Applications of AI to study of finite algebraic structures and automated theorem proving

Boris Shminke

Applications de l'IA à l'étude des structures algébriques finies et à la démonstration automatique de théorèmes

Boris Shminke

Laboratoire Jean Alexandre Dieudonné (LJAD)

Présentée en vue de l'obtention du grade de docteur en mathématiques **d'**Université Côte d'Azur
Dirigée par : Carlos Simpson
Soutenue le : 01/09/2023

Devant le jury, composé de :
David Alfaya Sánchez, Assistant Professor, Universidad Pontificia Comillas
Kevin Buzzard, Professor, Imperial College London
Mai Gehrke, Directrice de recherche, Université Côte d'Azur
Moa Johansson, Associate Professor, Chalmers University
Laura Kovács, Professor, Vienna University of Technology
Carlos Simpson, DR, Université Côte d'Azur
Martin Suda, Senior Researcher, Czech Technical University in Prague

UNIVERSITÉ CÔTE D'AZUR | INTERDISCIPLINARY INSTITUTE FOR ARTIFICIAL INTELLIGENCE 3IA CÔTE D'AZUR | FRANCE 2030

Applications de l'IA à l'étude des structures algébriques finies et à la démonstration automatique de théorèmes

Jury :

Présidente du jury :

Laura Kovács, Professor, Vienna University of Technology

Rapporteurs :

Kevin Buzzard, Professor, Imperial College London

Martin Suda, Senior Researcher, Czech Technical University in Prague

Examinateur·rice·s :

Mai Gehrke, Directrice de Recherche, Université Côte d'Azur

Moa Johansson, Associate Professor, Chalmers University

David Alfaya Sánchez, Assistant Professor, Universidad Pontificia Comillas

Directeur de Thèse :

Carlos Simpson, Directeur de Recherche, Université Côte d'Azur

Applications de l'IA à l'étude des structures algébriques finies et à la démonstration automatique de théorèmes

Résumé

Cette thèse contribue à une recherche de modèles finis et à la démonstration automatisée de théorèmes, en se concentrant principalement, mais sans s'y limiter, sur les méthodes d'intelligence artificielle. Dans la première partie, nous résolvons une question de recherche ouverte à partir de l'algèbre abstraite en utilisant une recherche automatisée de modèles finis massivement parallèles, en utilisant l'assistant de preuve Isabelle. À savoir, nous établissons l'indépendance de certaines lois de distributivité abstraites dans les binaires résiduels dans le cas général. En tant que sous-produit de cette découverte, nous apportons un client Python au serveur Isabelle. Le client a déjà trouvé son application dans les travaux d'autres chercheurs et de l'enseignement supérieur. Dans la deuxième partie, nous proposons une architecture de réseau neuronal génératif pour produire des modèles finis de structures algébriques appartenant à une variété donnée d'une manière inspirée des modèles de génération d'images tels que les GAN (réseaux antagonistes génératifs) et les autoencodeurs. Nous contribuons également à un paquet Python pour générer des semi-groupes finis de petite taille comme implémentation de référence de la méthode proposée. Dans la troisième partie, nous concevons une architecture générale de guidage des vérificateurs de saturation avec des algorithmes d'apprentissage par renforcement. Nous contribuons à une collection d'environnements compatibles OpenAI Gym pour diriger Vampire et iProver et démontrons sa viabilité sur des problèmes sélectionnés de la bibliothèque TPTP (Thousand of Problems for Theorem Provers). Nous contribuons également à une version conteneurisée d'un modèle ast2vec existant et montrons son applicabilité à l'incorporation de formules logiques écrites sous la forme clausal-normale. Nous soutenons que l'approche modulaire proposée peut accélérer considérablement l'expérimentation de différentes représentations de formules logiques et de schémas de génération de preuves synthétiques à l'avenir, résolvant ainsi le problème de la rareté des données, limitant notoirement les progrès dans l'application des techniques d'apprentissage automatique pour la démonstration automatisée de théorèmes.

Mots clés : intelligence artificielle, démonstration automatique de théorèmes, structures algebriques finies

Applications of AI to study of finite algebraic structures and automated theorem proving

Abstract

This thesis contributes to a finite model search and automated theorem proving, focusing primarily but not limited to artificial intelligence methods. In the first part, we solve an open research question from abstract algebra using an automated massively parallel finite model search, employing the Isabelle proof assistant. Namely, we establish the independence of some abstract distributivity laws in residuated binars in the general case. As a by-product of this finding, we contribute a Python client to the Isabelle server. The client has already found its application in the work of other researchers and higher education. In the second part, we propose a generative neural network architecture for producing finite models of algebraic structures belonging to a given variety in a way inspired by image generation models such as GANs (generative adversarial networks) and autoencoders. We also contribute a Python package for generating finite semigroups of small size as a reference implementation of the proposed method. In the third part, we design a general architecture of guiding saturation provers with reinforcement learning algorithms. We contribute an OpenAI Gym-compatible collection of environments for directing Vampire and iProver and demonstrate its viability on select problems from the Thousands of Problems for Theorem Provers (TPTP) library. We also contribute a containerised version of an existing ast2vec model and show its applicability to embedding logical formulae written in the clausal-normal form. We argue that the proposed modular approach can significantly speed up experimentation with different logic formulae representations and synthetic proof generation schemes in future, thus addressing the data scarcity problem, notoriously limiting the progress in applying the machine learning techniques for automated theorem proving.

Keywords: artificial intelligence, automated theorem proving, finite algebraic structures

Acknowledgements

My research leading to this thesis will not be possible without input from many people, and I use this page to say thanks to my wife Sveta, who supported me every time I needed it and unconsciously hinted at one of the key ideas; Édouard Balzin for initiating an innocent side project that resulted in writing this thesis; Carlos Simpson for supervising my look for the truth non-pervasively but consistently; Wesley Fussner for showing me the beauty of quantum logic models and directing me towards a crackable problem; Anthony Bordg for suggesting to try Nitpick for cases where Mace4 failed; Jean-Marc Lacroix for fighting processes in uninterruptible sleep spawned as by-products of my scripts running on the lab servers; Fabian Huch for becoming the first unaffiliated user of `isabelle-client`; Martin Suda, whose adherence to open-source software principles helped me understand much better, co k čemu; Michael Rawson for showing an example and encouraging me to undertake the most breathtaking experiments; Zacharaya Shabka for sharing his experience in graph neural networks for reinforcement learning; Athanasios Vasileiadis for questioning why my algorithms should work in the first place, particularly when I was sure they did; Zsolt Zombori for reanimating FLoP code for me; Konstantin Korovin for adding new features to iProver to facilitate my experiments; Ali Ballout for helping me discover semantic representations of logic formulae.

I am also grateful to people who did not contribute to my research directly but made other parts of my studies much brighter. I bow to: Clara Salaun for breaking a stereotype of slow French bureaucracy; Najwa Ghannoum for taking me through the PhD candidate survival guide and looking after Scott; Maryse De Micheli for guiding me along the rocky road of being a French language learner and always finding something to praise, even when I was not that sure I was doing well; Vivien Lake for helping me not to get mad while teaching Master students; Mehdi Zaïdi for persuading me that there is life after thesis.

I thank Kevin Buzzard for writing a report on this thesis and Mai Gehrke, Laura Kovács, Moa Johansson, and David Alfaya Sánchez for serving as jury members for its defence.

This work has been supported by the French government, through the 3IA Côte d'Azur Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-19-P3IA-0002. Part of this work was performed using HPC resources from GENCI-IDRIS (Grant 2021-AD011013125).

Finally, I bring my anonymised but sincere gratitude to all others who made my journey, if not always enjoyable, but at least not always impossible.

Contents

List of Figures

CHAPTER 1

General introduction

> Search for answers on the perennial quest
> Where dreams are followed, and time is a test
> Chuck Schuldiner, "Symbolic", 1995

I first learnt that computers could prove theorems when I was around twelve. A couple of years before entering the university, reading a Russian translation of one of Gottfried Wilhelm Leibniz's works brought to my attention the idea that some things, about which people often vividly argue, one can compute by applying a suitable calculus. I passed my university courses on mathematical logic with no particular interest because they were too dry and "applied" (translated into too many computations and a desperate lack of their meaningfulness). In parallel, during one of the summer holidays, I read a book on logic [31] "for humanity students", but an unorthodox one (at least, for Russian humanities departments), balancing between mathematical technicalities and philosophical discourse. This book only amplified my *calculemus* attitude, which probably crystallised when I learnt how to formalise mathematics in Mizar [44] shortly after my graduation. Soon after a paper [117] with my Mizar formalisation appeared, I moved on to a still booming "next big thing": artificial intelligence (AI). After working in the private sector and applying AI in entertainment and banking for five years, I could not imagine myself doing something as "impractical" as good old *calculemus* again. Even less can I express how happy I am I did.

I describe my journey in the following four chapters: the first two deal with applications of artificial intelligence to the study of finite algebraic structures, and the last two — with its applications to automated theorem proving.

In Chapter 2, we apply artificial intelligence in a more general sense (not including machine learning) to generate finite algebraic structures and partly solve a previously open problem in abstract algebra.

In Chapter 3, we talk about machine learning in general and deep learning in particular and contribute a novel deep neural network architecture to generate finite models of algebraic structures.

In Chapter 4, we dive into reinforcement learning and automated theorem proving and contribute an environment for training agents to prove theorems in different calculi.

In Chapter 5, we contribute a proof of concept of a generic reinforcement learning

prover of micro-service architecture. We also report the experimental results demonstrating the viability of our architecture and its ability to generalise in a meta-learning sense. Since I worked alone on this project and not for the total timespan of my studies, I did not target to build a competition-ready prover but to "use RL as a research tool to further our understanding of proof search dynamics" [72].

CHAPTER 2

Artificial Intelligence for Model Search of Finite Algebraic Structures

This chapter describes a collaborative project of applying artificial intelligence tools to finite algebraic structures studies. We published the main result (a solution to an open research problem in abstract algebra) as a short peer-reviewed paper [34] at an international conference. To get this result, the thesis author contributed an open-source software package [114], two consecutive versions of which were peer-reviewed and published ([68] and [64]) in the international conference proceedings. In addition, the thesis author submitted a pre-print [113] of a fuller package description.

Being funded by the Interdisciplinary Institute for Artificial Intelligence, this thesis author sees collaborations with mathematicians like this one as an integral part of his mission. Coming from private sector research, he knew that many domains could benefit from applications of novel software solutions, that one can automate many unlikely tasks for the benefit of all, and that artificial intelligence techniques need not be high-end (or even include machine learning) to be fruitful. We started this ingenuously interdisciplinary project on a what-if basis without guarantees that our approach might bring any noteworthy results. Nevertheless, we managed to make the machine help mathematicians in their work.

Outline of the chapter:

In Section 2.1, we recall definitions for the well-known algebraic structures for the convenience of an artificial intelligence practitioner who might not deal with them daily.

In Section 2.2, we remind definitions of more specific algebraic structures of particular interest for the project, which we will use throughout this chapter.

In Section 2.3, we introduce the problem we managed to solve using artificial intelligence tools and communicate the main mathematical result of the project.

In Section 2.4, we describe a specialised software package developed by the thesis author as his contribution to the project.

In Section 2.5, we detail the software architecture we used to arrive at the solution.

Finally, in Section 2.6, we discuss the consequent and possible future development of created software and its applications outside the original project. We also reflect on lessons learned from the project and potentially fruitful directions of research related to ours.

2.1 Basic algebraic structures

Definition 2.1 (Binar)**.** A *binar* $(A, \cdot)$ is a set equipped with a single binary operation with no other properties imposed. A binar is often called *magma* and sometimes — a *groupoid*.

Definition 2.2 (Semigroup)**.** A *semigroup* $(A, \cdot)$ is a binar (see Definition 2.1) in which a binary operation (usually called *multiplication*) is *associative*, i.e. for all $x, y, z \in A$:

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z \tag{2.1}$$

Remark 2.3. If a binary operation $\cdot$ satisfies an equation $x \cdot y = y \cdot x$, we talk about a *commutative* operation, and so of a commutative binar or a commutative semigroup.

Definition 2.4 (Monoid)**.** A *monoid* $(A, \cdot, e)$ is an algebraic structure where $(A, \cdot)$ is a semigroup (see Definition 2.2) and an element $e \in A$ (called *identity*) satisfies the following axioms for all $x \in A$:

$$e \cdot x = x \tag{2.2}$$

$$x \cdot e = x \tag{2.3}$$

Example 2.5. A prominent example of a (non-commutative) monoid in computer science is a set of string variables with the concatenation operation and an empty string as an identity.

