalence. Finally, we present *alpha-covariance*, a metric for measuring robustness against alpha-conversions that is applicable to any domain where alpha-equivalence is relevant.

Overall, our contributions can be summarized as follows.

1. Identify the problem of generalizing to larger vocabularies in (formal) language modeling tasks, and define an experimental protocol to study this problem.
2. Propose alpha-covariance, a novel metric for measuring robustness against alpha-conversions, applicable to any domain with interchangeable tokens.
3. Introduce a dual-part embedding method for vocabulary generalization and improved alpha-covariance, with negligible computational overhead.
4. Verify the proposed method thoroughly on three tasks: copying with extendable vocabulary, solving LTL formulae, and predicting assignments for propositional logic.

1.1 Thesis Outline

The rest of this thesis is structured as follows:

- **Chapter 2: Background & Related Work** reviews the necessary formal logic foundations, including linear temporal logic (LTL) and propositional logic. It also introduces the concept of alpha-equivalence, and surveys relevant literature in logic processing, specification mining, and the intersection of formal reasoning and language modeling.

- **Chapter 3: Proposed Method** formalizes the problem of learning alpha-equivalence-invariant representations. It introduces a dual-component embedding strategy comprising a shared semantic embedding and a randomized component for token uniqueness. The chapter also defines the alpha-covariance metric and details the architecture's components including normalization, projection, and loss functions.

- **Chapter 4: Experiments** evaluates the proposed method across tasks that involve reasoning over symbolic sequences with extendable vocabularies. These include the copying task with extendable vocabulary, LTL formula solving, and propositional logic assignment prediction. Baseline comparisons, ablation

studies, and an analysis of computational performance are also provided, along with an evaluation against large language models.

- **Chapter 5: Discussion** discusses the current limitations of the proposed method, and concludes with a summary of contributions and potential future work directions.

- **Appendix A: LLM Prompts** documents the exact prompts used when interacting with large language models in experimental evaluations.

CHAPTER 2

BACKGROUND & RELATED WORK

This chapter provides essential preliminary information about formal logic, alpha-equivalence, and language modeling. It also examines the relevant literature, focusing on problems such as solving formulae, mining specifications, and translating from natural language. Particular attention is devoted to the intersection between formal methods and machine learning, as well as the problem of learning an extensible vocabulary.

2.1 Formal Logic Overview

This section explains the basics of linear temporal logic and propositional logic. Additional implementation-specific considerations (such as symbolic traces, prefix notation, tokenization) are also explained here.

2.1.1 Linear Temporal Logic

Linear Temporal Logic (LTL) extends conventional logic by introducing the ability to reason about the evolution of propositions over time [47]. The syntax of LTL, defined over a finite set of atomic propositions P, is given in Equation 2.1, where $\mathbf{T}$ represents *True*, $p \in P$ an atomic proposition (or shortly AP), $\neg$ the negation operator, $\wedge$ the conjunction operator, $\mathbf{X}$ and $\mathbf{U}$ the temporal operators *next* and *until* respectively.

$$\phi := \mathbf{T} \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathbf{X}\phi \mid \phi_1 \mathbf{U} \phi_2 \tag{2.1}$$

Specifically:

- $\mathbf{X}\phi$ holds at time t if and only if ϕ holds at the next time step, i.e., at time $t+1$.
- $\phi_1 \mathbf{U} \phi_2$ means that ϕ_2 must hold at some future time t_2, and ϕ_1 holds at every time step t from the current time t_1 up to but not necessarily including t_2.

For instance, the formula $\mathbf{XX}a$ specifies that a must hold at the third time step. Similarly, the formula $\mathbf{TU}a$ requires that a holds at some point in the future. Finally, as a more complex example, the formula $\mathbf{X}b \wedge a\mathbf{U}c$ asserts that b holds at the second time step, c holds at some future time, and a holds at all preceding time steps.

An LTL formula is evaluated over a *trace*, which represents a sequence of truth values for atomic propositions over time. In this work, as in DeepLTL [28], we consider *symbolic* traces of *infinite* length. These traces are expressed in what is known as a *lasso* form, denoted uv^ω, where u is a finite prefix, and v is a finite sequence that repeats indefinitely.

A symbolic trace represents all traces that satisfy the propositional formulae at the respective time steps. For example, the symbolic trace $a, a \wedge \neg b, (c)^\omega$ describes all traces in which a holds at the first two time steps, b does not hold at the second time step, and c holds at every step from the third onward. This symbolic trace satisfies the formulae $\mathbf{TU}c$ and $\mathbf{X}\neg b \wedge a\mathbf{U}c$, but it violates the formula $\mathbf{XX}b$ since b is not guaranteed to hold at the third time step. Symbolic traces, such as this one, can be underspecified, meaning that certain propositions (e.g., a and b) may take arbitrary values at some time steps.

The LTL solving problem involves identifying a symbolic trace in lasso form uv^ω that satisfies a given input formula ϕ. We approach this as an autoregressive language modeling task: given an LTL formula and a partially generated symbolic trace, the model predicts the probabilities for the next token in the trace.

For compatibility with the dataset from DeepLTL [28], both our traces and formulae are represented in Polish (prefix) notation, where operators precede their operands. For instance, $a \wedge b$ is written as `&ab`, which avoids the need for parentheses to resolve ambiguities.

As described earlier, we assume that traces are infinite and represented in lasso form uv^ω. Alongside atomic propositions, constants (`True:1` and `False:0`), and logical operators, we use special symbols in the notation: "`;`" is a position delimiter, and "`{`" and "`}`" enclose the repeating period v. For example, the string "`a;&ab;{b}`" represents the symbolic trace $a, a \wedge b, (b)^\omega$. Each character is represented by a separate token in the transformer model.

2.1.2 Propositional Logic

Unlike LTL (Appendix 2.1.1), propositional logic does not feature any temporal operators, but we include the derived operators for equivalence ($\leftrightarrow$) and exclusive or ($\oplus$) alongside the basic negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$). This leads to the syntax given in Equation 2.2, defined over a finite set of atomic propositions P where $p \in P$ an atomic proposition.

$$\phi := \mathbf{T} \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \leftrightarrow \phi_2 \mid \phi_1 \oplus \phi_2 \tag{2.2}$$

In assignment prediction problem for propositional logic, the goal is to determine a Boolean assignment for every atomic proposition $p \in P$ such that the given formula is satisfied. We allow the assignments to be partial, e.g., just as $a = 1, b = 1$ is a valid assignment for the formula $a \vee b$, so is $a = 1$, which allows b to take any value.

To encode the assignments for the neural network, an alternating sequence of atomic propositions and values is used. For example, `a1b0` represents the assignment $a = 1$ and $b = 0$. To verify the outputs of the neural network and to generate datasets, `pyaiger` was used [64].

2.2 Alpha-Equivalence

Alpha-equivalence is a foundational concept in formal language theory, including functional programming, logic systems (such as LTL), and lambda calculus [1]. Two expressions are said to be *alpha-equivalent* if they differ only in the names of their

bound variables. This means that renaming bound variables in a consistent way does not affect the semantic meaning of an expression.

In the lambda calculus, this idea is often introduced through expressions such as $\lambda x.\, x$ and $\lambda y.\, y$, which are considered alpha-equivalent because the choice of bound variable (x or y) is arbitrary as long as it is used consistently within the expression.

In the context of LTL, consider function definitions that operate over atomic propositions using logical and temporal operators. For example, consider the function $f(a, b) = a \wedge \neg b$, where a and b are atomic propositions. Example alpha-equivalent variants of this expression are given in Equation 2.3. Renaming the variables to x and y as in $f(x, y) = x \wedge \neg y$, or even swapping a and b yields an alpha-equivalent expression. The renaming of parameters does not alter the underlying logical structure or the truth conditions of the function, i.e., it has the same underlying meaning despite being syntactically different.

$$\begin{aligned} f(a, b) &= a \wedge \neg b \\ f(b, a) &= b \wedge \neg a \\ f(x, y) &= x \wedge \neg y \\ f(c, d) &= c \wedge \neg d \end{aligned} \tag{2.3}$$

It is important to keep in mind that alpha-equivalence is a general property of many formal systems. It arises not only in LTL but also in first-order logic, functional programming languages, type theory, and other domains where variable binding plays a central role. Alpha-equivalence simply signifies that the semantics of expressions remain invariant under consistent renaming of bound identifiers.

In this thesis, the concept of alpha-equivalence is defined more broadly with respect to the input and output of a machine learning model, treating the whole model as a function. When we capture the whole context like this, all variables effectively become bound variables. Because renaming a variable in both the input and the output is expected to preserve semantics, demonstrating alpha-equivalence. Further details will be covered in Section 3.1.

2.3 Formal Logic Literature

The primary application area of temporal logic is formal verification, for specifying requirements and verifying system behaviors [16, 5]. Due to their expressiveness and similarity to natural language, temporal logics have become popular as a specification formalism in various fields, e.g., dynamic systems [12], robotics–especially in motion planning [34, 56, 20, 59], and biology [11].

We will divide the works in this domain into three categories based on the problem they tackle. The common pattern in all of these problems is the proliferation of machine learning methods, reflecting the advancements in computer vision and natural language processing. The three problems are as follows:

1. Solving formulae (Section 2.3.1)
2. Specification mining (Section 2.3.2)
3. Natural language (Section 2.3.3)

2.3.1 Solving Formulae

In formal logic, solving formulae is a fundamental and challenging task. It typically involves determining whether a given formula is satisfiable, and if so, identifying a satisfying trace or assignment. For Linear Temporal Logic (LTL) in particular, this problem is known to be PSPACE-complete [57], making it computationally intensive as the size of the formula increases.

Traditionally, classical algorithms have been the primary tools for addressing this problem. Among these, `spot` [19] is one of the most widely used tools due to its efficiency and robustness. Another classical tool, `aalta` [38], offers an alternative approach but is generally considered less common and somewhat dated in comparison.

However, due to the intrinsic complexity of the problem, classical algorithms often face scalability limitations when dealing with large or highly complex formulae. As the state space grows exponentially, these methods can become impractical, both in

terms of computation time and memory usage. In response to these challenges, recent research has increasingly turned toward machine learning (ML) approaches. These methods aim to either complement or replace traditional techniques by leveraging data-driven models that can learn heuristics or approximations for solving LTL formulae more efficiently.

Until recently, deep learning techniques were widely regarded as insufficient for tackling complex tasks involving symbolic reasoning. This skepticism stemmed largely from the belief that deep neural networks lack the reliability and structure-awareness required for solving intricate logical problems. As a result, applications of deep learning in the domain of formal logic have typically been limited to narrow subtasks within broader reasoning frameworks. Examples include learning heuristics for solver guidance [36, 6, 54], or predicting individual steps within formal proofs [41, 24, 7, 30].

More recently, however, this assumption has been increasingly challenged. Studies have shown that modern neural architectures, particularly transformers [63], exhibit promising capabilities in symbolic domains. For instance, [35] showed that Transformers can perform symbolic integration with surprising accuracy, while [51] demonstrated that self-supervised training enables neural models to develop non-trivial mathematical reasoning skills. Furthermore, [14] revealed that sufficiently large language models can acquire basic arithmetic abilities, even when trained predominantly on natural language data.

Building on these advancements, DeepLTL [28] was introduced as a novel approach to solving LTL formulae using deep learning. In particular, DeepLTL employs a transformer encoder-decoder architecture to generate satisfying traces for given LTL specifications. One of its most significant achievements is the ability to solve formulae that classical solvers fail to handle within practical time limits.

A key factor behind DeepLTL's performance is its use of tree-positional encoding [55], which enables the model to capture the structural hierarchy of logical formulas more effectively. This design allows the model to generalize to longer inputs than those seen during training. While DeepLTL is primarily focused on LTL, further experiments on propositional logic suggest that its generalization capabilities extend beyond

temporal logic alone.

Despite these impressive results, generalization was only considered in terms of input formula length. To the best of our knowledge, the prior works in the literature have not explored the generalization to a larger atomic proposition vocabulary in this context, which is the focus of this thesis.

2.3.2 Specification Mining

Specification mining is another interesting problem in formal logic. Extracting temporal logic formulae, typically as LTL or its variants, from system traces is the basis of specification (requirement) mining. Formulae extraction has many applications such as detecting bugs, testing for regressions, generating new tests, and so on [10, 49, 67, 9, 65, 43]. The resulting temporal logic formulae can also be used for the purposes of interpretability since LTL formulae are easily understood by human experts [10].

The previous works in specification mining utilized a wide variety of methods, including template-based techniques [31], methods based on decision trees [13, 33] or automata [15], and many others [65, 8]. These existing methods for specification mining either depend on human expertise (as in template-based methods) or suffer from combinatorial explosion problems. In particular, the experiments by [25] show that exhaustive combinatorial algorithms [3] and SAT-based solvers [45, 23] exhibit slow runtime performance, especially as the problem size grows, rendering them infeasible for practical applications. Although [25] improved these baselines by devising clever optimizations that exploit the properties of LTL, exhaustive combinatorial search scales poorly since the specification mining problem is NP-hard [21].

The introduction of neural networks to this domain occurred through Signal Temporal Logic (STL), which is a variant of temporal logic that operates on continuous signals instead of propositions. Specifically, STLCG [37] presented a framework that defines computation graphs for the quantitative semantics of STL formulae, thereby enabling backpropagation through them. This work depends on the fact that robustness metric for STL can be defined in a differentiable way. Unlike LTL, the satisfaction of a STL

formula is not binary; a robustness metric which denotes how well a signal fits a given formula can be calculated [61]. This enables numerical optimization methods that are not applicable for LTL.

Thanks to STLCG, the parameters of an STL formula can be optimized to satisfy the given signal(s). However, STLCG cannot generate the formula structures by itself, hence it's a template-based method. Similarly, [71] proposed wSTL-NN (weighted signal temporal logic neural network), which not only defines a differentiable computation graph for backpropagation, but also assigns learnable weights to subformulae. Although this method can eliminate undesired subformulae by reducing their weights through backpropagation, it's still dependent on templates.