Definition 2.6 (Lattice)**.** A *lattice* $(A, \wedge, \vee)$ is a set equipped with two binary operations, *meet* $\wedge$ and *join* $\vee$, such that $(A, \wedge)$ and $(A, \vee)$ are both commutative semigroups (see Definition 2.2), and the two following so-called *absorption laws* hold (for all $x, y \in A$):

$$x \wedge (x \vee y) = x \tag{2.4}$$

$$x \vee (x \wedge y) = x \tag{2.5}$$

Remark 2.7. One can view every lattice can as a partially ordered set with an order relation $\leq$ defined as

$$x \leq y \iff x \wedge y = x \tag{2.6}$$

Definition 2.8 (Distributive Lattice)**.** A *distributive lattice* $(A, \wedge, \vee)$ is a lattice (see Definition 2.6), such that for all $x, y, z \in A$:

$$x \wedge (y \vee z) = (x \vee y) \wedge (x \vee z) \tag{2.7}$$

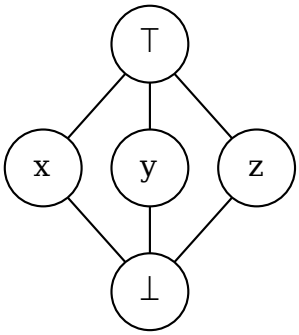

Figure 1: A Hasse diagram of the minimal non-distributive modular lattice.

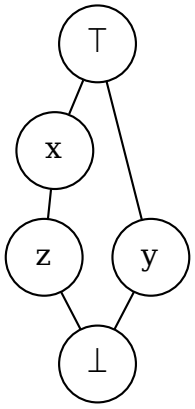

Figure 2: A Hasse diagram of the minimal non-modular lattice.

Example 2.9. Variables of a complex type of immutable set of strings in Python (annotated as `frozenset[str]`) form a distributive lattice with meet and join being `intersection`[1] and `union`[2] respectively.

Definition 2.10 (Modular Lattice)**.** A *modular lattice* $(A, \wedge, \vee)$ is a lattice (see Definition 2.6), such that for all $x, y, z \in A$:

$$(x \wedge y) \vee (x \wedge z) = x \wedge (y \vee (x \wedge z)) \tag{2.8}$$

Remark 2.11. Every distributive lattice is modular.

Example 2.12. The smallest possible non-distributive modular lattice is defined by a Hasse diagram from Figure 1. The smallest possible non-modular lattice has a Hasse diagram from Figure 2.

2.2 Residuated Algebraic Structures

Definition 2.13 (Residuated Binar)**.** A *residuated binar* is an algebraic structure $(A, \wedge, \vee, \cdot, /, \backslash)$ such that $(A, \wedge, \vee)$ is a lattice (see Definition 2.6), $(A, \cdot)$ is any binar (see Definition 2.1) and for all $x, y, z \in A$ the following axiom holds:

$$x \cdot y \leq z \iff y \leq x \backslash z \iff x \leq z / y. \tag{2.9}$$

Remark 2.14. If the underlying binar $(A, \cdot)$ of a residuated binar is a semigroup (see Definition 2.2), one talks of a *residuated semigroup*.

Definition 2.15 (Residuated Magma)**.** A *residuated magma* is an algebraic structure $(A, \leq, \cdot, /, \backslash)$ such that $(A, \leq)$ is a partially ordered set and the law 2.9 holds.

[1] `https://docs.python.org/3/library/stdtypes.html#frozenset.intersection`
[2] `https://docs.python.org/3/library/stdtypes.html#frozenset.union`

Remark 2.16. Since every lattice is a partially ordered set but not vice versa, one can see the residuated magma concept as a strong generalisation of residuated binars.

Definition 2.17 (Residuated Lattice)**.** A *residuated lattice* is an algebraic structure $(A, \wedge, \vee, \cdot, /, \backslash, e)$ such that $(A, \wedge, \vee, \cdot, \backslash, /)$ is a residuated binar (see Definition 2.13) and $(A, \cdot, e)$ is a monoid (see Definition 2.4).

Remark 2.18. The study of residuated binars and lattices can have many applications: for example, they serve as algebraic models of so-called *substructural logics* [36]. The latter are logic calculi used in quantum physics [30].

Since residuated binars have five binary operations, one can construct many different abstract distributivity laws using them. It is well known [33] that several such laws hold in any residuated binar, namely the following ones:

$$x \cdot (y \vee z) = (x \cdot y) \vee (x \cdot z) \tag{$\cdot\vee$}$$

$$(x \vee y) \cdot z = (x \cdot z) \vee (y \cdot z) \tag{$\vee\cdot$}$$

$$x\backslash (y \wedge z) = (x\backslash y) \wedge (x\backslash z) \tag{$\backslash\wedge$}$$

$$(x \wedge y) /z = (x/z) \wedge (y/z) \tag{$\wedge/$}$$

$$x/ (y \vee z) = (x/y) \wedge (x/z) \tag{$/\vee$}$$

$$(x \vee y) \backslash z = (x\backslash z) \wedge (y\backslash z) \tag{$\vee\backslash$}$$

Note that $(/\vee)$ and $(\vee\backslash)$ are not exactly distributivity laws between two binary operations. Such equations are sometimes called antidistributivity [50] laws and arise even in classical logic. General distributivity laws can represent inference rules in different models of quantum logic. They were also used to establish non-trivial categorical equivalences [35] and to obtain decidability results for models of program execution [138], among other things.

Examples of distributivity laws which can hold or not depending on a residuated binar at hand (for example, they can all be true in residuated lattices; see Definition 2.17) are the following six:

$$x \cdot (y \wedge z) = (x \cdot y) \wedge (x \cdot z) \tag{$\cdot\wedge$}$$

$$(x \wedge y) \cdot z = (x \cdot z) \wedge (y \cdot z) \tag{$\wedge\cdot$}$$

$$x\backslash (y \vee z) = (x\backslash y) \vee (x\backslash z) \tag{$\backslash\vee$}$$

$$(x \vee y) /z = (x/z) \vee (y/z) \tag{$\vee/$}$$

$$x/ (y \wedge z) = (x/y) \vee (x/z) \tag{$/\wedge$}$$

$$(x \wedge y) \backslash z = (x\backslash z) \vee (y\backslash z) \tag{$\wedge\backslash$}$$

2.3 One Previously Open Problem

Some distributivity laws can follow from combinations of others, namely:

Theorem 2.19 (Theorem 2.3 and Proposition 3.1 of [33])**.** *If in a residuated binar (see Definition 2.13) or a residuated semigroup (see Remark 2.14) the underlying lattice is a distributive one (see Definition 2.8) there are no other implications between the distributivity laws* $(\cdot\wedge)$,$(\wedge\cdot)$,$(\backslash\vee)$,$(\vee/)$,$(/\wedge)$, *and* $(\wedge\backslash)$, *except the following ones:*

1. $(\vee/)$ *and* $(\wedge\backslash)$ *implies* $(\backslash\vee)$.
2. $(\backslash\vee)$ *and* $(/\wedge)$ *implies* $(\vee/)$.
3. $(\cdot\wedge)$ *and* $(\vee/)$ *implies* $(/\wedge)$.
4. $(\wedge\cdot)$ *and* $(\backslash\vee)$ *implies* $(\wedge\backslash)$.
5. $(\wedge\backslash)$ *and* $(\cdot\wedge)$ *implies* $(\wedge\cdot)$.
6. $(/\wedge)$ *and* $(\wedge\cdot)$ *implies* $(\cdot\wedge)$.

[33] was published in Jan 2019, and since then until spring 2021, thus for more than two years, it was not known whether any combination of distributive laws implies one of them without lattice distributivity condition. From personal discussions, we are aware of attempts made to find counter-examples using specialised software (namely Mace4 [74]) and that the co-authors of [33] had opposite opinions on whether 2.19 can be true in the general case. Counter-examples backing 2.19 were of sizes 4 and 5, but Mace4 worked without finding anything for unreasonably long while traversing the model candidates of cardinality 6. We could only guess the reasons for it since for algebraic structures with similar numbers and arities of operations (residuated magmas, see Definition 2.15), one applied the Mace4 proof-finding counterpart Prover9 successfully [122, 55]. Prover9/Mace4 tandem has a long and glorious track record [90] of serving mathematicians, so one can already see its inability to find counter-examples as empirical evidence (by no means sufficient, of course) of their inexistence.

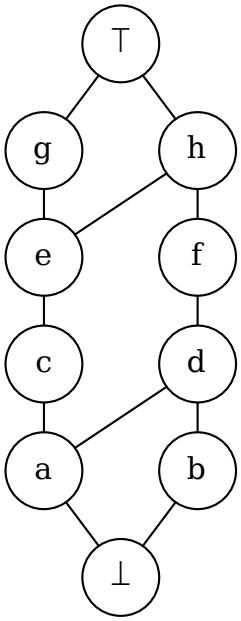

Figure 3: A Hasse diagram of the lattice reduct is the same for all the counter-examples found

Nonetheless, in [34], we found enough finite models to establish the following

Theorem 2.20. *In a general residuated binar, none of the distributivity laws* $(\cdot\wedge)$, $(\wedge\cdot)$, $(\backslash\vee)$, $(\vee/)$, $(/\wedge)$, *and* $(\wedge\backslash)$ *follows from any combination of the others.*

Proof. Let is take six finite residuated binars $\mathbf{A}_1, \mathbf{A}_2, \mathbf{A}_3, \mathbf{A}_4, \mathbf{A}_5$, and $\mathbf{A}_6$ having the same underlying lattice (also called *lattice reduct*, see Figure 3) and the following multiplication tables (one can uniquely reconstruct the tables for $\backslash$ and $/$ using the multiplication table, the lattice structure, and the residuation property 2.9). For $\mathbf{A}_1$ and $\mathbf{A}_2$:

$\cdot$	$\bot$	a	b	c	d	e	f	g	h	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
a	$\bot$	$\bot$	a	$\bot$	a	$\bot$	a	$\bot$	a	a
b	$\bot$	b	$\bot$	b	b	b	b	b	b	b
c	$\bot$	$\bot$	g	$\bot$	g	$\bot$	g	$\bot$	g	g
d	$\bot$	b	a	b	d	b	d	b	d	d
e	$\bot$	$\bot$	g	$\bot$	g	$\bot$	g	$\bot$	g	g
f	$\bot$	b	a	b	d	b	d	b	d	d
g	$\bot$	$\bot$	g	$\bot$	g	$\bot$	g	$\bot$	g	g
h	$\bot$	b	g	b	$\top$	b	$\top$	b	$\top$	$\top$
$\top$	$\bot$	b	g	b	$\top$	b	$\top$	b	$\top$	$\top$

$\cdot$	$\bot$	a	b	c	d	e	f	g	h	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
a	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
b	$\bot$	$\bot$	b	$\bot$	b	$\bot$	b	$\bot$	b	b
c	$\bot$	a	$\bot$	g	a	g	a	g	g	g
d	$\bot$	$\bot$	b	$\bot$	b	$\bot$	b	$\bot$	b	b
e	$\bot$	a	$\bot$	g	a	g	a	g	g	g
f	$\bot$	$\bot$	b	$\bot$	b	$\bot$	b	$\bot$	b	b
g	$\bot$	a	$\bot$	g	a	g	a	g	g	g
h	$\bot$	a	b	g	d	g	d	g	$\top$	$\top$
$\top$	$\bot$	a	b	g	d	g	d	g	$\top$	$\top$

For $\mathbf{A}_3$ and $\mathbf{A}_4$:

$\cdot$	$\bot$	a	b	c	d	e	f	g	h	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
a	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	g	$\bot$	g
b	$\bot$	$\bot$	b	$\bot$	b	$\bot$	b	$\bot$	b	b
c	$\bot$	$\bot$	$\bot$	g	$\bot$	g	$\bot$	g	g	g
d	$\bot$	$\bot$	b	$\bot$	b	$\bot$	b	g	b	$\top$
e	$\bot$	$\bot$	$\bot$	g	$\bot$	g	$\bot$	g	g	g
f	$\bot$	$\bot$	b	$\bot$	b	$\bot$	b	g	b	$\top$
g	$\bot$	$\bot$	$\bot$	g	$\bot$	g	$\bot$	g	g	g
h	$\bot$	$\bot$	b	g	b	g	b	g	$\top$	$\top$
$\top$	$\bot$	$\bot$	b	g	b	g	b	g	$\top$	$\top$

$\cdot$	$\bot$	a	b	c	d	e	f	g	h	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
a	$\bot$	$\bot$	$\bot$	g	$\bot$	g	$\bot$	g	g	g
b	$\bot$	$\bot$	b	$\bot$	b	$\bot$	b	$\bot$	b	b
c	$\bot$	$\bot$	$\bot$	g	$\bot$	g	$\bot$	g	g	g
d	$\bot$	$\bot$	b	g	b	g	b	g	$\top$	$\top$
e	$\bot$	$\bot$	$\bot$	g	$\bot$	g	$\bot$	g	g	g
f	$\bot$	$\bot$	b	g	b	g	b	g	$\top$	$\top$
g	$\bot$	g	$\bot$	g	g	g	g	g	g	g
h	$\bot$	$\bot$	b	g	b	g	b	g	$\top$	$\top$
$\top$	$\bot$	g	b	g	$\top$	g	$\top$	g	$\top$	$\top$

For $\mathbf{A}_5$ and $\mathbf{A}_6$:

$\cdot$	$\bot$	a	b	c	d	e	f	g	h	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
a	$\bot$	$\bot$	b	$\bot$	b	$\bot$	b	$\bot$	b	b
b	$\bot$	c	$\bot$	c	c	c	c	g	c	g
c	$\bot$	$\bot$	b	$\bot$	b	$\bot$	b	$\bot$	b	b
d	$\bot$	c	b	c	h	c	h	g	h	$\top$
e	$\bot$	$\bot$	b	$\bot$	b	$\bot$	b	$\bot$	b	b
f	$\bot$	c	b	c	h	c	h	g	h	$\top$
g	$\bot$	$\bot$	b	$\bot$	b	$\bot$	b	$\bot$	b	b
h	$\bot$	c	b	c	h	c	h	g	h	$\top$
$\top$	$\bot$	c	b	c	h	c	h	g	h	$\top$

$\cdot$	$\bot$	a	b	c	d	e	f	g	h	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
a	$\bot$	$\bot$	c	$\bot$	c	$\bot$	c	$\bot$	c	c
b	$\bot$	b	$\bot$	b	b	b	b	b	b	b
c	$\bot$	$\bot$	c	$\bot$	c	$\bot$	c	$\bot$	c	c
d	$\bot$	b	c	b	h	b	h	b	h	h
e	$\bot$	$\bot$	c	$\bot$	c	$\bot$	c	$\bot$	c	c
f	$\bot$	b	c	b	h	b	h	b	h	h
g	$\bot$	$\bot$	g	$\bot$	g	$\bot$	g	$\bot$	g	g
h	$\bot$	b	c	b	h	b	h	b	h	h
$\top$	$\bot$	b	g	b	$\top$	b	$\top$	b	$\top$	$\top$

Direct calculation verifies that

1. $(\cdot\wedge)$,$(\wedge\cdot)$,$(\backslash\vee)$,$(/\wedge)$,$(\wedge\backslash)$ are true for $\mathbf{A}_1$, but not $(\vee/)$
2. $(\cdot\wedge)$,$(\wedge\cdot)$,$(\vee/)$,$(/\wedge)$,$(\wedge\backslash)$ are true for $\mathbf{A}_2$, but not $(\backslash\vee)$
3. $(\cdot\wedge)$,$(\wedge\cdot)$,$(\backslash\vee)$,$(\vee/)$,$(\wedge\backslash)$ are true for $\mathbf{A}_3$, but not $(/\wedge)$
4. $(\cdot\wedge)$,$(\wedge\cdot)$,$(\backslash\vee)$,$(\vee/)$,$(/\wedge)$ are true for $\mathbf{A}_4$, but not $(\wedge\backslash)$
5. $(\wedge\cdot)$,$(\backslash\vee)$,$(\vee/)$,$(/\wedge)$,$(\wedge\backslash)$ are true for $\mathbf{A}_5$, but not $(\cdot\wedge)$
6. $(\cdot\wedge)$,$(\backslash\vee)$,$(\vee/)$,$(/\wedge)$,$(\wedge\backslash)$ are true for $\mathbf{A}_6$, but not $(\wedge\cdot)$

□

Remark 2.21. By 'direct calculation', we do not necessarily mean a manual one, but rather a computer-assisted one, since the tables, while not prohibitively huge to exhibit,

can still be a bit too complex to be treated with paper and pencil. See more details on that in the next section.

Remark 2.22. The underlying lattice of all our counter-examples (depicted in Figure 3) is non-modular. We tried to find similar counter-examples for the modular case but did not get anything for the implications mentioned in the theorem 2.19, even after gradually increasing the model size to 14 and running servers for several days. The theorem 2.19 might generalise to the modular lattices (see Definition 2.10). We observed similar behaviour when adding a multiplication associativity condition instead of lattice modularity (none of the multiplication tables found is associative).

Remark 2.23. The multiplication table of $\mathbf{A}_5$ is a transposition of one for $\mathbf{A}_6$. It is not surprising since they serve to deny similar equations $((\cdot\wedge)$ and $(\wedge\cdot)$ respectively), but we do not know what this fact might suggest.

We obtained Theorem 2.20 with the help of Nitpick [16], a highly efficient tool for the construction of finite counter-examples packaged with the Isabelle proof assistant [80]. We can only guess why it worked for us where MACE4 failed. It might be related to general progress in the field of finite model search done in the last decade. The version of MACE4 usually run by working mathematicians dates back to December 2007 [3]. Since then, the Paradox [22] system introduced so-called static symmetry reduction, a technique reducing the number of isomorphic models (see [11] for MACE4 and Paradox comparison). Later, Kodkod (see [137] for realisation details and comparison with Paradox) brought sparse representation of binary relations and even more symmetry-breaking schemes to the process of translating a model-search task into a propositional satisfiability (SAT) problem. Nitpick serves as a translator from Isabelle language to Kodkod, which relied (in 2021) on Jingeling ([14], the winner of SAT 2020 competition [32]). Our work exploits the Isabelle server implementation ability to run Nitpick tasks in parallel, yielding an environment for countermodel search with impressive computational advantages. Namely, we conducted our computational experiments yielding Theorem 2.20 on three Linux machines, the largest having 180 CPU cores (INTEL® XEON® Gold 6254 3.10GHz) and 832 GB of RAM, totalling to about two weeks of wall-clock time.

2.4 Python client for Isabelle server

2.4.1 General description

Isabelle interactive theorem prover (ITP) has included the Isabelle server as part of its standard distribution since 2018 [144]. The Isabelle server enables users to run multiple sessions and manage concurrent tasks to process Isabelle theory files through TCP. It makes, in principle, possible to communicate with the Isabelle server using any popular programming language [136], including Python. Python clients already existed for other

[3] https://www.cs.unm.edu/~mccune/mace4/gui/v05.html

major ITPs, for example, one [105] for Lean [78] or another one [99] for Coq [134]. Despite existing projects where Python and Isabelle were used together (see, e.g. [28]), there was no stand-alone and reusable Python client available.

The client relies on a standard Python package `asyncio` for low-level communication with the server. It implements wrapper methods for all commands of Isabelle server listed in its manual [145]. The package also includes a function for starting the Isabelle server from a Python script.

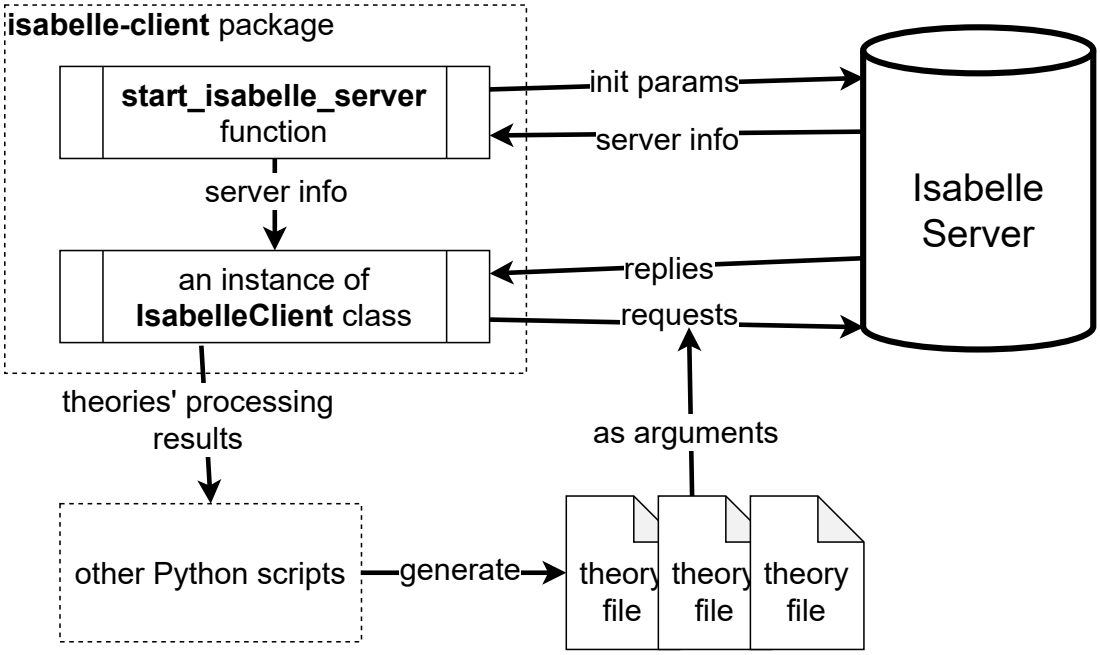

Figure 4: In a project where Python scripts generate Isabelle theory files, one can use the `isabelle-client` to start the Isabelle server, send these files for parallel processing and get back the results.

The package is available for Python 3.7+ on GNU/Linux and Windows. Every new build is tested in a continuous integration workflow against each supported Python version. Detailed documentation pages are built automatically, and the code is nearly 100% covered with tests. The package is hosted now not only on the Python Package Index (PyPI) but also on Conda Forge [23], which enables its installation with both `pip` and `conda` package managers. In addition, one can run the client inside a Docker container, for example, in a cloud using Binder [93]. This option provides the client coupled with the Isabelle server and is particularly useful for students specialising in logic who do not necessarily have much experience in information technologies.

2.4.2 Usage example

Figure 4 represents a typical scenario of the `isabelle-client` application. First, one should start the Isabelle server, for example, using a utility function `start_isabelle_server`

from the package. Second, one creates an instance of `IsabelleClient` object, e.g. using the factory function `get_isabelle_client`. Finally, one can issue any command supported by the Isabelle server (see section 4.4 in [145] for a complete list) using `IsabelleClient` object, which implements all these commands as methods. Usually, these commands rely on existing Isabelle theory files, for example, generated by third-party Python scripts (not by `isabelle-client`). See also listing 1 for a basic code snippet.

```
""" An example of the ``isabelle-client`` usage """
from isabelle_client import get_isabelle_client, start_isabelle_server

# first, we start Isabelle server
server_info, _ = start_isabelle_server(
    name="test", port=9999, log_file="server.log"
)
# then we create an ``IsabelleClient`` instance
isabelle = get_isabelle_client(server_info)
# now we can send theory files to the server and get a response
isabelle.use_theories(theories=["Example"], master_dir=".")
# or we can build a session document using ROOT and root.tex files
isabelle.session_build(dirs=["."], session="examples")
isabelle.shutdown()
```

Listing 1: How to use the `isabelle-client`.

2.5 Software solution architecture

2.5.1 General description

`isabelle-client` per se can not produce Theorem 2.20. It only serves as a part of larger script (`use_nitpick.py` from [116]) which namely does the following:

1. generate theory files templates (all possible combinations of statements studied)
2. fix the model cardinality (at least 2 for the lattice-based structures)
3. generate theory files from the templates by adding a Nitpick task for the cardinality fixed at step 2 (the task here can also be a Sledgehammer task for finding automated proofs rather than finite counter-examples)
4. pass the theory files to Isabelle server and store replies in a dedicated folder (here we use `isabelle-client`)
5. extract models found (if any) from server logs and store them in the result folder
6. remove the theory templates for which there were the models found

7. if there are still theory templates (with no finite models found) and we have not reached the cardinality limit, go to step 2

One can notice that we studied a question of axiom independence. Indeed, since we know that equations from $(\cdot\wedge)$-$(\wedge\backslash)$ are independent of the axioms of residuated binars, we could imagine defining a new algebraic structure where $(\cdot\wedge)$-$(\wedge\backslash)$ are additional axioms. Then Theorem 2.20 states that the axioms of this new algebraic structure are mutually independent. Using finite model finders for axiom independence is nothing new (see, for example, [21] for application of Paradox to establishing the independence of axioms of algebraic structures serving as models of fuzzy logics). Contrary to our work, the authors of [21] did not publish the code to reproduce their results.