2.3.3 Natural Language

A related challenge that has gained increasing attention in the temporal logic community is the problem of translating natural language statements to LTL formulae. Since temporal logic is a highly specialized formalism, most users, even domain experts, struggle to express specifications directly in LTL without support. Although these experts possess deep domain knowledge, they typically communicate their requirements in natural language, which is imprecise from a formal logic perspective despite being easy to understand. Consequently, a growing body of research concentrated on developing methods to bridge this gap by translating natural language descriptions into precise LTL formulae.

Reflecting the rapid progress in language modeling, early efforts to translate natural language into LTL formulae adopted recurrent neural network (RNN) encoder-decoder architectures [26, 46]. These initial models laid the groundwork but were limited in their ability to generalize to unseen inputs. Subsequent work shifted toward leveraging pre-trained large language models to improve performance and robustness, a trend pioneered by [40].

More recent approaches have introduced interactive frameworks to handle the inherent ambiguity of natural language requirements. For example, nl2spec [17] proposes a method where large language models assist users in incrementally refining

translations by aligning natural language fragments with corresponding subformulae. This iterative process simplifies correction and improves usability in practical specification tasks. Another domain-specific application is Cook2LTL [42], which translates cooking recipes into LTL formulae to facilitate robotic planning. By combining pretrained language models with a dynamic caching mechanism for action grounding, Cook2LTL efficiently generates temporally structured plans from real-world recipes, significantly reducing inference latency and cost during execution in a simulated kitchen environment.

2.4 Language Models

The autoregressive language modeling or sequence modeling in a broader sense, whose goal is to predict the next token given the past tokens, was revolutionized by the transformer architecture [63], replacing the step-by-step processing of recurrent neural networks (RNNs) with a parallelizable attention mechanism.

At the core of a transformer model lies the attention mechanism, which computes three vectors—query, key, and value—from input embeddings. This mechanism allows the model to measure the relative importance of different tokens, thereby capturing long-range dependencies coherently. In self-attention, these vectors come from the same sequence, while in cross-attention, key and value vectors come from a different sequence, as in encoder-decoder setups. The transformer consists of an encoder and a decoder, both of which feature self-attention and feed-forward layers. The primary difference is that the decoder adds cross-attention to integrate the encoder's output.

Unlike RNNs, attention mechanism does not process the input sequentially, which, despite the parallelization advantage, renders the model incapable of sensing the token order. As a result, positional encodings must be added to input embeddings to provide the sequential order information. During training, attention masking ensures causality in predictions, preventing future tokens from being considered when predicting the next one. This design enables transformers to efficiently handle complex tasks like machine translation and text generation.

2.4.1 Language Modeling and Formal Reasoning

The transformer architecture [63], now ubiquitous in modern deep learning, was initially proposed as a generative model to translate between natural languages autoregressively. This led to many successful attempts to frame formal reasoning tasks as language modeling problems, such as symbolic integration [35], symbolic regression [32, 62], LTL solving [28], and many more. Further developments shifted the field towards large language models (LLMs), e.g., by prompting a model pre-trained on a gigantic scale [22], by enhancing the prompt with retrieved references for proof generation [69, 72], by training an LLM on a specialized dataset for mathematics [4]. However, the reasoning abilities of LLMs were questioned by [60], who showed LLMs struggle with symbolic reasoning when semantics are decoupled, and by others [70].

2.4.2 Extensible Vocabulary

Efforts to create an extensible vocabulary for neural networks are scarce in the broader machine learning community, let alone the formal reasoning literature. Morazzoni et al. [44] exploited dictionary definitions to create extensible word embeddings. Wei et al. [68] proposed a framework for sign language recognition that allows vocabulary extensions by using a component based approach. In particular, the method identifies common components such as hand orientation, trajectory, axis, rotation, and shape, thereby enabling flexible sign gesture recognition. These studies depend on either external information (dictionary definitions) or properties specific to an application area (components of hand gesture); they do not attempt to design an extensible vocabulary for interchangeable tokens, which has been neglected by the literature alongside the concept of alpha-equivalence. Despite these attempts in other application areas, the formal reasoning literature has neglected the concept of extensible vocabulary and alpha-equivalence.

CHAPTER 3

PROPOSED METHOD

The proposed method builds on DeepLTL [28], using the same transformer encoder-decoder approach in an autoregressive language modeling setting to tackle formal logic problems, namely generating satisfying traces for LTL formulae and assignment prediction for propositional logic. The overall approach is visualized in Figure 3.1. The details of infix-prefix conversion is explained in Sections 2.1.1 and 2.1.2 for LTL and propositional logic tasks, respectively.

For simplicity, Figure 3.1 considers a simple sampling algorithm without search, such as greedy sampling (choosing the most probable token in each time step). In practice, however, the beam search algorithm is utilized, especially on logic tasks. Instead of keeping a single generated sequence, beam search keeps the top k sequences based on their probability, where k is the beam size. In the next time step, all of these sequences are expanded, and the process repeats until completion. Beam seach algorithm is illustrated in Figure 3.2, assuming a beam size of 2. Although probabilistic sampling is widely used in LLMs, beam search performs better in formal logic, as the experiments in DeepLTL [28] demonstrates.

The proposed method enhances the prior work with consideration for alpha-equivalence. The primary consequence of this extension is resilience against alpha-conversions, i.e., variable renaming. Another important advancement is the generalization capability across interchangeable tokens, i.e., tokens subject to alpha-equivalence, such as variable names. The formal problem definition is given in Section 3.1.

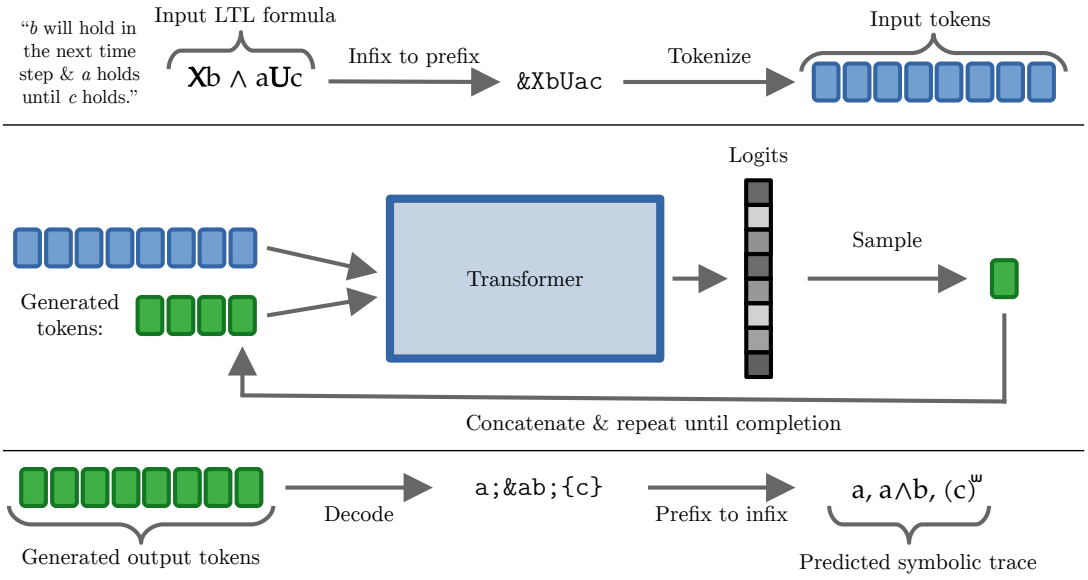


Figure 3.1: Using a transformer model to generate a trace for a given LTL formula.

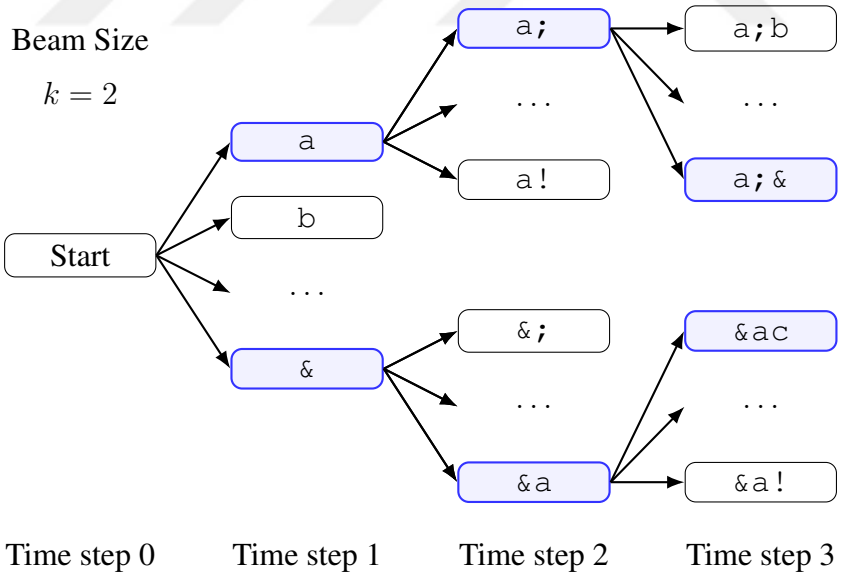


Figure 3.2: Beam search with beam size = 2. At each time step, only the top 2 sequences are kept for expansion.

3.1 Problem Definition

In language modeling, the goal is to predict the next token in the output sequence given the input and the past output, as explained in Section 2.4. Let $\mathbb{V}$ denote the set of all unique tokens, i.e., the vocabulary of a language modeling problem. We use $\mathbb{V}^*$ to denote the set of all finite sequences of tokens (strings) from $\mathbb{V}$. We assume that $\mathbb{V}_i$ is the set of interchangeable tokens and $\mathbb{V}_n = \mathbb{V} \backslash \mathbb{V}_i$ is the set of non-interchangeable tokens. The core idea behind alpha-equivalence is that renaming interchangeable tokens between each other in both input and output preserves meaning (Section 2.2). Let $f \colon \mathbb{V} \to \mathbb{V}$ be a bijection such that $f(x) = x$ for all $x \in \mathbb{V}_n$, i.e., f arbitrarily renames the interchangeable tokens between each other in one-to-one correspondence and preserves the rest of the tokens. We apply f to each token in a given pair of input $\boldsymbol{a} \in \mathbb{V}^*$ and output $\boldsymbol{b} \in \mathbb{V}^*$ strings, obtaining $\boldsymbol{a'} = (f(a_1), f(a_2), \ldots)$ and $\boldsymbol{b'} = (f(b_1), f(b_2), \ldots)$. We call this operation *alpha-conversion* or *alpha-renaming*. The set of interchangeable tokens $\mathbb{V}_i$ must be defined such that $\boldsymbol{a'}$ and $\boldsymbol{b'}$ form a valid input-output pair semantically equivalent to $(\boldsymbol{a}, \boldsymbol{b})$ for all possible f.

Our task is to design an embedding method that—alongside being resilient to alpha-renaming by construction—can support a new vocabulary $\mathbb{V}' = \mathbb{V}'_i \cup \mathbb{V}_n$ where $\mathbb{V}_i \subset \mathbb{V}'_i$ after training on $\mathbb{V}$. In other words, the model should be able to operate on a larger vocabulary than the one seen during training, as long as the newly introduced tokens belong to the class of interchangeable tokens. Although we don't impose any restrictions about the size of $\mathbb{V}'$ in this problem definition, the maximum size of $\mathbb{V}'$ in practice may change as a function of the number of embedding dimensions. Thus, while setting the hyperparameters, the expected size of $\mathbb{V}'$ must be considered.

Example. In the LTL solving problem (Section 2.1.1), the set of non-interchangeable tokens $\mathbb{V}_n$ includes the operators, constants, delimiter tokens ("`;`", "`{`", "`}`"), and any special tokens such as the end token. The set of interchangeable tokens equals to the set of atomic propositions (APs): $\mathbb{V}_i = P$. Assuming $P = \{\texttt{a}, \texttt{b}\}$, the formula-trace pair ("`&aXb`", "`a;b;{1}`") is alpha-equivalent to ("`&bXa`", "`b;a;{1}`"). Further, assume that the augmented set of interchangeable tokens is $\mathbb{V}'_i = P' = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$. Now, the aforementioned pair can also be equivalently represented as ("`&cXd`", "`c;d;{1}`"). The augmented vocabulary allows the expression of

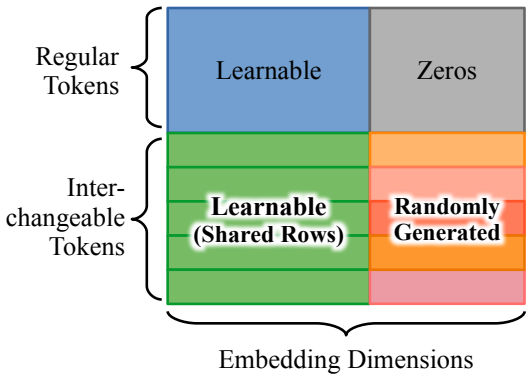


Figure 3.3: Visual structure of the embedding matrix in the proposed method.

formula-trace pairs that feature up to 4 APs instead of 2. For example, ("`&&abX&cd`", "`&ab;&cd;{1}`") cannot be expressed using $P = \{\texttt{a}, \texttt{b}\}$. Our goal is to create a model that can handle such inputs despite being trained on the limited vocabulary $\mathbb{V} = \mathbb{V}_n \cup P$.