At first, not only we intended to find counter-examples showing independence of $(\cdot\wedge)$-$(\wedge\backslash)$, but also we hoped to use automated theorem provers to prove that some statements follow from combinations of others under certain conditions (similar to the E prover usage in [21]). For example, we hope that lattice modularity 2.8 can establish implications from Theorem 2.19 even without lattice distributivity 2.7. Unfortunately, provers shipped with Isabelle (including E and Vampire) could not do anything with the propositions we studied, but we kept related scripts in the final version for anyone interested.

2.5.2 Theory templates generation

Since we have six statements $(\cdot\wedge)$-$(\wedge\backslash)$, we can construct $(2^5 - 1) \times 6 = 186$ different implications between them. One can argue that if we want to prove the independence of six statements, it is enough to find counter-examples for six 'five statements imply the one rest' propositions. And if anyone knew from the beginning that the six statements $(\cdot\wedge)$-$(\wedge\backslash)$ were, in fact, independent, yes, we could do that way. But since there were opposite opinions on whether Theorem 2.20 might have held, we had nothing but to start from scratch. It was not improbable that for some implications (e.g. those from Theorem 2.19), we could never find any reasonably-sized counter-examples. And it was the case for multiplication associativity or lattice modularity in our other experiments. In addition, propositions of form 'five statements imply the one rest' took incomparably longer than 'three statements of six imply one of six'. The more statements in the antecedent of the implication - the longer was search process. So, with a clean slate, it was not unreasonable to test all the 186 possibilities starting from the simplest ones.

One can verify the generated templates' correctness by passing them to the Isabelle server with a dummy task `oops` instead of Nitpick or Sledgehammer. It will make Isabelle check the syntax of theory files and the validity of import statements in them.

2.5.3 Storing and postprocessing the finite models

Isabelle server returns its logs in JSON format. The actual values inside these parcels can have varying formats from one version of Isabelle to another. We had to parse them using the regular expressions library in Python to get the binary and unary operations tables. In such form, we can store them on the disk in JSON or `pickle` (Python binary object) format. Then we packed these tables into objects of Python classes representing lattices and other structures and programmed procedures to extract lattice partial order relationships from the join and meet operations tables. After all these transformations, we wrote scripts to generate LaTeX code for Cayley tables displayed in Theorem 2.20 proof and used Graphviz [37] Python wrapper for drawing Hasse diagrams (e.g. that in Figure 3).

2.5.4 Found models verification

Nitpick sometimes can find spurious examples (it even prints a disclaimer about it). Of course, no software is bug-free, so one should verify with another system any counter-examples found by one system. Fussner, Simpson, and Shminke all independently verified that counter-examples mentioned in the proof of Theorem 2.20 were indeed:

- satisfying residuated binars axioms
- satisfying all but one propositions from $(\cdot\wedge)$-$(\wedge\backslash)$

Simpson and Shminke have open-sourced the scripts of their two independent checkers so anyone can reproduce the result. Unfortunately, we are not aware of researchers not having a conflict of interests (including anonymous reviewers of our previously published work) who had verified the counter-examples proving Theorem 2.20.

2.6 Conclusion and future work

The lessons learned from this project often repeated the known data science folklore. Nevertheless, we realised that:

- one can solve some open research problems in seemingly abstract and non-computational subfields of mathematics only by using more computing power (especially parallel computations) or contemporary software
- although working mathematicians never stopped using provers and finite model finders in their work [56] and the automated reasoning researchers never stopped writing new and updating older software, the latest advances of the latter often are not reusable by the former [4]

[4] "Mathematicians like a lot your tools! You should invest a little bit... a lot in your tools, so that we could use them, we could prove theorems, and all world would be much happier." (João Araùjo at the 7th Conference on Artificial Intelligence and Theorem Proving on September 6, 2022)

- one often does not need machine learning to make artificial intelligence work
- mathematicians need not only counter-examples, the search for which is easily parallelisable but also (and maybe even more) to get actual proofs. And we can not reduce the provers computational improvements to parallel computations (e.g. in Prover 9, the main loop is sequential by its nature)

Addressing these testimonies, we kept our research results as reusable as possible. For example, apart from its original application to discover Theorem 2.20, we used the `isabelle-client` running in a Docker container on Binder during the practical sessions of the Advanced Logic course taught at the Université Côte d'Azur in the autumn of the 2021-2022 academic year. The client helped students not trained in functional programming languages used for Isabelle development (Scala and StandardML) to concentrate on understanding the Isabelle language syntax and consequently generating theory files with Python scripts without installing and running the Isabelle GUI on their laptops. Also, a maintainer of the 'Proving for Fun' backend [49] notified us they were using the `isabelle-client` for debugging and suggested several technical improvements.

We examine how machine learning can improve finite models search in Chapter 3 of this thesis.

We also present our research on applications of artificial intelligence techniques to automated provers in Chapters 4 and 5.

CHAPTER 3

Neural Networks for Model Generation

This chapter describes a collaborative project of application of machine learning tools (namely, artificial neural networks) to the finite algebraic structures study. We previously submitted the work as a pre-print [9] to which the thesis author contributed a Python package for generating finite semigroup with deep learning networks.

This project was the first one after this thesis author received funding from the Interdisciplinary Institute for Artificial Intelligence and the first attempt to work with a working mathematician in a domain that one might consider unrelated to deep learning. We did not arrive at any conclusive results, but we laid the foundation for a better understanding of the relations of our respective fields of study and, even more important, for future work in this direction.

Outline of the chapter:

In Section 3.1, we remind several additional concepts from abstract algebra needed to understand the problems treated in this chapter.

In Section 3.2, we remind the basics of deep learning.

In Section 3.3, we formulate a problem of generating models of finite algebraic structures by deep learning networks and discuss challenges it poses if compared to other ways to incorporate machine learning into finite model search and to other domains where generatve deep learning networks show impressive results.

In Section 3.4, we describe the precise setup of our experiments with a proof of concept implementation of principles discussed in the previous section, including neural network architecture and training techniques applied. We also present the results of the experiments of generating finite semigroups with deep learning networks.

In Section 3.5, we discuss possible generalisations of proposed approaches, their known limitations and open problems worth further investigation.

3.1 Additional algebraic notions

Definition 3.1 (Quasigroup)**.** A *quasigroup* $(A, \cdot, /, \backslash)$ is an algebraic structure where $(A, \cdot)$ is a binar (see Definition 2.1) and for every $x, y \in A$ the following laws hold:

$$x \cdot (x \backslash y) = y \tag{$\cdot\backslash$}$$

$$x \backslash (x \cdot y) = y \tag{$\backslash\cdot$}$$

$$(y \cdot x) / x = y \tag{$\cdot/$}$$

$$(y / x) \cdot x = y \tag{$/\cdot$}$$

Remark 3.2. In a quasigroup, for every $a, b \in A$, there is a unique solution for an equation $a \cdot x = b$ (namely $x = a \backslash b$), and also there is a unique solution for an equation $x \cdot a = b$ (namely $x = b / a$).

Remark 3.3. One can define a quasigroup by giving only a multiplication table for $\cdot$ — the operation tables for $/$ and $\backslash$ will follow from it.

Example 3.4. A multiplication table for $\cdot$ always is a *Latin square*: each element of A is present in each row and each column of it exactly once. This observation gives us many examples of quasigroups, e.g. one depicted in Figure 5.

$\cdot$	e	a	b	c
e	e	a	b	c
a	a	e	c	b
b	b	a	e	a
c	c	b	c	e

Figure 5: Multiplication table of a quasigroup of 4 elements (so-called *Klein Vierergruppe*)

Definition 3.5 (Semigroups equivalence)**.** A *homomorphism* of semigroups (see Definition 2.2) $(A, \cdot) \to (B, \cdot)$ is a map of sets $h : A \to B$ such that for all $x, y \in A$, one has $h(x \cdot y) = h(x) \cdot h(y)$. If h is one-to-one map (bijection), h is called *isomorphism*. If, on the contrary, for all $x, y \in A$ we have $h(x \cdot y) = h(y) \cdot h(x)$, and h is a bijection, we call h an *anti-isomorphism*. If there is either isomorphism or anti-isomorphism between semigroups $(A, \cdot)$ and $(B, \cdot)$, we call such semigroups *equivalent*.

Definition 3.6 (Variety)**.** A *variety of algebras* is the class of all algebraic structures having the same number and arities of operations and satisfying a given set of equations.

Example 3.7. Semigroups, quasigroups (see Definition 3.1), and lattices (see Definition 2.6) are all varieties since we defined them by equations. Residuated binars (see Definition 2.13) are also a variety since there exists an equivalent definition by a list of equational axioms defining it instead of residuation propertry 2.9 we used.

3.2 Deep Learning Basics

3.2.1 Why use deep learning and how

Definition 3.8 (Deep Feedforward Network)**.** A *deep feedforward network* ([42], Chapter 6; or *fully connected* network) is a function $F : \mathbb{R}^n \to \mathbb{R}^{n'}$ defined by the following computation procedure

$$\begin{aligned} \forall x \in \mathbb{R}^n \quad & F(x) = \sigma(W_h a_h + b_h) \\ \forall 1 < i < h \quad & a_i = \sigma(W_{i-1} a_{i-1} + b_{i-1}) \\ \forall x \in \mathbb{R}^n \quad & a_1 = \sigma(W_1 x + b_1) \end{aligned}$$

where

- for all $1 \leq i \leq h$ we call a function $a_{i-1} \mapsto \sigma(W_{i-1} a_{i-1} + b_{i-1})$ a *hidden layer*
- the matrices $W_i \in \mathbb{R}^{l_{i-1} \times l_i}$ (we set $l_0 = n, l_h = n'$) are *weights*
- the vectors $b_i \in \mathbb{R}^{l_i}$ are *biases*
- the function $\sigma : \mathbb{R} \to \mathbb{R}$ (applied coordinate-wise to vectors) is an *activation function*
- the vectors a_i are *activations*
- items of weights and biases together are called *network parameters*, and we will denote them by $\theta = \{W_i, b_i\}_{i=1}^h$
- the number of hidden layers h is called a network *depth*
- for all $1 \leq i \leq h$, we call l_i hidden layer sizes and the maximum of them we denote by m and call the *width* of the network

Theorem 3.9 (Universal Approximation Theorem for Width-Bounded ReLU Networks, Theorem 1 from [70])**.** *For any Lebesgue-integrable function $f : \mathbb{R}^n \to \mathbb{R}$ and any $\varepsilon > 0$, there exists a fully-connected network F with* ReLU *activation $\sigma(x) = \max\{x, 0\}$ and with width $m \leq n + 4$, such that*

$$\int_{\mathbb{R}^n} |f(x) - F(x)| \, dx < \varepsilon \tag{3.1}$$

Remark 3.10. One builds the Theorem 3.9 proof in such a way that one has to increase the depth of the network for smaller ε, thus we speak of *deep neural networks (DNNs)* which often give us better approximations in practice than so-called shallow ones (having fewer hidden layers).

Often, to find a DNN F approximating observed data, we introduce so-called *loss function* $L : \mathbb{R}^{n'} \times \mathbb{R}^{n'} \to \mathbb{R}$ and try to solve the following optimisation task:

$$\max_{\theta} \sum_{j=1}^{N} L\left(F\left(x_j\right), y_j\right) \tag{3.2}$$

where the set $\{(x_j, y_j)\}_{j=1}^{N}$ is a *training set*, each (x_j, y_j) is a *data point*, and y_j are *(ground truth) labels*. L somehow measures the closeness of its arguments. Such task is called *supervised learning* (see more in [104], Chapter 19) since there is a 'teacher' who gives labels to the network to learn from, so the input data x_j is *labelled* and thus somehow structured by forces external to the network.

Since the sum in 3.2 can contain too many addends (for example, a popular ImageNet dataset [103] has more than 14 million images), it is impractical to optimise it as a whole. Instead, one usually samples *batches* (relatively small subsets, usually of several dozens) of datapoints and does the gradient descent steps on them (see *stochastic gradient descent (SGD)* in [110], Chapter 14). After we fix a batch size, we can talk about the whole batch as a new random variable $\mathbf{x}$ whose values belong to $\mathbb{R}^{b \times n \times n'}$ where b is the batch size, and $\mathbf{x}$ includes both all x_j and all y_j for a given batch. Then we can say we optimise $L(\theta, \mathbf{x})$ by varying θ.

3.2.2 Autoencoders

We do not always need labels to extract useful information from our data. If only some raw data points x_j are labelled (exist in a pair (x_j, y_j)), but we still want to be capable of labelling any x_j, we talk about *semi-supervised learning*. If there are no labels at all, and we do not have any structure of data in mind, we talk about *unsupervised learning* (e.g. clustering or anomaly detection tasks [104], Chapter 19). On the contrary, if we want to instil a particular structure we pre-suppose to exist in the data (e.g. that there are such and such labels, but hidden from us), we talk about *self-supervised learning*. A prominent case of self-supervised models is an *autoencoder* where instead of $L\left(F\left(x_j\right), y_j\right)$ in 3.2 we use $L\left(F\left(x_j\right), x_j\right)$. In other words, an autoencoder tries to get its original input after applying a series of non-trivial transformations. Of course, a well-trained DNN of an autoencoder per se is useless, so we look not at the output but rather at activations of a particular hidden layer, which usually serves as a representation of input (often, but not necessarily, of smaller dimension).

In practice, we rarely train simple autoencoders but rather *denoising* ones. Suppose our data points are distorted images (some parts of a photo were time-corrupted, lost during digitalisation, et cetera), and we want to reconstruct the original ones. To model this process, we take a batch $\mathbf{x}$ of undistorted images, then apply different types of corruption to them randomly to get $\bar{\mathbf{x}}$ and treat $\bar{\mathbf{x}}$ as a raw data point to label and $\mathbf{x}$ as the label. Again, we can apply SGD to $L\left(\left(\bar{\mathbf{x}}, \mathbf{x}\right), \theta\right)$, since randomness (or noise) exists

outside the set of network parameters θ. Independent of their original reconstructing ability, denoising autoencoders are known to much better capture the structure of the data than basic ones (with no noise, where input is the label). See Chapter 14 of [42] for more details and an extended bibliography.

3.2.3 Deep adversarial networks

Lately, DNNs have become famous for generating deceptively realistic images of objects and creatures never being in existence [60]. The basis of this technology is so-called *generative adversarial nets (GANs)* [43]. One formulates a GAN as a zero-sum game of two players, a *generator* and a *discriminator*. A generator samples pseudo-observations $\mathbf{x} = g(\mathbf{z}; \theta_g)$ where g is a deterministic function (modelled by a DNN), θ_g are parameters of the DNN g, and z is a random variable (*noise* or *source of randomness*). Instead of considering part of parameters of g to be random variables with a given probability distribution, we say that probability distributions parameters are also among deterministic parameters of g, and all the randomness we have (we denote it z) is input to g (although there is no real input to g from any dataset). Such an approach is called a *reparametrisation trick* and enables us to continue using SGD for the generator network as if the source of randomness was not in the network weights but came from the data batching process.

A discriminator plays with a network $d(\mathbf{x}; \theta_d)$ where d is a DNN, θ_d — its parameters, and $\mathbf{x}$ is data coming either from real-world data distribution $p_{\texttt{data}}$ or from a distribution of samples produced by the generator $p_{\texttt{gen}}$. Each turn, with probability $\frac{1}{2}$ one samples either from real-world data, point $\mathbf{x} \sim p_{\texttt{data}}$ or asks a generator to produce a fake one $\mathbf{x} \sim p_{\texttt{gen}}$. Then, the discriminator computes $d(\mathbf{x})$ to evaluate a probability of $\mathbf{x}$ being real. If it guesses, it wins, and the generator wins otherwise. Hence, the payoff of the discriminator (log-likelihood of data coming from different distributions) is

$$v(\theta_g, \theta_d) = \mathbb{E}_{\mathbf{x} \sim p_{\texttt{data}}} \log d(\mathbf{x}) + \mathbb{E}_{\mathbf{x} \sim p_{\texttt{gen}}} (1 - \log d(\mathbf{x})) \tag{3.3}$$

and the payoff of the generator is $-v(\theta_g, \theta_d)$.

Usually, after training a pair of a generator and a discriminator, the latter is discarded since there is no guarantee it could discriminate fake data coming from other distributions (produced by other generators, not trained in a couple with it). In principle, a discriminator can be an oracle rather than a neural network we train side-by-side with the generator. Such approaches (with both generator and discriminator doubled by oracles) were recently shown [91] to demonstrate better convergence properties and final DNN qualities than the original GAN formulation.

3.3 Generating algebraic structures with deep learning networks

3.3.1 Similar tasks

As seen in Chapter 2, using intelligent software to find finite models of logic theories can help working mathematicians prove theorems otherwise unreachable. A contemporary piece of software producing counter-examples is usually called an SMT (satisfiability modulo theory) solver (e.g. Z3 [25]). Deep neural networks are sometimes applied as parts of well-known model search algorithms instead of search-guiding heuristics, making the search process faster (see, e.g. `fastSMT`[8]). Even in less general cases of generating only semigroups [118], authors take a similar approach. On the other hand, the world has recently seen an overwhelming success of deep neural networks generating all sorts of objects, from images (e.g. StyleGAN [61]) to texts (see [54] for a survey), based on expected qualities of these objects (i.e. their style in case of works of art). It begs the question: can we create a deep neural network (see Definition 3.8) generating algebraic structures as a whole instead of serving only as a part of a generation algorithm? For example, can we build a neural network whose outputs are the whole semigroup multiplication tables?

3.3.2 Denoising as a simpler task

This thesis author got the inspiration from a project [87] of solving a popular puzzle game of *sudoku* using convolutional neural networks (CNNs, [42], chapter 9). From an algebraist point of view, solved sudoku is a multiplication table of a quasigroup (see Definition 3.1) of 9 elements with some additional constraints. Since such multiplication tables have properties easily identifiable by the naked eye (they are Latin squares; see Example 3.4), we can consider solved sudoku as an image of 9×9 pixels with a colour channel having values from 1 to 9. Then a sudoku to solve is the same image but with some hidden pixels (e.g. colour set to black or 0). The task of computing the missing pixels' colour is well known in computer vision and called *(image) denoising*. CNN denoise well both images and quasigroup multiplication tables (sudokus). Even if not generated from scratch, can we at least denoise semigroup multiplication tables as well as it worked for sudokus?

3.3.3 Suitable neural network type

A semigroup multiplication table has no evident visual marks easily captured by the human eye. In semigroups, the order of the elements is arbitrary, and permuting them in any way gives an equivalent (see Definition 3.5) semigroup. It means that using CNNs relying on topological structures (like lines and shapes in an image) might not be as applicable, but more general neural network architectures for denoising any input

signal type exist, namely denoising autoencoders (see Subsection 3.2.2 or [42], chapter 14 for more).

Table 3.1: Number of semigroups up to equivalence ([26, 27]).

Cardinality	# of semigroups
1	1
2	4
3	18
4	126
5	1,160
6	15,973
7	836,021
8	1,843,120,128
9	52,989,400,714,478
10	12,418,001,077,381,302,684

3.3.4 Training data

To train a denoising autoencoder, one should first get a dataset of complete examples and then add random noise as part of the training process. For sudoku and photos, there are algorithms to generate and solve a sudoku, and photographs of any kind are available in abundance nowadays. For semigroups, the situation is a bit different. On the one hand, semigroups of sizes up to and including 8 are catalogued and freely available as a part of `smallsemi` package [26] of GAP system [135]. The semigroups of sizes 9 and 10 are all known [27], but not stored anywhere (it could take from around a terabyte to hundreds of petabytes of disk space; see Table 3.1). The systematisation of the semigroups of more than ten elements at the moment of writing seemed beyond reach.

·	1	2	3	4	5
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	2	1	2
4	1	1	1	2	1
5	1	1	2	1	2

$\Leftarrow$

·	1	2	3	4	5
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1			
4	1	1			
5	1	1			

$\Rightarrow$

·	1	2	3	4	5
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	2	2	2
4	1	1	2	2	2
5	1	1	2	2	2

Figure 6: Different semigroups arising from the same partially filled table

3.3.5 Non-unique ground truth

When denoising images, one expects a neural network to discover the exact original image (ground truth). Although (in general) one can complete an incomplete quasigroup multiplication table in more than one way, one designs sudoku to have only one solution. In case of incomplete semigroups, we either have to ignore the fact of several completions existing (see, e.g. Figure 6) thus punishing the network for finding a

different completion, or allow it to generate something associative, but not necessarily unique. We compared these two approaches in our experiments.

3.3.6 Loss function

A typical GAN (generative adversarial network) (see Subsection 3.2.3) includes two subnetworks: a generator and a discriminator (see, e.g. Figure 7). The generator's output mimics images with desired properties based on random input (noise), and the discriminator tries to distinguish between a currently generated image and a randomly chosen real one.

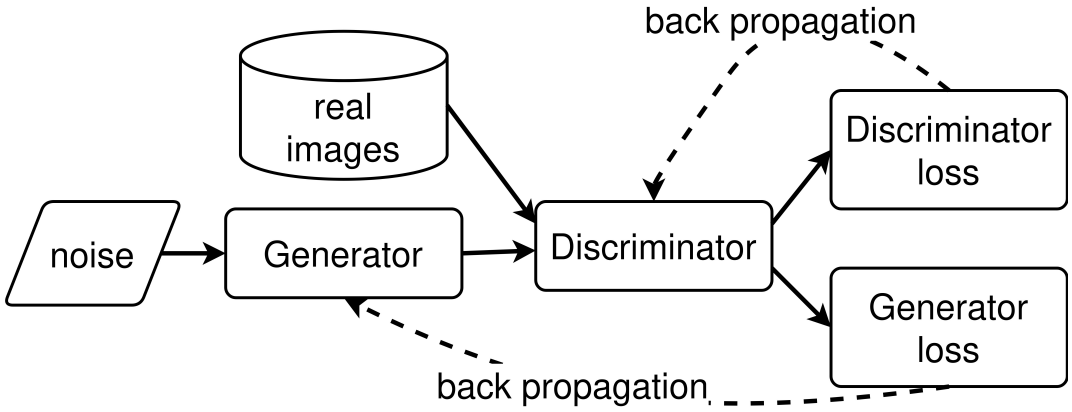

Figure 7: A typical GAN training pipeline

In case of algebraic structures from a given variety (see Definition 3.6), there is no difference between a freshly generated and a "real" (previously catalogued) multiplication tables. The only thing which matters is whether the variety equations hold. So, in contrast to images, we do not need to train a discriminator network since we already have a pre-determined oracle which in addition does not need any non-generated input. Such an AGN (algebraic generative network) architecture is depicted in Figure 8. We can even think of the discriminator-oracle as a part of generator loss function in this case.

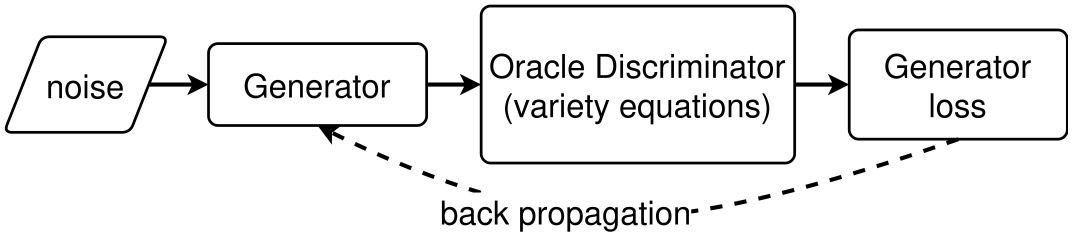

Figure 8: A possible generative network for algebraic structures

AGN looks very similar to a typical denoising autoencoder (DAE, see Figure 9), the difference being that DAE needs non-generated input to generate the output and compute loss.