3.2 Embedding Matrix

To address the problem of learning semantically equivalent but distinguishable (alpha-equivalent) tokens, our method employs two ideas: sharing some part of the embeddings between such tokens to convey their semantic equivalence; and assigning a unique randomly-generated vector to the rest of the embedding for each interchangeable token, allowing the model to distinguish between them. The number of shared and randomly-generated dimensions are denoted by d_α and d_β respectively. The sum of these two yields the total number of embedding dimensions in the model, denoted by $d_{\text{model}} = d_\alpha + d_\beta$. For non-interchangeable tokens, d_α dimensions contain separate learnable parameters and d_β dimensions are set to 0. The structure of the embedding matrix is visualized in Figure 3.3.

3.2.1 Construction of the Embedding Matrix

For a vocabulary with n non-interchangeable tokens and m interchangeable tokens, $\boldsymbol{L} \in \mathbb{R}^{n \times d_\alpha}$ represents the matrix of learnable embeddings for non-interchangeable

tokens, $\boldsymbol{\alpha} \in \mathbb{R}^{1\times d_\alpha}$ the shared learnable embedding for interchangeable tokens, and $\boldsymbol{\beta}_i \in \mathbb{R}^{1\times d_\beta}$ the randomly-generated embedding for the ith interchangeable token where $1 \leq i \leq m$. Note that $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}_i$ are row vectors. A zero matrix of size $i \times j$ is represented by $\mathbf{0}^{i,j}$. In addition, we define two row-based L2 normalization functions $f_{bn}(\boldsymbol{X})$ and $f_{fn}(\boldsymbol{X})$ that divide each row $\boldsymbol{X}_{i,:}$ by its L2 norm $\|\boldsymbol{X}_{i,:}\|$. These two functions are identical but can be disabled independently from each other, hence the separation. Finally, the overall structure of the embedding matrix $\boldsymbol{U}$ is shown in Equation 3.1. In this construction, the interchangeable tokens are assumed to come after the non-interchangeable tokens. Note that it's also possible to implement multiple sets of different interchangeable tokens via a trivial extension.

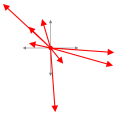
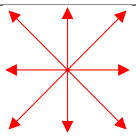
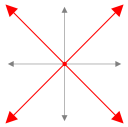
$$\boldsymbol{U} = f_{fn}\left(\begin{bmatrix} f_{bn}(\boldsymbol{L}) & \mathbf{0}^{n,d_\beta} \\ f_{bn}(\boldsymbol{\alpha}) & f_{bn}(\boldsymbol{\beta}_1) \\ f_{bn}(\boldsymbol{\alpha}) & f_{bn}(\boldsymbol{\beta}_2) \\ & \vdots \\ f_{bn}(\boldsymbol{\alpha}) & f_{bn}(\boldsymbol{\beta}_m) \end{bmatrix}\right) \tag{3.1}$$

During training, the embedding matrix must be reconstructed in each forward pass with resampled random vectors $\boldsymbol{\beta}_1$ to $\boldsymbol{\beta}_m$. Resampling $\boldsymbol{\beta}_i$ for $1 \leq i \leq m$ during training prevents the model from adapting to the idiosyncracies of a particular random generation and forces it to distinguish between interchangeable tokens regardless of the contents of $\boldsymbol{\beta}_i$. During inference, it's created once at the start and remains the same since the autoregressive generation involves multiple forward passes on the same input.

3.2.2 Normalization

There are several concerns that warrant the heavy use of normalization while constructing $\boldsymbol{U}$, as seen in Equation 3.1. Firstly, d_α dimensions and d_β dimensions should not overwhelm each other in terms of magnitude. Normalizing $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}_i$ separately addresses this issue. The magnitude of the concatenated embedding is another concern, which is handled by the final normalization. The normalization of $\boldsymbol{L}$ is redundant (since the final normalization does the same operation after the concatenation

Table 3.1: Comparison of random vector generation methods.

Method	**Normal Distribution**	**Neighboring Points**	**Hypercube Vertices**
Formula	$\mathrm{a}_i \sim \mathcal{N}(0,1)$	$\mathrm{a}_i \in \{-1,0,1\}$ $\|\mathbf{a}\| \neq 0$	$\mathrm{a}_i \in \{-1,1\}$
Size for n-dims	Continuous	$3^n - 1$	2^n
Sample Visualization			

with zeros) but kept in Equation 3.1 for readability.

3.3 Random Embedding Generation

This section will explain how the distinguishing part of the interchangeable token embeddings, $\boldsymbol{\beta}_i, 1 \leq i \leq m$, are created. To this end, we developed 3 methods to generate random vectors. Table 3.1 provides a summary at a glance. The first method simply samples the standard normal distribution for each dimension. The second one uses the neighboring grid points around the origin, which correspond to the 8 directions in 2D. For each interchangeable token, a unique vector in this set is sampled. The last method is similar, but its set consists of the vertices of a hypercube centered around the origin, i.e., diagonal direction vectors.

3.3.1 Uniqueness Constraint

In the normal distribution method, we don't have any additional constraints to ensure distinguishability between vectors. However, in other two methods, we need to make sure that each interchangeable token gets assigned to a unique vector since the sampling set is finite.

A naive solution involves generating the sampling set in its entirety and then sampling the desired number of points from it. This process is described in Algorithm 1. Sampling from hypercube vertices is similar: choices variable is set to $[-1, 1]$ instead of $[-1, 0, 1]$ and the redundant center point check is removed. Despite its simplicity,

this method faces scalability issues due to heavy memory usage. As the dimensions increase, generating the whole sampling set becomes exponentially more expensive.

Algorithm 1 Naive Neighboring Points Sampling in n-Dimensional Space

Input: number of points k, number of dimensions d

`choices` $\leftarrow [-1, 0, 1]$

`offsets` $\leftarrow$ cartesian product of d copies of `choices`

Remove vectors from `offsets` where all elements are 0 {Exclude center point}

`sampled_neighbors` $\leftarrow$ randomly sample k vectors from `offsets`

Return stack of `sampled_neighbors` as a tensor

To achieve this quickly and space-efficiently, we define a mapping from integers to possible vectors. The unique vectors are generated by sampling m unique random integers (which can be calculated efficiently using the reservoir sampling technique), and then using the defined mapping to convert these integers to the vectors. This strategy avoids materializing the whole set of possible vectors. In the hypercube vertices method, we map the binary digits of an integer in $[0, 2^{d_\beta})$ to $\{-1, 1\}$. Algorithm 2 describes this efficient sampling method.

Algorithm 2 Sampling Hypercube Vertices Efficiently in n-Dimensional Space

Input: number of points k, number of dimensions d

Let $N \leftarrow 2^d$ {Total number of hypercube vertices}

`indices` $\leftarrow$ randomly sample k integers from $[0, N)$

`vectors` $\leftarrow$ empty list

for each i in `indices` **do**

$b \leftarrow$ binary representation of i with d bits (zero-padded)

$$v \leftarrow \text{vector where } v_j \leftarrow \begin{cases} -1 & \text{if } b_j = 1 \\ 1 & \text{otherwise} \end{cases}$$

Append v to `vectors`

end for

Return stack of `vectors` as a tensor

Although “Neighboring Points" is simply the ternary version of the same idea, avoiding the zero vector requires special care. The zero vector maps to the integer $i_z = (3^{d_\beta} - 1)/2$. Therefore, we define our domain as the integers in $[0, 3^{d_\beta} - 1)$ and

add 1 to the integer i before converting it if $i \geq i_z$. This approach is described in Algorithm 3.

Algorithm 3 Sampling Neighboring Points Efficiently in n-Dimensional Space

Input: number of points k, number of dimensions d

Let $N \leftarrow 3^d$ {Total number of ternary neighbor vectors}

Let $c \leftarrow (N-1)/2$ {Index of the central point (all zeros)}

`indices` $\leftarrow$ randomly sample k integers from $[0, N-1)$

`vectors` $\leftarrow$ empty list

for each i in `indices` **do**

$$i' \leftarrow \begin{cases} i+1 & \text{if } i \geq c \\ i & \text{otherwise} \end{cases} \quad \{\text{Remove the center point}\}$$

$b \leftarrow$ ternary representation of i' with d digits (zero-padded)

$v \leftarrow$ vector where $v_j \leftarrow \texttt{int}(b_j) - 1$

Append v to `vectors`

end for

Return stack of `vectors` as a tensor

Integer mapping approach for generating unique vectors works well for up to 32 dimensions, after which the limits of integer representation become an issue for reservoir sampling. Therefore, in such cases, we simply disable the uniqueness check because the exponentially growing size of the sampling set renders the probability of drawing the same sample negligible.

3.4 Projection

In this section, we focus on the final layer of the transformer model, which projects the feature vectors to logit vectors, and examine the relevant modifications made by the proposed method.

3.4.1 Weight Tying

In a traditional language modeling setting, since both the embedding and projection matrices are entirely composed of learnable parameters, it's not necessary to share them, even though there are many advantages of weight tying [48]. However, we construct the embedding matrix manually in our method, which makes weight tying a requirement. Furthermore, since we perform our experiments on an encoder-decoder architecture in this thesis, we utilize a three-way weight tying approach, whereby the embedding matrices of encoder and decoder are tied in addition to the final projection matrix. Three-way weight tying is particularly appropriate for the LTL solving task since many tokens are shared between the LTL formulae and traces.

3.4.2 Feature Normalization

Given the output of the last layer before the final projection $\boldsymbol{v}$ (henceforth called feature vector), instead of directly applying the final projection as in $\boldsymbol{U}\boldsymbol{v}$, we apply L2 normalization to the feature vector $\boldsymbol{v}$ before passing it through the final projection: $\boldsymbol{U}f_{fn}(\boldsymbol{v})$. This matrix multiplication constitutes taking a dot product with each row. Since $\boldsymbol{a} \cdot \boldsymbol{b} = \|\boldsymbol{a}\|\|\boldsymbol{b}\|cos(\theta)$ where θ is the angle between $\boldsymbol{a}$ and $\boldsymbol{b}$, normalizing both the embeddings and the feature vector leaves only the cosine term to determine the logits. This forces the model to distinguish between tokens based solely on the directions, which may improve the gradient flow.

3.4.3 Cosine Loss

If we normalize both the embeddings and the feature vector, the only thing that determines each logit is the cosine of the angle between the feature vector and the embedding. Applying the softmax loss to such logits is known as cosine loss in the literature. Although cosine-based loss functions were successful in face recognition [52, 66], it proved sensitive to hyperparameter settings in these losses. To avoid this problem, we use AdaCos loss function [73] that scales the logits adaptively throughout training.

Despite the attractiveness of AdaCos in this context, it is not directly applicable in

a language modeling setting due to the additional sequence length dimension, and no prior work explored this application to the best of our knowledge. To overcome this, we modify the AdaCos loss function as follows: First, we combine the batch and length dimensions while ignoring the padding tokens, effectively treating both dimensions as batch dimensions. However, since this change greatly increases the number of batch dimensions, it can lead to numerical issues, even with the log-sum-exp trick. Therefore, we clip the scale value calculated by AdaCos to a maximum of 100 to avoid numerical issues. This loss formulation can also be used with conventional embeddings, as we do in our experiments.

Our variant of the AdaCos loss function is shown in Algorithm 4.

Algorithm 4 AdaCos variant for sequence modeling

Input: cosine similarity logits C, labels Y, scale s

Assert $\max(C) \leq 1.0$ {Each logit must be cosine of an angle}

Reshape C to 2D and flatten Y {Eliminate sequence dim}

Filter out elements in C and Y where $Y < 0$ {Remove ignored tokens such as padding tokens}

With disabled gradients:

$s \leftarrow$ adacos_scale_update(C, Y, s) {Update scale}

$s \leftarrow \min(s, 100)$ {Clamp scale}

Return $(\text{CrossEntropyLoss}(C \times s, Y), s)$

3.5 Alpha-Covariance

Given a vocabulary of n interchangeable tokens and an input-output pair containing k interchangeable tokens, it's possible to write ${}^nP_k = n!/(n-k)!$ alpha-equivalent pairs. In particular, for the first interchangeable token, we could choose one from n different interchangeable tokens from the full vocabulary. For the second interchangeable token, however, we would have only $n-1$ options as it must be different from the first. Continuing this pattern, we would have $n(n-1)\ldots(n-k+1) = n!/(n-k)! = {}^nP_k$ possibilities.

Since all of these alpha-equivalent pairs are semantically equivalent, we expect the

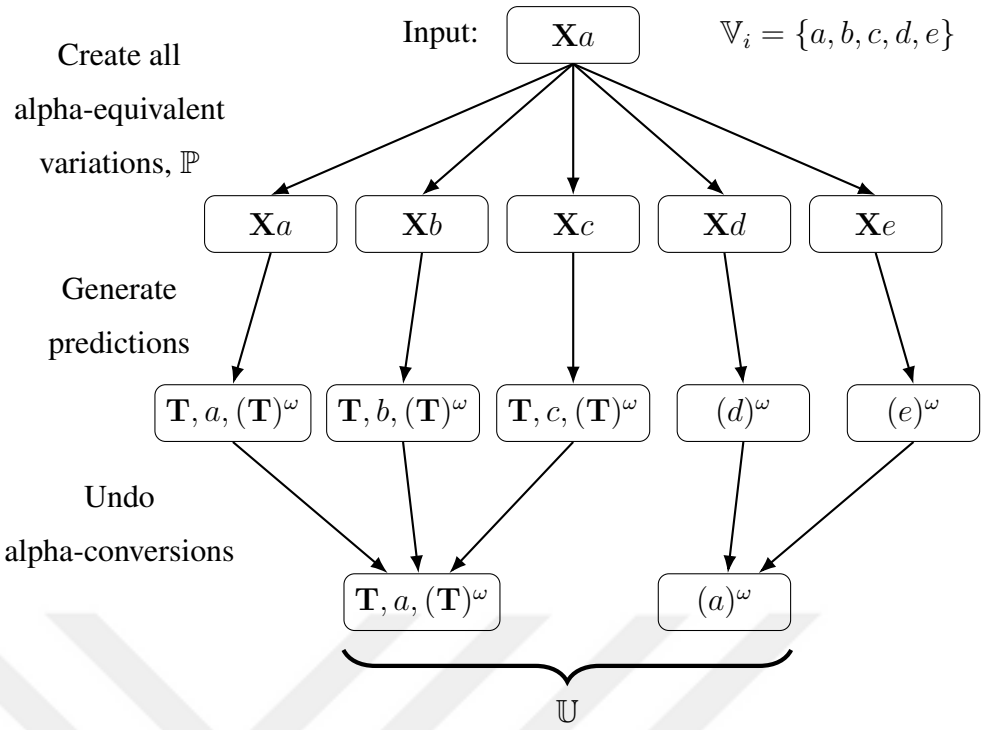


Figure 3.4: Computation of the set $\mathbb{U}$ for alpha-covariance calculation.

model's predictions to be the same after undoing the alpha-conversions for all of them. To the best of our knowledge, there is no metric to quantify this in the literature. Thus, we develop and present a new metric called *alpha-covariance*.