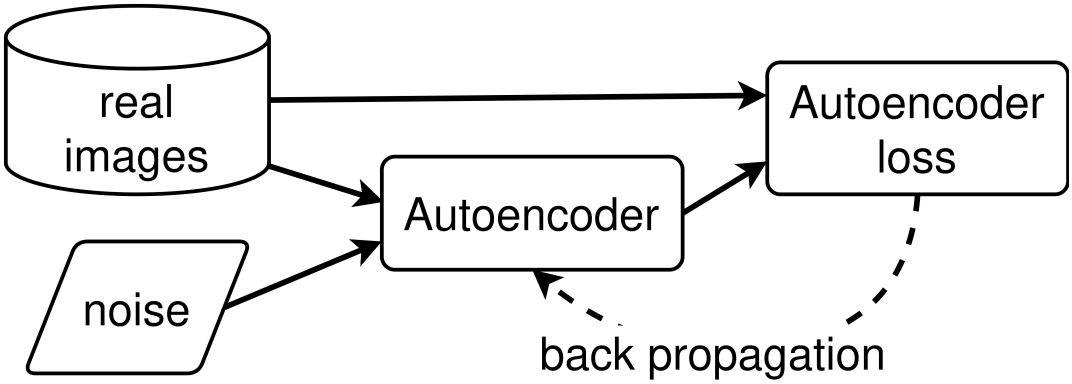

Figure 9: A typical denoising autoencoder training pipeline

This thesis author proposed a combination of DAE and GAN (see Figure 10) for effective algebraic structures generation, which we describe in detail in consequent sections.

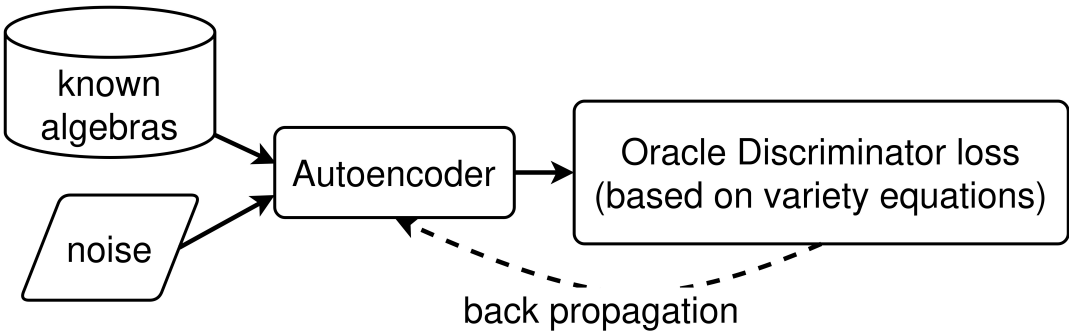

Figure 10: A novel generation process for algebraic structures

3.4 Experiment setup

3.4.1 Data representation

Definition 3.11 (Probabilistic Cayley Table)**.** Given a set $S = \{e_i\}_{i=1}^n$, consider a function $F : S \times S \times S \to [0, 1]$. We want to treat this function as a probability distribution for the potential multiplication: $\mathbb{P}(e_i \cdot e_j = e_k) = F(e_i, e_j, e_k)$. For this to make sense, the function F must satisfy the following condition: $\sum\limits_{k=1}^{n} F(e_i, e_j, e_k) = 1$ for all possible choices of i and j. We call such a function F a *probabilistic Cayley table*. A probabilistic Cayley table F is *filled at* $1 \leq i, j \leq n$ if there exists k such that $F_{i,j,k} = F(e_i, e_j, e_k) = 1$.

Any semigroup structure on S provides the probabilistic Cayley table filled at all i, j:

$$F_{i,j,k} = \begin{cases} 1, & \text{if } e_i \cdot e_j = e_k \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

However, not every Cayley table sampled from the distribution defined by a probabilistic

table F corresponds to an associative multiplication. For this reason, define:

Definition 3.12 (Solvable table)**.** A probabilistic Cayley table (see Definition 3.11) $F : S \times S \times S \to [0, 1]$ is *solvable* if there exists a semigroup structure on S (which we will call a *completion*) described by conditions in form 3.4 which can be sampled from a probability distribution described by F with non-zero probability.

As noted in Subsection 3.3.5, a solvable F can have multiple semigroup completions.

We can store $F_{i,j,k}$ as a tensor (in PyTorch [88] parlance) of one axis of dimension n^3, e.g. using a lexicographical order of triples of indices (i, j, k). We use these tensors as the principal data representation method in this work.

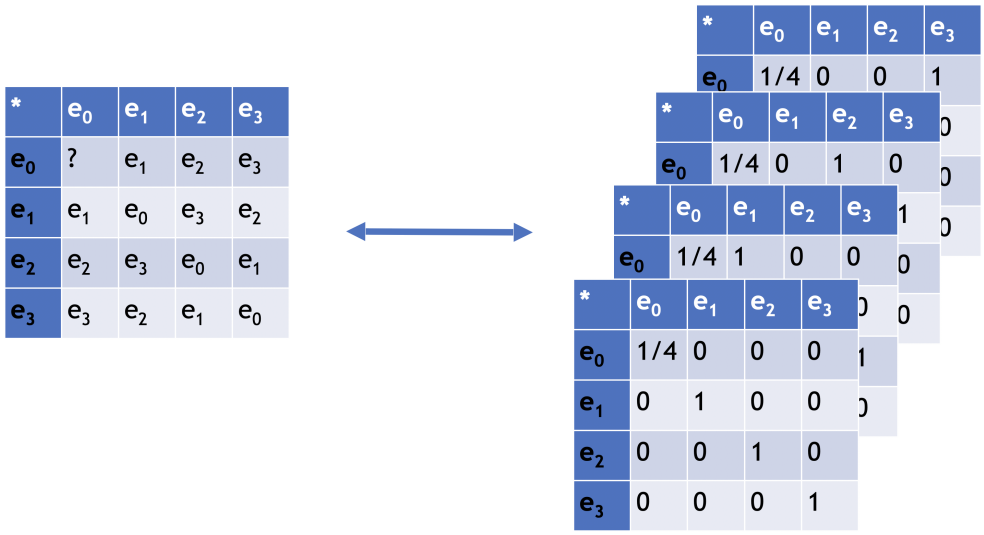

Figure 11: Translating a multiplication table into a partially filled one. $F : \{0, 1, 2, 3\}^3 \to [0, 1]$.

Assume now that for a set S, we specified only some multiplications for a semigroup structure. It still allows us to define the function F. If the result of multiplication $e_i \cdot e_j$ is missing, we can extend by employing a uniform distribution in such cases, i.e. assume in that case that $F_{i,j,k} = \frac{1}{n}$ for all k (Figure 11).

3.4.2 Network architecture

One can consider a probabilistic Cayley table as a result of adding noise to a corresponding filled table. Similarly, we can view arbitrary probabilistic tensors as noisy counterparts of zero-or-one tensors. We can apply a training pipeline of denoising autoencoders (Figure 9) to remove noise and restore the original input.

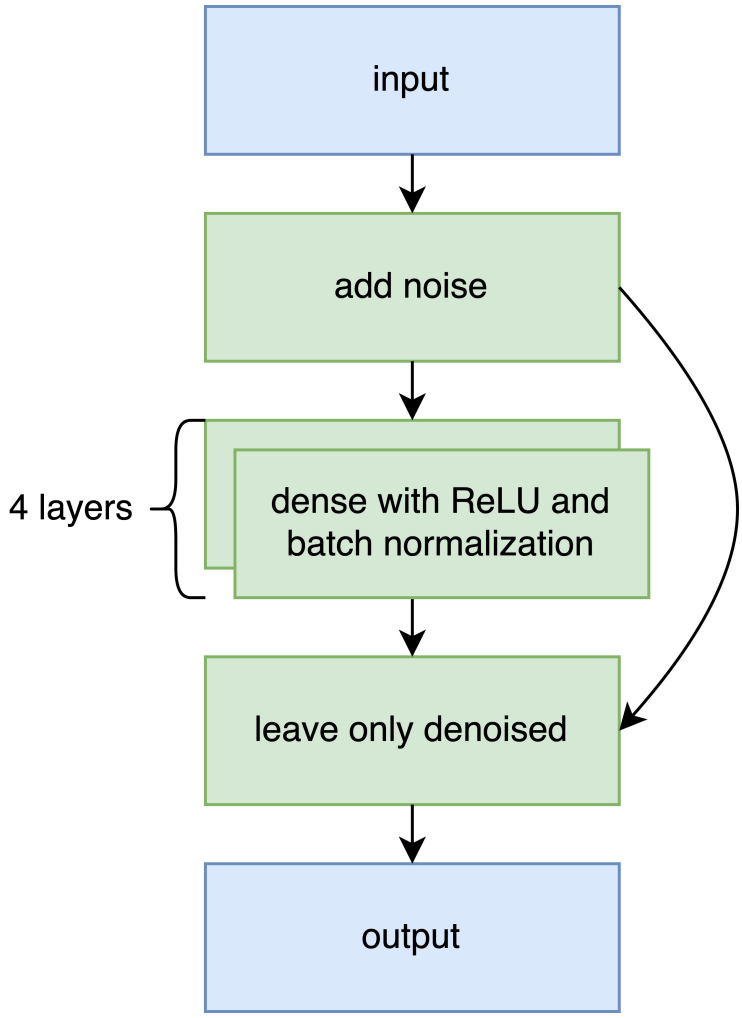

Figure 12: Autoencoder architecture used to generate Cayley tables. The arrow from the input with added noise (masking out cells) to the "leave only denoised" layer corresponds to restoring the values of initially known cells.

For a scheme of the autoencoder architecture used in our experiments, see Figure 12. Besides adding noise to its input, this network also cleans its output of guesses of the cells, which we did not mask during noise addition; these cells correspond to known fillings of the Cayley table. In other words, if the input was filled at i, j so that $e_i \cdot e_j = e_l$ and the output $F_{i,j,k}$ during the forward pass is a float between 0 and 1, it is then redefined as $F_{i,j,k} = 0$ or 1 corresponding to $k \neq l$ or $k = l$.

Another particular thing to note is that we usually have encoders, which move from higher dimensions to lower ones. Here we have the input and output of dimension n^3 and all the hidden layers of the dimension n^5.

3.4.3 Loss functions

If x is an input for an autoencoder and y is its output, we can define its loss function $L(x, y)$ in many ways. Since, in our case, values of x and y are probability distribu-

tions, it could be a good idea to use some measure of dissimilarity between these two distributions, e.g. their Kullback-Leibler divergence (see Chapter 21 of [104]):

$$\mathrm{KL}\,(x, y) = \sum_{i=1}^{n} x_i \log \frac{x_i}{y_i} \tag{3.5}$$

Note that this choice of a loss function does not explicitly enforce any notion of associativity. The problem with this function is that after applying noise to x, one can often recover it as y non-uniquely, yet the loss function $(x, y) \mapsto \mathrm{KL}(x, y)$ will prefer $y = x$ to any other value of y, even if that value is associative.

Another choice of a loss function is what we call the *associator loss*. First, remember that y corresponds to the probability distribution $y_{ijk} = \mathbb{P}\{e_i \cdot e_j = e_k\}$ and all events $e_i \cdot e_j = e_k$ are independent. Then we can calculate probabilities of double multiplications:

$$\begin{aligned}\mathbb{P}\{(e_i \cdot e_j) \cdot e_k = e_l\} &= \sum_{m=1}^{n} \mathbb{P}\{e_m \cdot e_k = e_l | e_i \cdot e_j = e_m\}\, \mathbb{P}\{e_i \cdot e_j = e_m\} \\ &= \sum_{m=1}^{n} y_{mkl} y_{ijm}.\end{aligned}$$

Now we can define the loss function as a KL-divergence between the distributions $\mathbb{P}\{(e_i \cdot e_j) \cdot e_k = e_l\}$ and $\mathbb{P}\{e_i \cdot (e_j \cdot e_k) = e_l\}$:

$$\mathrm{AL}\,(x, y) = \mathrm{KL}\left(\sum_{m=1}^{n} y_{ijm} y_{mkl}, \sum_{m=1}^{n} y_{iml} y_{jkm}\right). \tag{3.6}$$

Remark 3.13. Effectively, such a loss function, independent of the input without noise, corresponds not to a classical DAE (Figure 9) but to a novel training pipeline proposed by this thesis author (Figure 10).

3.4.4 Noise

In our case, the noise the autoencoder is treating corresponds to the absence of some cells in a Cayley table. In our experiments, both for training and testing, we take tables of semigroups of cardinality 5. Given any table $F : S \times S \times S \to \mathbb{R}$ we then add noise by re-setting $F_{i,j,k} = \frac{1}{5}$ for i, j corresponding to randomly chosen 50% of cells of the original Cayley table.