Let $(\boldsymbol{x}, \boldsymbol{y})$ be an input-output pair for the model, and let $\mathbb{P} = \{(\boldsymbol{x}^1, \boldsymbol{y}^1), \ldots, (\boldsymbol{x}^n, \boldsymbol{y}^n)\}$ be n input-output pairs alpha-equivalent to $(\boldsymbol{x}, \boldsymbol{y})$. We define α_i as the alpha-conversion function for the ith input-output pair such that $\alpha_i(\boldsymbol{x}) = \boldsymbol{x}^i$ and $\alpha_i(\boldsymbol{y}) = \boldsymbol{y}^i$. To compute the alpha-covariance of a model with respect to $\mathbb{P}$, we generate predictions for each input in $\mathbb{P}$, obtaining the prediction $\hat{\boldsymbol{y}}^i$ for each x^i. We define a set that contains the predictions with alpha-conversion undone: $\mathbb{U} = \{\alpha_i^{-1}(\hat{\boldsymbol{y}}^i) \mid 1 \leq i \leq n\}$. The overall process of computing $\mathbb{U}$ is visualized through an example in Figure 3.4.

Note that if we defined $\mathbb{U}$ for the ground truth outputs in $\mathbb{P}$, we would get $\{\boldsymbol{y}\}$ since $\alpha_i^{-1}(\boldsymbol{y}^i) = \boldsymbol{y}$ holds for each $\boldsymbol{y}^i$ by definition. The model's sensitivity to alpha-conversions could be quantified by simply $|\mathbb{U}|$, but this value may be hard to interpret since it depends on $|\mathbb{P}|$. To normalize this value intuitively, we define the alpha-

covariance of a model with respect to $\mathbb{P}$ as in Equation 3.2.

$$1 - \frac{|\mathbb{U}| - 1}{|\mathbb{P}| - 1} \tag{3.2}$$

Intuitively, when alpha-covariance is 1, none of the alpha-conversions in $\mathbb{P}$ affect the model. An alpha-covariance of 0 indicates that $|\mathbb{U}| = |\mathbb{P}|$, i.e., the model's prediction for each alpha-equivalent pair is unique after undoing the alpha-conversion. This is unwanted because alpha-conversions should not change the semantic meaning.

Thanks to the embedding randomization in our method, an alpha-conversion does not necessarily change the embeddings, and conversely, there are multiple ways to embed the same input due to randomness. Since our method is invariant to alpha-conversions by construction (as the discerning part of our embeddings depends solely on randomness, not specific symbols), measuring the alpha-covariance of our method amounts to measuring the model's robustness against the differences in random embeddings.

CHAPTER 4

EXPERIMENTS

4.1 Experimental Setup

We use a transformer encoder-decoder architecture in all experiments. We always use the same embedding size in both encoder and decoder due to weight tying. We use the RoPE [58] as the positional encoding method in the decoder. In the encoder, we use tree-positional encoding if applicable (logic tasks), RoPE otherwise (copying task). The hyperparameter settings are given in Table 4.1 in Section 4.1.2.

4.1.1 Baselines

We train three types of baseline models with traditional embeddings: the first one on the original dataset, the second one on a dataset with the same parameters but using a larger vocabulary size, and the third one on the original dataset but using a data augmentation strategy. Specifically, for the third baseline, the number of interchangeable token embeddings matches that of the test set, and we apply random alpha-renaming at each forward pass during training. This ensures that the model is exposed to all tokens in the test set, but the number of unique interchangeable tokens the model sees in each sample remains limited as in the training set. Note that this is an internal baseline that doesn't exist in the literature to the best of our knowledge. Please keep in mind that the first baseline cannot handle inputs with larger vocabularies.

Table 4.1: Hyperparameter choices.

(a) Model architecture hyperparameters.

Experiment	Embedding	Layers	Heads	FC size
Copy (Sections 4.2.2 and 4.2.3)	64	2	4	64
Copy (Section 4.2.6)	128	6	8	128
LTL (Section 4.3)	128	8	8	1024
Propositional Logic (Section 4.4)	132	6	6	512

(b) Training hyperparameters.

Experiment	Batch Size	Train Steps
Copy (Sections 4.2.2 and 4.2.3)	512	20K
Copy (Section 4.2.6)	512	20K
LTL (Section 4.3)	768	52K
Propositional Logic (Section 4.4)	1024	50K

4.1.2 Hyperparameters

The constant hyperparameter choices for all experiments are given in Table 4.1. These hyperparameters are kept constant within an experiment. The hyperparameters for the logic tasks are taken from DeepLTL [28]. For the LTL task, we used the same hyperparameters. On the other hand, for the propositional logic task, we had to make some changes to adapt them to our updated architecture. Firstly, since we utilize weight sharing, we cannot separate the embedding dimensions of encoder and decoder. As a result, instead of having an embedding dimension of 128 for the encoder and 64 for the decoder, we use 128 for both. However, since there are 6 attention heads, we round it up to 132.

4.2 Copying with Extendable Vocabulary

We introduce a new toy problem designed to evaluate the vocabulary generalization capabilities of our embedding method. We create various training datasets that con-

tain 10 million random strings with a limited vocabulary size. A string is given as input, and the model is expected to produce the input string exactly via autoregressive generation. This embodies a helpful toy problem for our method because all tokens are interchangeable, barring the special tokens (start/end). In these experiments, we expect the model to generalize to larger vocabulary sizes unseen during training.

Using edit distance as our evaluation metric, we first assess the vocabulary generalization capabilities (Section 4.2.2). Since our method excels in this task, we then explore generalization in both vocabulary size and string length (Section 4.2.3), performing a hyperparameter search over the settings of our embedding method (Section 4.2.4). Finally, we scale up the vocabulary size and the string lengths to evaluate our method (Section 4.2.6). Our method exhibits perfect performance in the out-of-distribution domain as shown in Figure 4.1. We also examine our method's sensitivity to randomness in embeddings (Section 4.2.5), and propose using the random embedding with median cross entropy loss as a proxy for average performance.

4.2.1 Evaluation method

We generate the predictions using greedy sampling in the copying task. We use the edit distance between the prediction and the ground truth as our evaluation metric. To generate the evaluation datasets (validation and test splits), we create 100 samples for each possible combination of unique character count and string length, starting from a minimum of 3. Consequently, the total evaluation dataset is arranged in a matrix in which the rows represent unique character count in the string and the columns represent the string length. This matrix is upper triangular since the unique character count cannot exceed the string length. For random embeddings, we repeat the evaluation 10 times and report the average. To evaluate up to the string length of 30 in this setup, $10 \times 100 \times 406 = 406000$ predictions are required, where 406 is the number of upper triangular elements in a 28×28 matrix. To minimize the impact of random factors, we train each model three times and report the results only for the best.

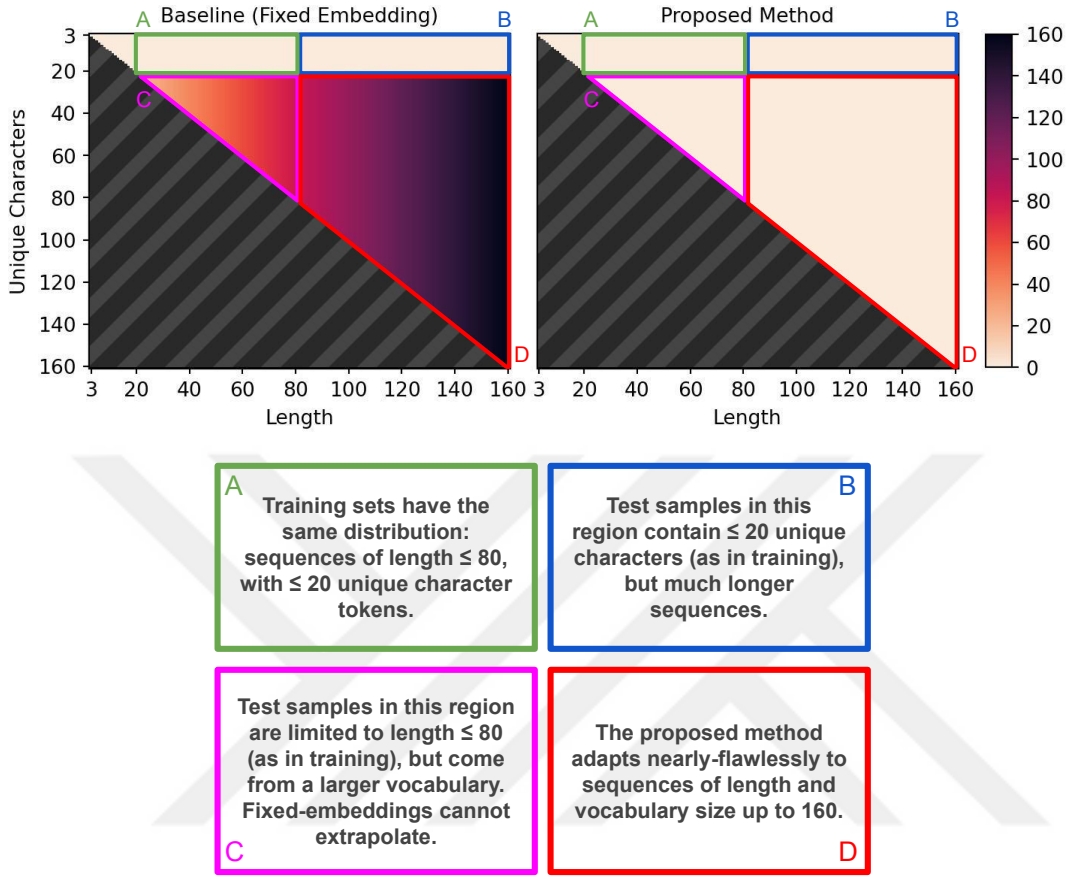


Figure 4.1: Two annotated heatmaps visualizing the test-set edit distance between prediction and ground truth in copying task with extendable vocabulary. Both heatmaps share the same y-axis. The green box represents the number of unique characters (y-axis) and the maximum length (x-axis) in the training dataset. Each point shows the average test error, except the lower triangular part of each heatmap (gray hatch pattern) corresponding to the impossible combinations of length and unique character counts. The traditional approach (left), using ubiquitously utilized fixed (learned) token embeddings, cannot extrapolate to vocabulary expansions. The proposed method (right) enables generalization to larger vocabulary sizes at longer sequence lengths, compared to what is observed during training.

4.2.2 Generalization to larger vocabularies

We create a dataset consisting of 10 million strings whose lengths vary between 3 and 30 with at most 5 unique characters. We evaluate the models on strings up to length 30 with at most 30 unique characters. Out of 27 models we trained with dual-part embeddings, 20 of them achieve an average edit distance of 0.0, i.e., no error. The worst model's average edit distance is 1.0. For comparison, an output sequence of length 30 can have a maximum edit distance of 30.

4.2.3 Generalization to larger vocabularies and lengths

We create a dataset consisting of 10 million strings whose lengths vary between 5 and 10 with at most 5 unique characters. We evaluate on the same validation set as before, expecting the model to generalize to both longer lengths and larger vocabulary sizes. In the next subsection, we perform a hyperparameter search over random embedding methods, d_β values, and whether f_{bn}, f_{fn}, AdaCos are enabled.

4.2.4 Hyperparameter Search

On the smaller copying task, we train multiple models that use different random embedding methods (Section 3.3) with different d_β values. While altering d_β, we keep the total number of embedding dimensions $d_\alpha + d_\beta$ constant. We train each model at least 3 times with different seeds and report the results for the best one in Tables 4.2 (proposed method) and 4.3 (baselines).

The results in Tables 4.2 and 4.3 exhibit high variance with no clear patterns that indicate which methods are better. Therefore, we perform an analysis based on correlation coefficients between these hyperparameters and the edit distance using the results from all 277 models we've trained (not including the baseline models). For this analysis, we assume that the value of Boolean properties (such as f_{bn}, f_{fn} and AdaCos) are 0 or 1. The correlation coefficients are as in Table 4.4.

Accordingly, the best random embedding method is "Neighboring Points" since it's the only one that correlates negatively with edit distance. The correlation observed

Table 4.2: Mean edit distance for various models using proposed method. The numbers in the header row represents d_β for each random embedding method. In the first column, enabled normalization features are listed. AC refers to AdaCos, which can only be enabled when f_{fn} is used.

(a) Random embedding method: Normal distribution

Enabled Features	d_β = 2	4	8	16	32
f_{bn} + f_{fn} + AC	13.6	5.4	4.6	8.1	8.1
f_{fn} + AC	7.6	13.1	4.6	2.2	5.2
f_{bn} + f_{fn}	13.7	10.6	8.3	3.8	11.8
f_{fn}	15.4	10.6	8.2	3.7	10.1
f_{bn}	10.6	16.6	11.8	6.9	8.2
-	16.5	11.6	12.6	12.5	9.0

(b) Random embedding method: Neighboring points

Enabled Features	d_β = 4	6	8	16	32
f_{bn} + f_{fn} + AC	1.9	13.0	2.2	1.0	2.1
f_{fn} + AC	8.7	11.5	2.8	2.9	2.2
f_{bn} + f_{fn}	11.9	5.7	3.7	7.4	8.3
f_{fn}	8.1	12.3	6.4	13.4	9.9
f_{bn}	5.8	3.0	0.6	7.8	14.3
-	12.5	3.7	9.5	5.9	13.5

(c) Random embedding method: Hypercube vertices

Enabled Features	d_β = 5	6	8	16	32
f_{bn} + f_{fn} + AC	2.8	0.4	7.5	8.4	3.9
f_{fn} + AC	0.5	3.7	3.2	4.2	4.1
f_{bn} + f_{fn}	2.2	13.1	21.5	19.4	20.9
f_{fn}	2.5	1.7	12.5	2.1	12.8
f_{bn}	12.8	13.8	19.4	22.9	11.6
-	12.7	9.6	8.6	15.9	16.6

Table 4.3: Mean edit distance for various baseline models. In the first column, enabled normalization features are listed. AC refers to AdaCos, which can only be enabled when f_{fn} is used. Note that f_{bn} is not applicable for baseline models. The results for the first type of baseline are omitted since it cannot generalize to larger vocabularies. The second baseline was trained on a dataset with a vocabulary size of 30. The third baseline uses the same limited vocabulary dataset like the proposed method, but uses alpha-renaming as data augmentation.

Enabled Features	Baseline 2nd Type	Baseline 3rd Type
f_{fn} + AC	6.1	1.9
f_{fn}	4.9	11.3
-	5.5	12.9

Table 4.4: The correlation coefficients between the hyperparameters and the edit distance across 277 models. First three columns are the random embedding methods as listed in Table 3.1, the fourth column is d_β, and the last three columns represent whether the given feature is enabled.

N.D.	N.P.	H.V.	d_β	f_{bn}	f_{fn}	AdaCos
0.02	-0.14	0.11	0.01	0.10	-0.29	-0.41

for d_β is negligible. Introducing f_{bn} increases the edit distance, but the statistical significance is not ideal (p-value 0.04). Both f_{fn} and AdaCos loss have a positive and statistically significant impact on edit distance, with p-values smaller than 10^{-6}.

We determine the best model for the proposed method and the baseline on the validation set, evaluate them on the test set and visualize the results in Figure 4.2. Since the baseline model cannot process larger vocabularies, we assume that the prediction is empty if the unique character count exceeds the training set's vocabulary, hence the edit distance equals length in that area. Our best model trained on limited length uses Hypercube Vertices with d_β set to 6 and f_{fn} + AdaCos enabled. It achieves a mean edit distance of 0.38 on the test set. The first baseline's mean edit distance is 0.51 (calculated up to 5 unique characters, only for this model). The second and third baselines' mean edit distances are 4.93 and 1.85 respectively. However, the significance of this difference is highly questionable, as these models exhibit high variance across different training runs.

4.2.5 Sensitivity to randomness in embeddings

We analyze the impact of the randomization that the proposed method performs on embeddings. The minimum, mean, and maximum edit distance (on test set) obtained by ten different embedding randomizations of the second model in Figure 4.2 are 0.25, 0.38, 0.55 respectively, with a sample standard deviation of 0.09. The pooled standard deviation of the edit distance across all 277 models evaluated on the validation set is 1.73. However, our best models are more resilient against randomness: this value is 0.74 for top 10% models.

To reduce the computational cost of evaluation in other experiments (All LTL experiments and Section 4.2.6), we generate 10 random embeddings, sort them by their cross entropy loss on the evaluated dataset, and use the median one. We find that this serves as a decent proxy for the average performance. Across the validation set evaluations of all 277 models, the percent difference in edit distance between this median method and the real mean is 1.4% on average (meaning that the result from the median method is worse), and 9.1% if we consider the absolute differences.

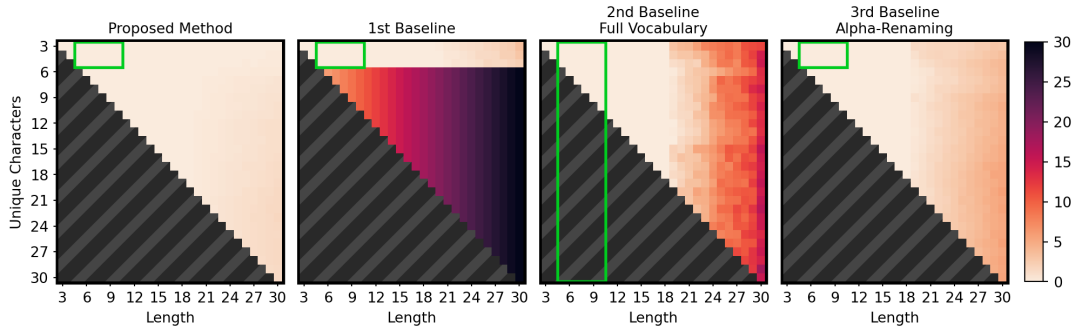


Figure 4.2: Edit distance heatmaps on test set. The first and second heatmaps are the proposed and baseline (first type) models respectively, trained on strings up to length 10 and a vocabulary size 5. The third heatmap is the second baseline, which uses a new training dataset with a larger vocabulary. The last heatmap is the third baseline that uses the same dataset as the proposed method but incorporates alpha-renaming in training. The difference between the last two baselines is that the alpha-renaming baseline is not exposed to more than 5 unique characters per sample. The lower triangular part of each heatmap (gray hatch pattern) represents the impossible combinations of length and unique character count. The green box represents the number of unique characters (y-axis) and the maximum length (x-axis) in the training dataset. Note that all heatmaps share the same y-axis.

4.2.6 Scaling up

We increase the length of the strings from 5-10 to 20-80, and vocabulary size from 5 to 20. We create the evaluation sets by generating 20 samples for each combination of unique character count and string length. The mean edit distance of our best model is 0.0. The heatmap is given in Figure 4.1. All baselines also attain perfect performance in this task on the vocabulary sizes they support. Therefore, only the first type of baseline is shown in Figure 4.1.

4.3 LTL Solving

In this section, we train models on the LTLRandom35 dataset from DeepLTL [28] and other synthetic datasets created with the same method. To evaluate the correctness of the generated formulae, we utilize `spot` framework version `2.11.6` [19]. We use tree-positional encoding [55] in the encoder and RoPE [58] in the decoder. We generate predictions using beam search with beam size = 3.

Baselines. We trained all of the baseline models from scratch. For the first type of baseline, we aimed to reproduce the results from Hahn et al. [28]. Hence, we used the best hyperparameters they reported (Section 4.1.2). Unlike Hahn et al. [28], we experimented with RoPE (in the decoder) and AdaCos, but did not observe a noteworthy improvement on the validation set.[1] After determining the best baseline model on the validation set, we evaluated it on the test split of LTLRandom35 and obtained a correct rate of 98.2% against the 98.5% reported by Hahn et al. [28].

4.3.1 Dataset Perturbations

To demonstrate that our method creates a helpful inductive bias, we created a perturbed version of the LTLRandom35 dataset by renaming the APs such that the order of the first AP appearances in the trace is always the same. As the empirical evidence in Table 4.5 confirms, both our method and the alpha-renaming baseline are naturally

[1] Using RoPE in the decoder increased the ratio of correct predictions from 97.8% to 98.0% on the validation set. Introducing AdaCos in addition to RoPE increased this value to 98.2%.

Table 4.5: Evaluation of the baselines, our method, and Llama 3.2 on the LTLRandom35 dataset. The alpha-renaming baseline was trained using 5 AP embeddings since vocabulary generalization is not evaluated here. First two columns denote the training dataset and the model. Next two columns indicate the ratio of the correct predictions and exact matches on 99,989 test set samples as evaluated by `spot`. Last three columns display mean alpha-covariance values for varying atomic proposition (AP) counts, evaluated on all alpha-equivalent variants of 1000 test samples. The results indicate that our method induces a robust inductive bias for alpha-equivalence.

Training		**Evaluation**		**Alpha-Covariance**		
Dataset	**Model**	**Correct**	**Exact**	**3 AP**	**4 AP**	**5 AP**
Normal	Baseline	98.23%	83.23%	96.87%	95.86%	91.80%
Perturbed	Baseline	34.13%	12.12%	64.93%	57.99%	40.91%
Perturbed	Alpha-Renaming	**97.96%**	**77.66%**	**99.55%**	**99.49%**	**98.86%**
Perturbed	Proposed	95.94%	76.45%	97.66%	97.76%	98.29%
Pretrained	Llama 3.2 3B	24.33%	0.34%	68.17%	63.27%	62.34%

immune to these alterations. We train these methods only on the perturbed dataset since training them again on the normal dataset amounts to training with different random samples.

While the original model performs significantly worse under perturbation, both alpha-renaming and proposed models match the baseline performance in correctness ratio despite perturbation. This observation suggests that these modifications introduce a robust inductive bias that makes the models resistant to perturbations in the data. A minor decrease in the ratio of exact matches is noted, but this may signify less overfitting and a better bias-variance tradeoff in the larger context. Section 4.3.2 continues this experiment with limited amount of training samples instead of perturbations.

4.3.2 Limited Dataset

Table 4.6 contains evaluations of the baseline, the alpha-renaming model, and the proposed model trained with a severely limited number of samples: 80,000 instead of 799,909. We kept the number of epochs constant, and as a result, the number of training steps were also divided by ten (approximately).

The result of limiting the number of training samples is similar to the dataset perturbation, albeit much less pronounced for the baseline model. Unlike in the perturbation experiment, where the baseline model's performance plummets, all models trained on the reduced dataset maintain similar correctness ratios. The biggest difference is observed in the alpha-covariance values, particularly in the 5 AP category, whose ranking aligns with the perturbation experiment.

Since LTLRandom35 is a synthetic dataset, it exhibits minimal inherent bias, even when the dataset size is limited. Consequently, limiting the dataset size has a smaller effect than introducing perturbations. Furthermore, since the alpha-renaming model was trained using 5 AP embeddings in this experiment, it loses its vocabulary generalization capability unlike our proposed method. Training the alpha-renaming baseline with more APs would require learning a new embedding for each AP, which would reduce its performance.

Table 4.6: Evaluation of the baselines and our method trained on different versions of LTLRandom35. The alpha-renaming baseline was trained using 5 AP embeddings since vocabulary generalization is not evaluated here. First two columns denote the training dataset and the model. Next two columns indicate the ratio of the correct predictions and exact matches on 99,989 test set samples as evaluated by `spot`. Last three columns display mean alpha-covariance values for varying atomic proposition (AP) counts, evaluated on all alpha-equivalent variants of 1000 test samples.

Training		**Evaluation**		**Alpha-Covariance**		
Dataset	**Model**	**Correct**	**Exact**	**3 AP**	**4 AP**	**5 AP**
Normal	Baseline	98.23%	83.23%	96.87%	95.86%	91.80%
Limited	Baseline	87.47%	63.61%	94.37%	91.70%	85.64%
Limited	Alpha-Renaming	**89.50%**	**64.15%**	**99.02%**	**98.67%**	**97.82%**
Limited	Proposed	87.32%	59.04%	97.94%	96.12%	94.34%

4.3.3 Alpha-Covariance

In this section, we focus on the alpha-covariance (Section 3.5) results. For the proposed method, we generate the random embeddings once at the start of an evaluation run using the heuristic explained in Section 4.2.5. Thus, alpha-conversions in this context are equivalent to shuffling the random embeddings in our method, which amounts to measuring our model's robustness against the differences in random embeddings.

We report the results in Table 4.5, which demonstrates that our method has a positive impact on the alpha-covariance, especially in limited data settings. Since the LTLRandom35 dataset was created synthetically, it doesn't have any noteworthy biases and even the baseline enjoys a high alpha-covariance thanks to this. However, when the dataset is perturbed by introducing a bias to the order of APs, the baseline struggles heavily with alpha-covariance, whereas our method does not.

Table 4.7: Mean alpha-covariance values for varying AP counts, evaluated on 1000 test samples, each with 120 random alpha-equivalent variants. The best value for each AP count is highlighted in bold.