3.4.5 Training and testing datasets

For this work, we used an extensive database of finite semigroups of up to eight elements from [26].

In our experiments, we used semigroups of 5 elements for experiments: this corresponds to 1160 semigroups with 183732 possible Cayley tables. In detail, we divided this set of 1160 equivalence classes into three subsets: training, validation, and testing

in proportion 10/10/80. We then produced all Cayley tables of isomorphic and anti-isomorphic semigroups corresponding to these equivalence classes, a procedure one can view as a data augmentation technique (see Chapter 12 from [42]).

Finally, we applied the noise described in Subsection 3.4.4 to only validation and testing sets. The training set gets its noise during the training process, and 50% of cells for being masked are chosen at random for every batch and not fixed in advance for the training process. Note that all probabilistic tables appearing here are solvable (see Definition 3.12).

3.4.6 Quality metrics

Since we train autoencoders or something resembling GANs, it is natural to use the following metrics:

Definition 3.14. The *guess rate* is the percentage of outputs of a network which coincide with their inputs before applying noise. The *associative rate* is the percentage of outputs of a network which satisfy the associativity condition.

The associative rate appears to be a more mathematically relevant quality metric since we are interested not in the exact reconstruction of inputs but in generating associative tables. It serves as a discriminator loss of a kind. One can complete a half-filled table into different semigroups, but the guess rate will accept only the original table for its score.

3.4.7 Training process

We trained all the networks using the PyTorch [88] framework, using an Adam optimizer [62] with the learning rate set to 0.0001. We performed training for a maximum of 1000 epochs with an early stopping applied if the loss did not go down for ten consecutive epochs. For training hardware, we relied on Google Colaboratory [15] free cloud resources, where the training took several hours. We performed batch normalisation on each layer and used random network parameter initialisation.

3.4.8 Experiment results

First, we note that teaching an autoencoder to reconstruct its exact input without knowing anything about associativity proved to be not only unnatural but bringing poorer results. Even in terms of its principal goal – finding the original table from the input with added noise – the KL divergence loss is less adequate than the associator loss (AL): see Table 3.2 for exact numbers. One way to interpret the KL-AL guess ratio difference might be by observing that the AL network does better at the associativity task overall: in particular, it does better at reconstructing the original table.

Table 3.2: Comparison of impacts of a loss function choice. The associator loss network fares better not only at producing associative tables, but also at guessing the original table to which we applied noise.

Loss function used	Guess rate	Associative rate
KL divergence	0.0977	0.5838
Probabilistic associator loss	0.1453	0.8212

The AL network results are rather promising. We produced a full associative table given only half of the filled cells as an input in 82% of cases. That is even more impressive given that we relied only on 10% of all available tables from the database, thus generalising to 80% (which went to the test set). Our results remain dependant on the choice of these 10% tables for a training set, with deviation representing about 2 per cent of the loss (see Table 3.3 for the details).

Table 3.3: Comparison of impacts of training set choice on guess and associative rates (AL network).

Metric	min	average	max	std deviation
Guess rate	0.1362	0.13979	0.1463	0.0036
Associative rate	0.7878	0.8181	0.8468	0.0187

3.5 Conclusion and future work

In this project, we made the first steps toward a neural network generating finite algebraic structures of a given cardinality and from a given variety. Indeed, one could view the associator loss as the suitable "architectural adaptation" to the case of semigroups: instead of convolutional layers, we are dealing with algebraic equations written into the loss function in probabilistic terms. Moreover, we got 3.6 by applying a probabilistic representation of operators to the associativity law 2.1. So, if we have more operators and more variety-defining equations, we can still construct similar losses. A reasonable research question in this direction is how to combine losses steaming from different equations.

Then we can generate algebraic structures from noise only since we might not have readily available training datasets for varieties other than semigroups. Balzin and Bulić conducted such experiments during the internship of the latter, for which this thesis author served as a consultant. It is worth mentioning that this thesis author finished his work on the code more than half a year before Bulić's contribution (see `semigroups-generation` notebook in the source code [1]). Thus we consider the goal of leaving our research results in the state reusable by others, set in Section2.6, as fulfilled.

Another potentially fruitful direction of research we envision is to not only add equations but also their negations. It can help algebraists find counter-examples similar

[1] `https://github.com/inpefess/neural-semigroups`

to those we got to establish Theorem 2.20. The exact changes to the loss function needed seem to be a viable research question.

Here finishes our research in applications of artificial intelligence to finite algebraic structures study. In the following Chapters (4 and 5), we focus on another fundamental task working mathematicians face every day: finding theorem proofs.

CHAPTER 4

Reinforcement Learning for Automated Theorem Proving

This chapter describes a project of the thesis author dedicated to applying the reinforcement learning (RL) paradigm to automated theorem proving. Parts of this chapter were previously reported at a peer-reviewed international conference [112] and published in a peer-reviewed international journal [111].

We found inspiration for this work in the idea of the HOList [10] environment for machine learning of theorem proving and the lack of resembling environment-ish projects in the world of saturation provers (despite of existence of Deepire [128] and ENIGMA [53] which inspired us immensely). Notably, we wanted to propose such a way of guiding provers by machine learning that we could relatively effortlessly transfer solutions implemented in one to others leading to the mutual enrichment of ideas and research results.

Outline of the chapter:

In Section 4.1, we remind automated theorem proving basics.

In Section 4.2, we remind RL concepts pertinent to our work.

In Section 4.3, we formulate a problem we strive to solve and outline the general architecture of a proposed solution. We make a short overview of related work and explain our design choices.

In Section 4.4, we present an RL environment for conducting experiments with different automated provers with no additional work.

In Section 4.5, we discuss possible future work.

4.1 Automated reasoning basics

4.1.1 Interactive and automated theorem provers

Automated reasoning systems include two prominent types: *interactive theorem provers (ITPs)* and *automated theorem provers (ATPs)*. An ITP strives to find a proof of a math-

ematical theorem semi-automatically by completing the proof steps input by a human, and an ATP tries to generate the whole proof itself with no input but the theorem statement and axioms. Examples of ITPs are Isabelle [80] which we used in Chapter 2, Coq [134], HOL Light [48], HOL4 [119], and Lean [78] among many others. An ITP relies on a human-readable and machine-verifiable formal language to write down theorems and proofs. All the ITPs we mentioned in this chapter use different languages, and for any given pair of such languages, an automatic translator from one to the other usually does not exist (to our best knowledge). It is not only because of the grammatical peculiarities of these languages but also because the ITPs might be using different foundations of mathematics. Each mature ITP usually comes with a library of mathematical statements proven in its formalism, one of the first and the largest among such libraries belonging to the Mizar system [44].

Examples of ATPs are Prover 9 [74] which we used in Chapter 2, Vampire [66], E [108], iProver [29], and Leo III [126], among many others. In contrast to ITPs, all popular ATPs support theorem statement input in a common language — that of the Thousands of Problems for Theorem Provers (TPTP) library [131]. The TPTP does not contain ATP-generated proofs but axioms and theorem statements. Such a common library enables comparison of ATPs by a competition (who solves more problems from the TPTP, in a nutshell), namely the CASC [132].

In principle, an ATP can move from axioms and theorem assumptions, combining them according to inference rules of a chosen deductive system and trying to produce the theorem conclusion. Or it can start with the axioms, the assumptions, and the negated conjecture and infer their logic consequences until (hopefully) reaching falsehood. In these two opposite cases, we talk about *forward proofs* and *proofs by contradiction*, respectively. Nowadays, practically all popular ATPs are of the latter type, so we also talk about *refutation provers*.

In many ITPs, a computer can propose the next step of a proof started by a human or even complete it. Often this functionality relies on ATPs addressed as *hammers* in this situation. A user can also ask the ITP to apply a complex proof state transformation, often called *tactic*, to get to the next step instead of manually inputting it. In principle, one can imagine proof in ITP generated automatically, step by step, by applying suitable hammers or tactics, so, to some extent, ITPs can be viewed as ATPs with much more complex languages and systems of deduction rules.

4.1.2 First-order logic and Clausal Normal Form

In our work, we focused on a particular formal language:

Definition 4.1 (First-order logic). ([104], Chapter 8)
First-order logic (FOL) is a tuple (V, F, P, σ, L) where

- V is a set of *variables*, F — a set of *function symbols*, P — a set of *predicate*

symbols (these sets are, in principle, countably infinite, but in our case, they will be all finite)

- $\sigma : F \cup P \to \mathbb{N}$ ($\mathbb{N}$ is a set of natural numbers including zero) is a function called *arity*. If for a symbol $s \in F$ (or $s \in P$), we have $\sigma(s) = n$, we say that a symbol s has arity n (or is an n-ary symbol). If for some $f \in F$ we have $\sigma(f) = 0$, we call such a function symbol f a *constant*.

- L is a set of well-formed *formulae* of the language recursively defined as follows:

  1. every variable and every constant is a *term*
  2. for every n-ary function symbol f and any n terms $t_1, t_2, \ldots, t_n$, an expression $f(t_1, t_2, \ldots, t_n)$ is a term
  3. for every n-ary predicate symbol p and any n terms $t_1, t_2, \ldots, t_n$, an expression $p(t_1, t_2, \ldots, t_n)$ is an *atom*. Any 0-ary predicate symbol is also an atom. All atoms are formulae
  4. if A is a formula, $\neg A$ is a formula. If A is an atom, then both A and $\neg A$ are called *literals*
  5. if A and B are formulae, then $A \wedge B$, $A \vee B$ are formulae
  6. if A is a formula and x is a variable, then $\exists x A$ and $\forall x A$ are formulae

One can transform every FOL formula to an equisatisfiable formula in so-called *clausal normal form (CNF)*: $C_1 \wedge C_2 \wedge \cdots \wedge C_n$. n depends on a formula at hand, and each formula $C_i, 1 \leq i \leq n$ is called a *clause* and has a form $L_1 \vee L_2 \vee \cdots \vee L_m$. m depends on the clause at hand and each formula $L_i, 1 \leq i \leq m$ is a literal. In CNF, all variables are implicitly universally quantified, i.e. we write $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ instead of $\forall x_1 \forall x_2 \ldots \forall x_N C_1 \wedge C_2 \wedge \cdots \wedge C_n$ where $x_1, x_2, \ldots, x_N$ is a list of all variables occurred in clauses $C_1, C_2, \ldots, C_n$. To get rid of existence quantifiers ($\exists$), we apply so-called *Skolemization* by introducing *Skolem functions*. For example, a FOL formula $\forall x (\exists y f(y) \wedge \neg g(x, y))) \vee (\exists z g(z, x))$ when Skolemized, becomes $\forall x (f(s_1(x)) \wedge \neg g(x, s_1(x)))) \vee (g(s_2(x), x))$. The new symbols introduced s_1, s_2 are Skolem functions representing the existence of variables satisfying the original formula and depending on the variable x from an outer scope. For more details, see [104], Chapter 9.

4.1.3 Given clause algorithm

An implementation of a backbone algorithm for proof search in many ATPs called *given clause* algorithm (Algorithm 1) first appeared in Otter [75], a Prover 9 predecessor. To apply it, one should first pre-process a theorem in the following way:

1. from a theorem $B_1, B_2, \ldots, B_n \Longrightarrow C$ in a theory with axioms $A_1, A_2, \ldots, A_N$, we produce one formula $T = A_1 \wedge A_2 \wedge \cdots \wedge A_N \wedge B_1 \wedge B_2 \wedge \cdots \wedge B_n \wedge \neg C$ (a negation of the theorem in the theory)

2. we transform formula T to the formula $T' = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ (in CNF)
3. we treat T' as a set of clauses $\{C_1, C_2, \ldots, C_m\}$ to use as input to the given clause algorithm

Algorithm 1 Given clause algorithm

Require: input is a set I of clauses
  $P \leftarrow \varnothing$ (processed set)
  $U \leftarrow I$ (unprocessed set)
  while $U \neq \varnothing$ **and** $\bot \notin U$ **do**
    select given clause $g \in U$
    $P \leftarrow P \cup \{g\}$
    $N \leftarrow \varnothing$ (new clauses)
    for all i from inference rules **do**
      use i to infer all possible clauses N_i from g (and clauses from P if needed by i)
      $N \leftarrow N \cup N_i$
    end for
    $U \leftarrow (U \cup N) \setminus \{g\}$
  end while

A `while` loop in Algorithm 1 has an invariant: all possible inferences from clauses in a processed set done (before the iteration starts). At first, the processed set is empty, so the invariant holds trivially, and then it is easy to see how it persists through the loop. It guarantees we will not miss any inference which might bring us to a refutation.