(a) 3 to 6 APs

Task	**Model**	**Alpha-Covariance**			
		3 AP	**4 AP**	**5 AP**	**6 AP**
LTL	Full Vocabulary	54.09%	45.51%	45.23%	**42.07**%
	Alpha-Renaming	50.64%	43.00%	40.95%	37.49%
	Proposed	**54.30**%	**46.05**%	**45.64**%	41.88%
Propositional Logic	Full Vocabulary	39.77%	30.08%	30.37%	26.64%
	Alpha-Renaming	42.29%	32.36%	33.45%	**30.28**%
	Proposed	**43.36**%	**32.49**%	**33.65**%	30.04%

(b) 7 to 10 APs

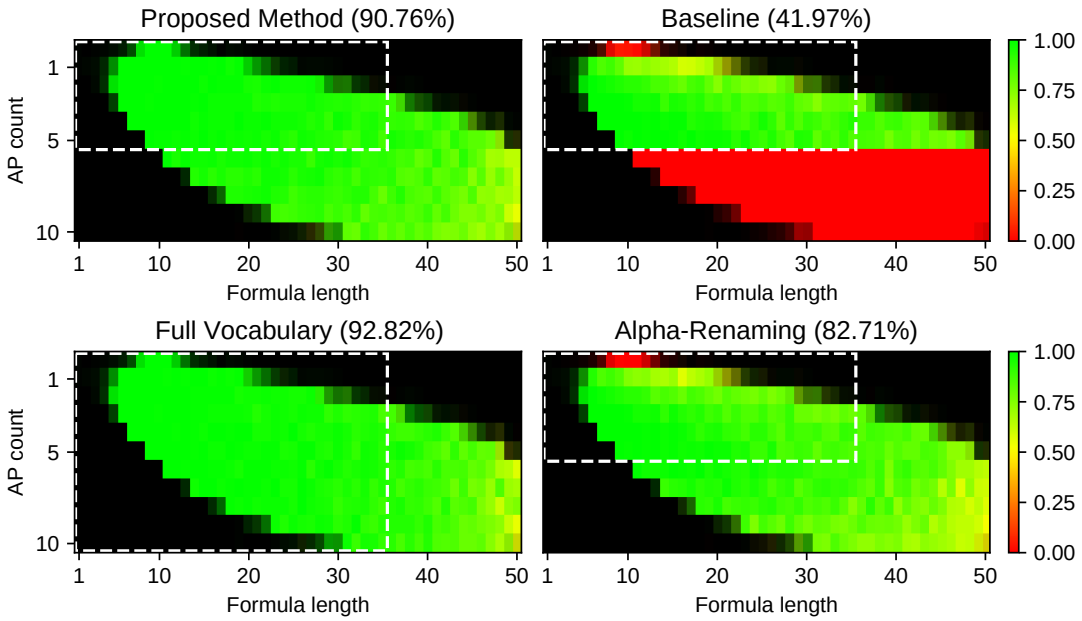
Task	**Model**	**Alpha-Covariance**			
		7 AP	**8 AP**	**9 AP**	**10 AP**
LTL	Full Vocabulary	33.54%	34.47%	32.36%	**28.42**%
	Alpha-Renaming	30.80%	30.30%	28.76%	25.57%
	Proposed	**33.89**%	**35.29**%	**33.18**%	28.34%
Propositional Logic	Full Vocabulary	20.97%	22.97%	18.80%	17.20%
	Alpha-Renaming	24.91%	26.47%	**22.29**%	19.83%
	Proposed	**25.00**%	**26.63**%	21.99%	**20.75**%

4.3.4 Generalization

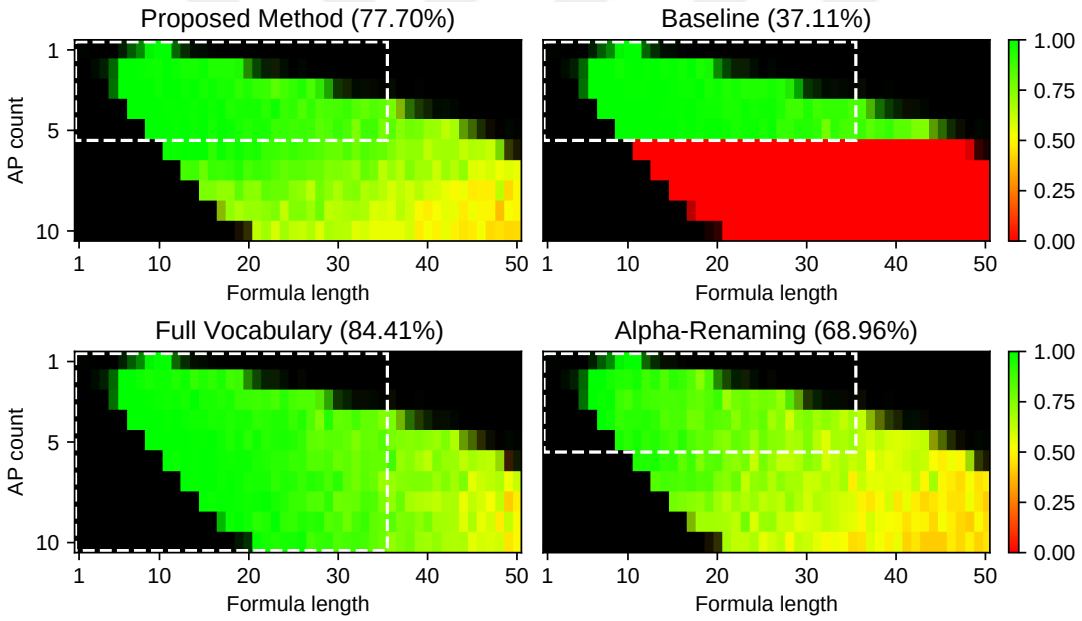
The test dataset for this experiment contains at most 100 formula-trace pairs for each combination of AP count and formula length, whose maximum is 50 instead of 35. We report the results for our model (using Hypercube Vertices, $d_\beta = 5$) and the three baselines in Figure 4.3a. The first baseline uses the same training dataset, whereas the second baseline uses a new LTL dataset with 10 APs, which we create using the same method as LTLRandom35 with `spot`. For the third baseline, we train a fixed embedding model with 10 APs using the same 5 AP dataset but we shuffle the AP embeddings in each forward pass during training. This amounts to creating alpha-equivalent variations of the inputs and outputs.

Discussion. Despite seeing only 5 APs during training, our method performs only slightly worse than the full vocabulary baseline, which represents what a transformer-based model can do with 10 APs. Our method outperforms both the vanilla and the alpha-renaming baselines by a considerable margin, which is significant since the latter is the only other model that can generalize to more APs. Based on this, we hypothesize that the proposed stochastic AP embeddings provide a more explicit enforcement towards learning embedding-covariant transformations in the model, as opposed to training with alpha-renaming, where the learned embeddings may still carry unwanted token-specific biases. Furthermore, unlike the baseline models, our model does not have to learn the concept of AP from scratch for each AP token thanks to the shared embedding part. This could explain why our method shone against the alpha-renaming baseline in the LTL task where the interchangeable tokens are more complex than the copying task.

Motivation for generalization. The generalization to larger AP counts is important especially when considering the exponential growth of the dataset generation time. In Figure 4.4, we visualize the growth pattern of the trace checking duration based on increasing formula length and AP count. The times are relative to the fastest trace checking time. The exact times will vary depending on the machine. In our experiments, generating 100000 samples of exact formula length 50 with at most 10 APs took 2 hours and 21 minutes on a system with 56 threads.



(a) LTL Solving



(b) Propositional Logic

Figure 4.3: Heatmaps visualizing the ratio of correct predictions on a special test set, for LTL solving (top) and propositional logic (bottom) tasks. The brightness of the color depends on the sample size, with full brightness representing 100 samples. The dashed white box represents the boundaries of the training dataset. Our model is competitive with the full vocabulary baseline despite being only trained on formulae with at most 5 APs, and outperforms other baselines.

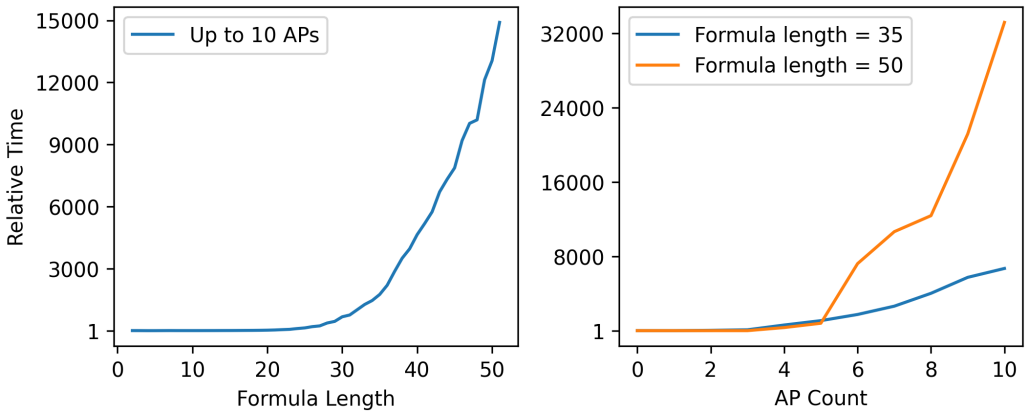


Figure 4.4: Scaling behavior of the trace generation using `spot`.

Alpha-covariance. On the same generalization dataset, we evaluate the alpha-covariance performance of these models in Table 4.7. Note that since 10 APs lead to a lot more naming permutations than 5 APs, the alpha-covariance values are remarkably smaller compared to Table 4.5. Unlike the results from Table 4.5, however, our method outperforms the alpha-renaming approach here. This shows that our method excels in out-of-distribution settings, but trades off some in-distribution performance. Although the full vocabulary baseline performs very similarly to our method, it's important to note that this region is in-distribution for that model. Overall, these results align with Figure 4.3a.

4.4 Assignment Prediction for Propositional Logic

To further demonstrate the applicability and generalization capabilities of our method, we evaluate it on a considerably different logical problem: predicting assignments for propositional logic (Section 2.1.2). The experimental setup is based on DeepLTL [28] with minor differences in hyperparameter choices (Section 4.1.2). We use `pyaiger` [64] to generate datasets and evaluate predictions. In Section 4.4.1, we provide additional details about our experimental setup.

We perform the generalization experiment as in Section 4.3.4 and report the results in Figure 4.3b. The rankings of the methods remain the same, with our method outperforming the vanilla and alpha-renaming baselines. However, performance gaps

are slightly larger overall. Once more, the proposed method is superior to all approaches that use the same 5 AP training dataset, beaten only by the full vocabulary model which sidesteps the challenge of AP generalization due to its enhanced training dataset.

We continue propositional logic experiments in Table 4.7 and Section 4.4.2, which focus on alpha-covariance and dataset perturbations respectively. The results of these experiments also align with the LTL experiments.

4.4.1 Experimental Setup Details

We use PropRandom35 from DeepLTL [28] as our main 5 AP dataset, and create other datasets using the same approach. In particular, propositional logic formulae are generated randomly, with negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$) operators having an equal weight. Equivalence ($\leftrightarrow$) and exclusive or ($\oplus$) operators each have half as much weight since they are derived operators. The corresponding assignment is generated by querying the `pyaiger`'s SAT solver for a minimal unsatisfiable core of the negated formula.

As in the LTL experiments, we use a transformer encoder-decoder architecture with three-way weight tying [48]. The positional encoding method is tree-positional encoding [55] for the encoder and RoPE [58] for the decoder. Predictions are generated using beam search with a beam size of 3.

Since the network outputs the assignments as a sequence (Section 2.1.2), the same assignment can be encoded in multiple ways by changing the order. For example, both `a1b0` and `b0a1` represent the same set of assignments $a = 1$ and $b = 0$, which can be written as $\{(a, 1), (b, 0)\}$ in set notation. We consider such pairs exact matches in the propositional logic experiments. If the predicted assignment does not exactly match the ground truth, we use `pyaiger` to evaluate the correctness.

Table 4.8: Evaluation of the baselines, our method, and Llama 3.2 on the PropRandom35 dataset. The alpha-renaming baseline was trained using 5 AP embeddings since vocabulary generalization is not evaluated here. First two columns denote the training dataset and the model. Next two columns indicate the ratio of the correct predictions and exact matches on 100,000 test set samples as evaluated by `pyaiger`. Last three columns display mean alpha-covariance values for varying atomic proposition (AP) counts, evaluated on all alpha-equivalent variants of 1000 test samples.

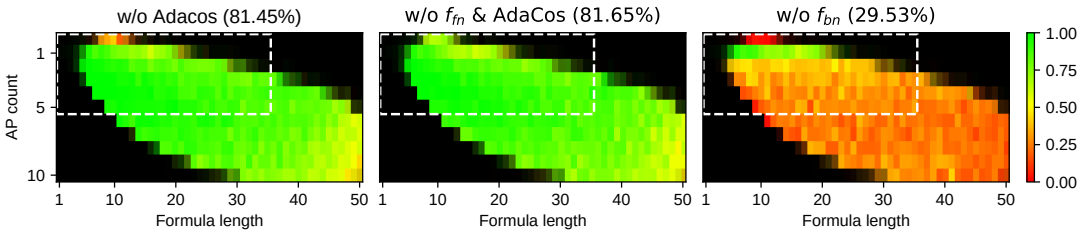
Training		**Evaluation**		**Alpha-Covariance**		
Dataset	**Model**	**Correct**	**Exact**	**3 AP**	**4 AP**	**5 AP**
Normal	Baseline	95.62%	57.94%	95.70%	93.69%	76.02%
Perturbed	Baseline	41.57%	9.04%	14.96%	16.85%	10.65%
Perturbed	Alpha-Renaming	93.85%	57.24%	99.56%	99.60%	93.23%
Perturbed	Proposed	93.25%	56.45%	99.23%	99.42%	92.98%
Pretrained	Llama 3.2 3B	29.03%	1.56%	50.75%	27.96%	11.25%

4.4.2 Dataset Perturbations

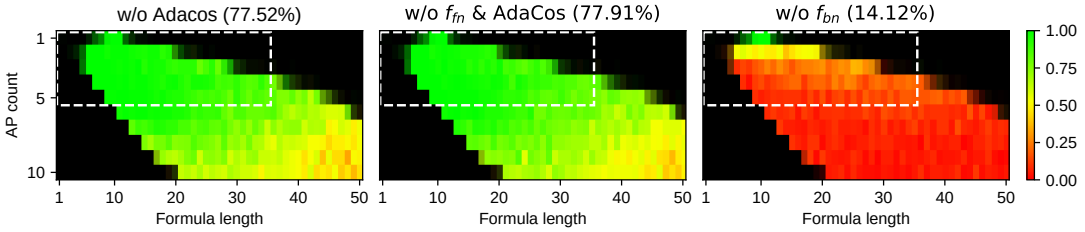
In this section, we repeat the dataset perturbation experiment (Section 4.3.1) for the propositional logic task. The perturbation is introduced in a similar manner by renaming the APs such that the order of the first AP appearances in the label (sequence denoting the Boolean assignment) is always the same. As shown in Table 4.8, the experimental results once again confirm that our method introduces a robust inductive bias for alpha-equivalence.

4.5 Ablation Studies

The hyperparameter search in Section 4.2.4 operates on the copying task, and, alongside searching over the embedding hyperparameters, experiments with disabling the normalization features and AdaCos, thereby constituting an ablation study. For the LTL and propositional logic tasks, we always kept the normalization features and



(a) LTL Solving



(b) Propositional Logic

Figure 4.5: Heatmaps for the ablation studies. The results are reported on the same test set as in Figure 4.3.

AdaCos enabled in the previous sections. In this section, we evaluate the impact of these features by disabling them on our best-performing models for these two logic tasks. We ablate one aspect at a time, except for f_{fn}, which is disabled together with AdaCos because AdaCos depends on f_{fn} to function correctly.

Figure 4.5 presents the results, which demonstrate the critical importance of the f_{bn} normalization component. Removing f_{bn} leads to dramatic performance drops (from 90.76% to 29.53% on LTL, and from 77.70% to 14.12% on propositional logic), confirming that maintaining balance between the common and randomized embedding parts is essential for our method's success. The experiments with AdaCos and f_{fn} indicate task-dependent benefits: they provide significant improvements on LTL (90.76% vs. 81.45% when AdaCos is removed), while showing negligible impact on propositional logic.

4.6 Comparison with LLMs

To contextualize the effectiveness of our proposed approach, we evaluate the performance of a general-purpose LLM (large language model), specifically, the 3B parameter version of Llama 3.2 [27], on the LTL task.

4.6.1 LLM Setup

We use the 3B-parameter version of Llama 3.2 [27], quantized with `Q4_K_M`, and run it using Ollama 0.4.7 as our LLM backend. We first experimented with greedy sampling (by setting top-k=1) since Ollama does not support beam search. However, we found that the default sampling options (top-k=40 and top-p=0.9) yielded better results. Therefore, we use these default settings for all experiments.

Unlike our specialized models, which operate on prefix (Polish) notation, we prompt the LLM using infix notation for input formulas (and output traces in the LTL task), as this format is more prevalent in natural language and more familiar to general-purpose LLMs. To output the assignments in the propositional logic task, we use JSON format, and constrain the LLM's output using a JSON schema. The exact prompts are provided in Appendix A.

We set the random seed to 42 for each sample. Although the reason behind this choice is reproducability, it also seems to improve alpha-covariance. For example, the alpha-covariance values reported for Llama 3.2 in Table 4.5 are 68.17%, 63.27%, 62.34% for 3 to 5 APs, respectively, which decrease to 41.94%, 43.10%, 44.62% when the random seed is no longer fixed.

4.6.2 LLM Results

In the last row of Table 4.5, we report the performance of Llama 3.2 on the test split of LTLRandom35. These results (e.g., 24.33% correct) are drastically lower than those achieved by our proposed method (95.94%). On propositional logic, Llama 3.2 achieves a slightly better accuracy but much worse alpha-covariance (Table 4.8 in

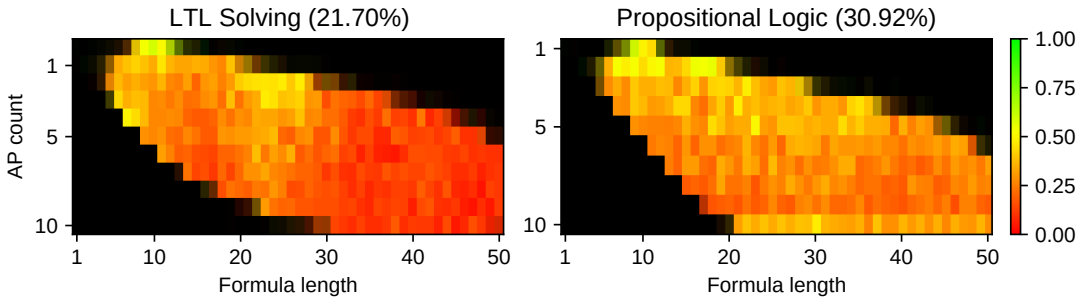


Figure 4.6: Llama 3.2 heatmaps for the two logic tasks.

Section 4.4.2). Additionally, we replicate the setups in Figure 4.3 using Llama 3.2 on the same datasets and sample sizes. As shown in Figure 4.6, the resulting accuracies are 21.70% (LTL solving) and 30.92% (propositional logic), compared to 90.76% and 77.70% by our method. This striking gap illustrates the limitations of general-purpose LLMs in highly specialized domains such as LTL solving, even when the model size far exceeds that of our dedicated architectures.

4.7 Computational Efficiency

To evaluate the practical applicability of our method, we analyze its computational overhead compared to baseline approaches. We report training times, inference speeds, and memory requirements across different experimental settings.

To summarize, our method incurs a modest 13% training overhead compared to the baseline in LTL solving task. At inference, embedding preparation takes only 0.0003 seconds and is required just once at the beginning of an evaluation session, making its cost negligible relative to model execution (0.206 seconds for a forward pass and 9.808 seconds for autoregressive generation). Our optimized method for generating unique random vectors with integer reservoir sampling (Section 3.3) scales efficiently to a large number of vectors unlike the naive approach (Figure 4.7). While the parameter count of traditional embeddings scales linearly with interchangeable token count, our method's parameter count remains constant, as embeddings are shared across interchangeable tokens.

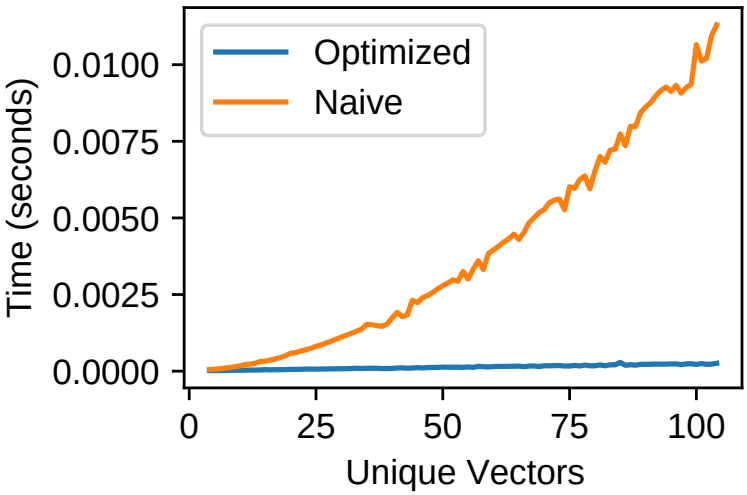


Figure 4.7: Average runtime cost of generating 8-dimensional unique random vectors from Neighboring Points with different uniqueness checking methods.

4.7.1 Training Efficiency

We measured training durations for models trained on NVIDIA H100 GPUs using identical hyperparameter settings. In LTL solving task, the average training times were as follows:

- Baseline (traditional embeddings): 2 hours 12 minutes
- Alpha-renaming baseline: 2 hours 33 minutes
- Proposed method: 2 hours 29 minutes

The proposed method introduces minimal training overhead compared to the baseline, with only a 13% increase in training time. This modest overhead stems from the additional embedding preparation steps required during training.

4.7.2 Inference Performance

We conducted a runtime analysis using our best-performing LTL model on NVIDIA A4000 hardware. The model uses Hypercube Vertices randomization with uniqueness checking enabled, evaluated with batch size 768 and beam search (beam size =

3). In this setup, a forward pass takes 0.206 seconds, and autoregressive generation 9.808 seconds. On the other hand, the embedding preparation time is measured at 0.0003 seconds, which is negligible compared to model execution. Importantly, during inference, embeddings need only be generated once at the start of the evaluation session, making the amortized cost even smaller for batch processing.

4.7.3 Memory Overhead

Our method reduces the total parameter count compared to traditional approaches since only one common embedding is learned for all interchangeable tokens, regardless of their quantity. The memory overhead comes primarily from constructing the embedding matrix during runtime, which requires temporary storage for the randomized components. However, this additional memory requirement is on the same order of magnitude as the embedding matrix itself, which represents a small fraction of total model parameters in transformer architectures.

The parameter efficiency of our method scales favorably with vocabulary size. Unlike traditional approaches that require learning separate embeddings for each token (thereby scaling linearly with the vocabulary size), our method's parameter count remains constant regardless of the number of interchangeable tokens. However, two factors require consideration for very large vocabularies:

1. **Sampling set size**: In discrete random generation methods, the sampling set is naturally bounded (Table 3.1). However, the sampling set grows exponentially with the number of dimensions, ensuring sufficient diversity even for large vocabularies.

2. **Uniqueness checking**: For vocabularies with hundreds of thousands of tokens, uniqueness verification becomes computationally expensive, but the probability of collisions decreases exponentially with increasing embedding dimensions.

CHAPTER 5

DISCUSSION

5.1 Limitations

While our method provides an effective framework for enforcing alpha-equivalence in formal languages, it is not directly applicable to natural language, in which tokens carry semantic and contextual information that is often essential for interpretation. For instance, even though variable names like `electricity_bill` and `water_bill` may be functionally interchangeable in certain code constructs, they convey distinct meanings that are not preserved under alpha-conversions when their embeddings are randomized. As such, enforcing alpha-equivalence may reduce interpretability and degrade performance in tasks that rely on linguistic connotations. Thus, applying our approach to problems in which the interchangeable tokens have meaningful names (e.g., human-written variable names) represents an intriguing area for future research.

Another limitation is the requirement to manually define the set of interchangeable tokens, which may not be feasible in some settings where token interchangeability is context-dependent or dynamically evolving. Our method assumes this set is known a priori. Moreover, our method requires training from scratch due to modifications in the embedding architecture, posing challenges for integration with pretrained models.

Although our dual-part embedding method demonstrates generalization capabilities, its performance in the LTL solving task decreases slightly for in-distribution data (Table 4.5). The future work can tackle this issue, which may eventually lead to Pareto improvements in bias-variance tradeoff. Finally, new randomization and normalization methods for our embeddings can be explored.

Scalability presents additional trade-offs. Although the method scales well by design (requiring only a single shared component for all interchangeable tokens) the discrete random generation mechanism introduces complexities. For example, increasing the embedding dimension expands the sampling set size and improves uniqueness guarantees, yet it may necessitate a corresponding increase in overall model capacity. Despite these concerns, we believe these limitations highlight promising directions for future work, particularly in adapting the method to natural language applications and enabling dynamic identification of interchangeable tokens.

5.2 Conclusion

A central goal in machine learning is to generalize to out-of-distribution samples, for which the model design and its inductive biases play a vital role. In this work, we tackle the challenge of generalizing to larger vocabulary sizes unseen during training and creating an inductive bias for alpha-equivalence. We also contribute the alpha-covariance metric for measuring the model consistency against alpha-equivalent inputs. These contributions embody a foundation for learning extensible vocabularies for interchangeable tokens, which is especially useful for formal reasoning tasks in which alpha-equivalence naturally arises.

Bibliography

[1] Conversion (Chapter 2). In H. P. Barendregt, editor, *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*, pages 22–49. 1984.

[2] E. Abbe, S. Bengio, A. Lotfi, and K. Rizk. Generalization on the unseen, logic reasoning and degree curriculum. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31–60, 23–29 Jul 2023.

[3] M. F. Arif, D. Larraz, M. Echeverria, A. Reynolds, O. Chowdhury, and C. Tinelli. Syslite: Syntax-guided synthesis of pltl formulas from finite traces. In *2020 Formal Methods in Computer Aided Design (FMCAD)*, pages 93–103, 2020.

[4] Z. Azerbayev, H. Schoelkopf, K. Paster, M. D. Santos, S. M. McAleer, A. Q. Jiang, J. Deng, S. Biderman, and S. Welleck. Llemma: An open language model for mathematics. *ArXiv*, abs/2310.10631, 2023.

[5] C. Baier and J.-P. Katoen. Principles of model checking. 2008.

[6] M. Balunovic, P. Bielik, and M. Vechev. Learning to solve smt formulas. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 10317–10328. Curran Associates, Inc., 2018. URL `http://papers.nips.cc/paper/8233-learning-to-solve-smt-formulas.pdf`.

[7] K. Bansal, S. M. Loos, M. N. Rabe, C. Szegedy, and S. Wilcox. HOList: An environment for machine learning of higher-order theorem proving. In *arXiv preprint arXiv:1904.03241*, 2019.

[8] E. Bartocci, L. Bortolussi, and G. Sanguinetti. Data-driven statistical learning of

temporal logic properties. In *Formal Modeling and Analysis of Timed Systems*, pages 23–37, Cham, 2014.

[9] E. Bartocci, N. Manjunath, L. Mariani, C. Mateis, and D. Ničković. Automatic failure explanation in cps models. In *IEEE International Conference on Software Engineering and Formal Methods*, 2019.

[10] E. Bartocci, C. Mateis, E. Nesterini, and D. Nickovic. Survey on mining signal temporal logic specifications. *Information and Computation*, 289:104957, 2022. ISSN 0890-5401.

[11] G. Batt, D. Ropers, H. de Jong, J. Geiselmann, R. Mateescu, M. Page, and D. Schneider. Validation of qualitative models of genetic regulatory networks by model checking: analysis of the nutritional stress response in escherichia coli. *Bioinformatics*, 21 Suppl 1:i19–28, 2005.

[12] C. Belta, B. Yordanov, and E. Aydin Gol. *Formal Methods for Discrete-Time Dynamical Systems*. Studies in Systems, Decision and Control. Springer, 2017.

[13] G. Bombara, C.-I. Vasile, F. Penedo, H. Yasuoka, and C. Belta. A decision tree approach to data classification using signal temporal logic. pages 1–10, 04 2016.

[14] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. 2020.

[15] A. Camacho and S. A. McIlraith. Learning interpretable models expressed in linear temporal logic. *Proceedings of the International Conference on Automated Planning and Scheduling*, 29(1):621–630, May 2021.

[16] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. Handbook of model checking. In *Cambridge International Law Journal*, 2018.

[17] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models, 2023. URL `https://arxiv.org/abs/2303.04864`.

[18] S. Dargan, M. Kumar, M. R. Ayyagari, and G. Kumar. A Survey of Deep Learning and Its Applications: A New Paradigm to Machine Learning. *Archives of Computational Methods in Engineering*, 27(4):1071–1092, Sept. 2020. ISSN 1886-1784.

[19] A. Duret-Lutz, E. Renault, M. Colange, F. Renkin, A. G. Aisse, P. Schlehuber-Caissier, T. Medioni, A. Martin, J. Dubois, C. Gillard, and H. Lauko. From Spot 2.0 to Spot 2.10: What's new? In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*, volume 13372 of *Lecture Notes in Computer Science*, pages 174–187, Aug. 2022.

[20] G. Fainekos, A. Girard, H. Kress-Gazit, and G. Pappas. Temporal logic motion planning for dynamic robots. *Autom.*, 45:343–352, 2009.

[21] N. Fijalkow and G. Lagarde. The complexity of learning linear temporal formulas from examples. In *Proceedings of the Fifteenth International Conference on Grammatical Inference*, volume 153 of *Proceedings of Machine Learning Research*, pages 237–250, 23–27 Aug 2021.

[22] S. Frieder, L. Pinchetti, R.-R. Griffiths, T. Salvatori, T. Lukasiewicz, P. Petersen, A. Chevalier, and J. J. Berner. Mathematical capabilities of chatgpt. *ArXiv*, abs/2301.13867, 2023.

[23] J. Gaglione, D. Neider, R. Roy, U. Topcu, and Z. Xu. Learning linear temporal properties from noisy data: A maxsat-based approach. In *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Proceedings*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 74–90, Germany, 2021.

[24] T. Gauthier, C. Kaliszyk, and J. Urban. Tactictoe: Learning to reason with hol4 tactics. *arXiv preprint arXiv:1804.00595*, 2018.

[25] E. Ghiorzi, M. Colledanchise, G. Piquet, S. Bernagozzi, A. Tacchella, and L. Natale. Learning linear temporal properties for autonomous robotic systems. *IEEE Robotics and Automation Letters*, 8:2930–2937, 2023.

[26] N. Gopalan, D. Arumugam, L. L. S. Wong, and S. Tellex. Sequence-to-sequence language grounding of non-markovian task specifications. *Robotics: Science and Systems XIV*, 2018.

[27] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[28] C. Hahn, F. Schmitt, J. U. Kreber, M. N. Rabe, and B. Finkbeiner. Teaching temporal logics to neural networks. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, 2021.

[29] J. M. Han, J. M. Rute, Y. Wu, E. W. Ayers, and S. Polu. Proof artifact co-training for theorem proving with language models. *ArXiv*, abs/2102.06203, 2021.

[30] D. Huang, P. Dhariwal, D. Song, and I. Sutskever. Gamepad: A learning environment for theorem proving. *arXiv preprint arXiv:1806.00608*, 2018.

[31] X. Jin, A. Donzé, J. V. Deshmukh, and S. A. Seshia. Mining requirements from closed-loop control models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1704–1717, 2015.

[32] P.-A. Kamienny, S. d'Ascoli, G. Lample, and F. Charton. End-to-end symbolic regression with transformers. *ArXiv*, abs/2204.10532, 2022.

[33] A. Ketenci and E. A. Gol. Synthesis of monitoring rules via data mining. In *2019 American Control Conference (ACC)*, pages 1684–1689, 2019.

[34] M. Kloetzer and C. Belta. Temporal logic planning and control of robotic swarms by hierarchical abstractions. *IEEE Transactions on Robotics*, 23(2): 320–330, 2007.

[35] G. Lample and F. Charton. Deep learning for symbolic mathematics. *ArXiv*, abs/1912.01412, 2019.

[36] G. Lederman, M. N. Rabe, E. A. Lee, and S. A. Seshia. Learning heuristics for quantified boolean formulas through deep reinforcement learning. 2020. URL `http://arxiv.org/abs/1807.08058`.

[37] K. Leung, N. Aréchiga, and M. Pavone. Backpropagation through signal temporal logic specifications: Infusing logical structure into gradient-based methods. *The International Journal of Robotics Research*, 42(6):356–370, 2023.

[38] J. Li, Y. Yao, G. Pu, L. Zhang, and J. He. Aalta: an ltl satisfiability checker over infinite/finite traces. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 731–734, New York, NY, USA, 2014.

[39] B. Liu, J. T. Ash, S. Goel, A. Krishnamurthy, and C. Zhang. Transformers learn shortcuts to automata. 2023.

[40] J. X. Liu, Z. Yang, B. Schornstein, S. Liang, I. Idrees, S. Tellex, and A. Shah. Lang2LTL: Translating natural language commands to temporal specification with large language models. In *Workshop on Language and Robotics at CoRL 2022*, 2022.

[41] S. Loos, G. Irving, C. Szegedy, and C. Kaliszyk. Deep network guided proof search. In *LPAR*, 2017.

[42] A. Mavrogiannis, C. Mavrogiannis, and Y. Aloimonos. Cook2ltl: Translating cooking recipes to ltl formulae using large language models. *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 17679–17686, 2023. URL `https://api.semanticscholar.org/CorpusID:263333981`.

[43] S. Mohammadinejad, J. V. Deshmukh, and A. G. Puranic. Mining environment assumptions for cyber-physical system models. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pages 87–97, 2020.

[44] I. Morazzoni, V. Scotti, and R. Tedesco. Def2vec: Extensible word embeddings from dictionary definitions. In *International Conference on Natural Language and Speech Processing*, 2023.

[45] D. Neider and I. Gavran. Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2018.

[46] R. Patel, E. Pavlick, and S. Tellex. Grounding language to non-markovian tasks with no supervision of task specifications. *Robotics: Science and Systems XVI*, 2020.

[47] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977.

[48] O. Press and L. Wolf. Using the output embedding to improve language models. In *Conference of the European Chapter of the Association for Computational Linguistics*, 2016.

[49] A. G. Puranic, J. V. Deshmukh, and S. Nikolaidis. Learning from demonstrations using signal temporal logic in stochastic and continuous domains. *IEEE Robotics and Automation Letters*, 6(4):6250–6257, 2021.

[50] M. N. Rabe, D. Lee, K. Bansal, and C. Szegedy. Mathematical reasoning via self-supervised skip-tree training. *arXiv: Learning*, 2020.

[51] M. N. Rabe, D. Lee, K. Bansal, and C. Szegedy. Mathematical reasoning via self-supervised skip-tree training. 2020.

[52] R. Ranjan, C. D. Castillo, and R. Chellappa. L2-constrained softmax loss for discriminative face verification, 2017.

[53] V. Sanh, A. Webson, C. Raffel, S. H. Bach, L. Sutawika, Z. Alyafeai, A. Chaffin, A. Stiegler, A. Raja, M. Dey, M. S. Bari, C. Xu, U. Thakker, S. S. Sharma, E. Szczechla, T. Kim, G. Chhablani, N. V. Nayak, D. Datta, J. Chang, M. T.-J. Jiang, H. Wang, M. Manica, S. Shen, Z.-X. Yong, H. Pandey, R. Bawden, T. Wang, T. Neeraj, J. Rozen, A. Sharma, A.-. drea Santilli, T. Févry, J. A. Fries, R. Teehan, T. L. Scao, S. Biderman, L. Gao, T. Wolf, and A. M. Rush. Multitask prompted training enables zero-shot task generalization. In *International Conference on Learning Representations*, 2022. URL `https://api.semanticscholar.org/CorpusID:276421109`.

[54] D. Selsam and N. Bjørner. Guiding high-performance SAT solvers with unsat-core predictions. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019,*

Proceedings, pages 336–353, 2019. doi: 10.1007/978-3-030-24258-9_24. URL https://doi.org/10.1007/978-3-030-24258-9_24.

[55] V. L. Shiv and C. Quirk. Novel positional encodings to enable tree-based transformers. In *NeurIPS 2019*, Dec. 2019.

[56] Y. Shoukry, P. Nuzzo, A. Balkan, I. Saha, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada. Linear temporal logic motion planning for teams of underactuated robots using satisfiability modulo convex programming. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 1132–1137, 2017.

[57] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. In *Symposium on the Theory of Computing*, 1982.

[58] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomput.*, 568(C), Feb. 2024. ISSN 0925-2312. doi: 10.1016/j.neucom.2023.127063. URL https://doi.org/10.1016/j.neucom.2023.127063.

[59] D. Sun, J. Chen, S. Mitra, and C. Fan. Multi-agent motion planning from signal temporal logic specifications. *IEEE Robotics and Automation Letters*, PP:1–1, 2022.

[60] X. Tang, Z. Zheng, J. Li, F. Meng, S.-C. Zhu, Y. Liang, and M. Zhang. Large language models are in-context semantic reasoners rather than symbolic reasoners. *ArXiv*, abs/2305.14825, 2023.

[61] P. Varnai and D. V. Dimarogonas. On robustness metrics for learning stl tasks. In *2020 American Control Conference (ACC)*, pages 5394–5399, 2020.

[62] M. Vastl, J. Kulhánek, J. Kubalík, E. Derner, and R. Babuška. Symformer: End-to-end symbolic regression using transformer-based architecture, 2022.

[63] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[64] M. Vazquez-Chanlatte and M. Rabe. py-aiger, 2024. URL `https://github.com/mvcisback/py-aiger`.

[65] M. Vazquez-Chanlatte, J. Deshmukh, X. Jin, and S. Seshia. Logical clustering and learning for time-series data. pages 305–325, 07 2017.

[66] F. Wang, X. Xiang, J. Cheng, and A. L. Yuille. Normface: L2 hypersphere embedding for face verification. In *Proceedings of the 25th ACM international conference on Multimedia*, MM '17, Oct. 2017.

[67] F. Wang, Z. Cao, L. Tan, and H. Zong. Survey on learning-based formal methods: Taxonomy, applications and possible future directions. *IEEE Access*, 8: 108561–108578, 2020.

[68] S. Wei, X. Chen, X. Yang, S. Cao, and X. Zhang. A component-based vocabulary-extensible sign language gesture recognition framework. *Sensors (Basel, Switzerland)*, 16, 2016.

[69] S. Welleck, J. Liu, X. Lu, H. Hajishirzi, and Y. Choi. Naturalprover: Grounded mathematical proof generation with language models. *ArXiv*, abs/2205.12910, 2022.

[70] Z. Wu, L. Qiu, A. Ross, E. Akyürek, B. Chen, B. Wang, N. Kim, J. Andreas, and Y. Kim. Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks. In *North American Chapter of the Association for Computational Linguistics*, 2023.

[71] R. Yan, A. Julius, M. Chang, A. Fokoue, T. Ma, and R. Uceda-Sosa. Stone: Signal temporal logic neural network for time series classification. In *2021 International Conference on Data Mining Workshops (ICDMW)*, pages 778–787, 2021. doi: 10.1109/ICDMW53433.2021.00101.

[72] K. Yang, A. M. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. J. Prenger, and A. Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *ArXiv*, abs/2306.15626, 2023.

[73] X. Zhang, R. Zhao, Y. Qiao, X. Wang, and H. Li. Adacos: Adaptively scaling cosine logits for effectively learning deep face representations. *2019 IEEE/CVF*

Conference on Computer Vision and Pattern Recognition (CVPR), pages 10815–10824, 2019.

APPENDICES

A LLM Prompts

The input prompts for the LTL and propositional logic tasks are given in Listing 5.1 and Listing 5.2, respectively. For each sample, the "`{formula}`" substring in the prompt is replaced by the input formula, and the prompt is given as a user message to the LLM.

Listing 5.1: LLM Prompt for the LTL solving task.

```
Your task is to generate a satisfying trace for a given LTL (Linear Temporal Logic) formula.
Lowercase letters denote the atomic propositions.
The output trace should be in lasso form composed of two parts: the prefix part and the cycle part.
Timesteps in the trace should be separated by semicolons, and the cycle part should be enclosed in curly braces, preceeded by the keyword "cycle".

Temporal operators:
X: Next operator
U: Until operator

Logical operators:
 : AND operator
 : OR operator
!: NOT operator
The output trace is a symbolic trace, which means that the logical operators are allowed, but not temporal operators.

Constants:
0: False
```

```
1: True
Note that other numbers are invalid.

Example 1
Formula: X((a   Xa) U XXb)
Trace: 1; 1; 1; b; cycle  1

Example 2
Formula: !c U X(1 U b)
Trace: 1; b; cycle  1

Example 3
Formula: X!X!(b   Xb)
Trace: 1; 1; b; b; cycle  1

Example 4
Formula: !(1 U !c)
Trace: cycle  c

Your Turn
Formula:  formula
Please generate the corresponding trace. Output the trace only.
```

Listing 5.2: LLM Prompt for the propositional logic task.

```
Your task is to generate an assignment that satisfies a given
   propositional logic formula.
Lowercase letters denote the atomic propositions.
The output is a JSON object representing the assignment.

Logical operators (ordered from highest precedence to lowest):
!: NOT operator
 : AND operator
 : OR operator
xor: Exclusive OR operator
   : Logical equivalence operator (biconditional)

Constants:
```

```
0: False
1: True
Note that other numbers are invalid.

Example 1
Formula: !a   c   (b     c)
Assignment:   "a": false

Example 2
Formula: !(a     (!a xor !e))
Assignment:   "a": true, "e": true

Example 3
Formula: a   (!a     !c   d)
Assignment:   "a": true, "c": true, "d": false

Example 4
Formula: !(a   !(!d   b   d))
Assignment:   "a": false, "d": false

Your Turn
Formula:  formula
Please generate an assignment that satisfies this formula. Output
   the assignment only, in JSON format.
```