The processed set grows linearly (one given clause each loop iteration), but the unprocessed one grows much faster (folklore based on experimental evidence suggests quadratic law). This property of a given clause algorithm makes it prone to out-of-memory errors if running for a long time.

Remark 4.2 (Portfolios). The time (and the total number of loop iterations) needed to find a proof depends tremendously on how one selects a given clause. Usually, ATPs rely on a clever mixture of heuristics for the selection task. A well-known and easily implementable approach (see, e.g. [109] for educational Python implementation) is to organise two priority queues from the unprocessed set: one contains all the unprocessed clauses sorted by their 'age' (the given clause algorithm step number at which step they appeared among the unprocessed), and the other keeps the same clauses ordered by 'weight' (the number of logic symbols occurred in the clause). Then one defines a strategy consisting of repetitively choosing a given clause from one queue n times and then — choosing it from the other for m following times. Moreover, such strategies can be packed into *portfolios* (see, for example [97]), and a prover might be running the whole portfolio in parallel (with shorter time limits) and guessing which strategy would be the best one given the problem input.

Remark 4.3. ATPs based on the given clause algorithm are also called *saturation provers* (see [101] for more context and precise definition). Not all existing ATPs are

saturation provers. Another popular class of provers are *connection* (or *tableaux*) provers. In contrast to the given clause algorithm, which is inherently sequential, the analytic tableaux method (see Chapter 3 in [47] for more details) is parallelisable. Moreover, it does not need an input formula to be in the CNF, thus eliminating the need for Skolemization. One can exploit these advantages for building ATPs (e.g. Goéland [20]) working better on some problem classes. Nevertheless, tableaux provers rarely made it to the top at the CASC competition, and all the ATPs mentioned in this chapter are saturation ones if not said otherwise.

4.1.4 Deductive systems

The given clause algorithm will not terminate with a guarantee with just any system of inference rules.

Definition 4.4 (Refutation completeness)**.** (see Chapter 3 of [47]) A system of inference rules is *refutation complete* if, for an unsatisfiable set Γ of clauses, it can derive the empty one and verify that unsatisfiability (non-existence of a model in which every clause from Γ holds).

Refutation completeness of an inference rules system (as stated in Definition 4.4) is a necessary condition for the given clause algorithm to terminate.

For example, the following system is known to be refutation complete [17]:

$$\frac{C_1 \vee A_1, C_2 \vee \neg A_2}{\sigma\left(C_1 \vee C_2\right)}, \sigma = mgu\left(A_1, A_2\right) \qquad \text{resolution}$$

$$\frac{C_1 \vee s \approx t, C_2 \vee L\left[r\right]}{\sigma\left(L\left[t\right] \vee C_1 \vee C_2\right)}, \sigma = mgu\left(s, r\right) \qquad \text{paramodulation}$$

$$\frac{C \vee A_1 \vee A_2}{\sigma\left(C \vee A_1\right)}, \sigma = mgu\left(A_1, A_2\right) \qquad \text{factoring}$$

$$\frac{C \vee s \not\approx t}{\sigma\left(C\right)}, \sigma = mgu\left(s, t\right) \qquad \text{reflexivity resolution}$$

where C, C_1, C_2 are clauses, A_1, A_2 are atomic formulae, L is a literal, r, s, t are terms, and σ is a substitution (most general unifier). $L\left[t\right]$ is a result of substituting the term t in $L\left[r\right]$ for the term r at only one chosen position.

The resolution rule itself is refutation complete in languages without the equality symbol. The paramodulation rule is one of the rules that successfully works with equality. Another one, which is a restriction of paramodulation, is called *superposition* [6].

In addition to different rules to produce new clauses, ATP practitioners employ other ones that help to remove unnecessary clauses. A first-order clause C *subsumes* another D, if there is some substitution σ such that the set of literals of $\sigma\left(C\right)$ is a subset of that of the literals of D.

$$\frac{C, \sigma(C) \vee D}{C} \quad \text{subsumption}$$

Namely, *forward subsumption* checks if any processed clause subsumes the given clause and discards the latter if that is the case instead of making other inferences from it. *Backward subsumption* removes processed clauses subsumed by the given clause instead of adding them to the unprocessed set. See more in [109]

For proofs of refutation completeness of more general versions of a given clause algorithm serving as base loops of contemporary ATPs and including different inference rules types mentioned in this section, see [142].

4.1.5 Hints and proof sketches in saturation provers

A *hint* to the given clause algorithm is a clause C such that if there is a clause D among unprocessed ones which subsumes (see subsumption) C or which is subsumed by C, then the algorithm prioritises the clause D for selection as a given one. Hints might come from a *proof sketch* — a list of clauses which form a (hopefully backbone) part of a complete proof (a list of clauses itself). For example, a mathematician might design a proof sketch based on their intuition and previous experience in proving similar theorems or theorems from the same domain. The technique of hints appeared in Prover 9 predecessor Otter [139], and was preserved by Prover 9. Another source of hints might be proof of a weaker theorem, which an ATP can solve contrary to a more general theorem. In this case, the hint list is not a real sketch of a general theorem proof in the strict sense because the clauses-hints do not necessarily appear in the final proof. That is why we spoke about subsumption (instead of the exact equivalence of clauses) and prioritisation rather than taking the hint immediately as a given clause. Working mathematicians describe hints as a "particularly powerful method" [63], and for a reason: they solved many open problems in mathematics with it [140]. E prover further generalised the idea of hint lists, calling them watchlists [102] and not insisting on a particular nature of their source. For example, in the ENIGMAWatch [40] project, authors tried to learn the priority of an unprocessed clause based on its aggregated similarity to clauses from a series of watchlists composed of clauses that appeared in proofs (generated by the same ATP) of theorems from the same domain. It means that one can generate hints from previous experience with an ATP rather than write them manually from the experience of a human mathematician.

4.1.6 TPTP language

The TPTP library [131] language is an impressively expressive one [1] and comprises a multitude of dialects. The most useful for our purpose is one for writing FOL formulae.

[1] `https://www.tptp.org/TPTP/SyntaxBNF.html`

It somehow mirrors Definition 4.1 (notice that alphanumeric characters include the underscore in the TPTP):

- variables are alphanumeric strings starting with a capital letter, e.g. `X` or `Y1`
- function and predicate symbols are alphanumeric strings starting with a lowercase letter, e.g. `join` or `is_subset`
- some function and predicate symbols have special meaning and start with a dollar sign, e.g. `$false` stands for tautological falsehood
- for predicates and functions, the default form is the prefix one. One does not write `X * Y` in the TPTP, but rather `$product(X, Y)`. Nevertheless, the equality and inequality predicates are infix: `X != Y` (for $X \neq Y$) or `X = Y`
- terms are combinations of variables, function symbols, brackets, commas, and whitespace as usual, e.g. `join(X, Y)`
- negation ($\neg$), disjunction ($\vee$), and conjuction ($\wedge$) are `~`, `|`, and `&` respectively, e.g. `(~man(X) | mortal(X)) & man(socrates)`. There are also many other logic connectors in the TPTP that we do not use, e.g. `=>`. Note that the TPTP defines only the syntax, not the semantics, so `~man(X) | mortal(X)` and `man(X) => mortal(X)` are different character strings. Whether they are equivalent in some sense follows from axioms at hand, not the TPTP itself
- quantifiers $\forall$ and $\exists$ are `!` and `?` respectively, e.g. `![X, Y]: join(X, Y) = join(Y, X)` or `![X]: ?[Y]: X != Y`.

A complete FOL formula in TPTP can have the following form (linebreaks are whitespace, one can remove them or change them for tabulations or spaces):

```
cnf(
    label,
    role,
    formula,
    inference(inference_rule, useful_info, inference_parents)
).
```

where `fof` stands for 'first order formula' and

- **label** is an alphanumeric formula label. It can be something meaningful, like `socrates_is_mortal`, or an order number like `10`.
- **role** is a formula role, an alphanumeric like `axiom` or `negated_conjecture`
- **formula** is a FOL formula in a format described above

- **inference_rule** is an inference rule according to which we got the formula, an alphanumeric string like `input` (we read the formula from a file as is) or `resolution` (the prover produced the formula using the resolution rule)
- **useful_info** will be empty for all cases we examine
- **inference_parents** is a list of labels of the formulae used to infer the one at hand, e.g. an empty list (for the `input` rule)

Here is an example of a classical syllogism in TPTP (again, line breaks are whitespace, and the full-stop character serves as a delimiter between formulae):

```
fof(1, axiom, ![X]: (man(X) => mortal(X)), inference(input, , [])).
fof(2, axiom, man(socrates), inference(input, ,[])).
fof(3, lemma, mortal(socrates), inference(resolution, , [1,2])).
```

Another TPTP dialect we will mainly use in this thesis is a CNF one. It is nearly the same, but the formula must be in CNF, and the trailing header is different, e.g.

```
cnf(1, axiom, ~man(X) | mortal(X), inference(input, , [])).
```

Apart from listing formulae, the TPTP file can contain an `include` statement for copying contents of other files (typically, the axiom sets used by different problem files), e.g.

```
include('Axioms/SET001-0.ax').
```

The path in `include` statements is relative to the TPTP root folder (not the prover working directory).

TPTP problem files obey the following naming conventions:

- the filename extension for problems is `.p`, and for axioms — `.ax`
- the filename without the extension has the form `DOMXXXYZZ` where
- `DOM` is a three-letters domain acronym (e.g. `SET` — set theory, `GRP` — group theory)
- `XXX` is a three-digits order number of a problem, starting with `001`
- `Y` is a dialect-defining delimiter, e.g. `+` for FOL and `-` for CNF. One file can not contain formulae written in different dialects
- `ZZ` is a one or two digits problem version number, usually starting from `1`. E.g., there can be two problems with the same theorem statement using two slightly different axiom sets

4.2 Reinforcement learning basics

4.2.1 Reinforcement learning glossary and Markov Decision Processes

Reinforcement learning (RL) is one of the paradigms of machine learning (ML) (see Chapter 20 of [104]). In contrast to supervised learning that we applied in Chapter 3 where an agent (ML model) passively observes example input/output pairs provided by a "teacher", an RL *agent* (also called *actor* sometimes) actively learns from its own experience of interaction with an *environment*. Usually, the agent interacts with the environment in a finite series of *steps* of discrete time. At each step, the agent sends a signal called *action* to the environment and receives back another signal called *reward*, as well as a (partial) snapshot of the environment's current state called *observation*. Mathematically such situations are often modelled ([104], Chapter 17) by a

Definition 4.5 (Markov Decision Process)**.** *Markov Decision Process (MDP)* is a tuple (S, A, P, R) where

- S is a set of *states* of the environment
- A is a set of *actions* (*action space*). For each $s \in S$, we have a subset $A_s \subseteq A$ of actions available to the agent when interacting with the environment in state s. For a *terminal* state s, we have $A_s = \varnothing$
- $P : S \times S \times A \to [0, 1]$ is a *transition model*, such that for each $s, s' \in S, a \in A$ the value of $P(s'|s, a)$ is a probability of transition to state s' from state s as a result of agent's action a. Notice that P is *Markovian*: it depends only on the current state s of the environment, not on the history of all previous states. For a *terminal state* s, for each $s' \in S, a \in A$ and $s \neq s'$ we have $P(s'|s, a) = 0$.
- $R : S \times A \times S \to \mathbb{R}$ is a *reward* function, such that for each $s, s' \in S, a \in A$ the value of $R(s, a, s')$ is a reward received by an agent after a transition from state s to state s' as a result of agent's action a

Agent behaviour is modelled by a *policy* π, a stochastic process mapping a pair (s, t) of a state $s \in S$ and a step number t to a probability distribution over the A_s. Sometimes a policy does not depend on t, then we speak about *static* policies. In our work, all of them will be *dynamic* (depending on t) by default.

The objective of an agent trying to solve an MDP is to find a policy which maximizes its *gain*:

$$\mathbb{E}_{s_{t+1} \sim P(.|s_t, \pi(s_t, t))} \left[\sum_{t=0}^{H} \gamma^t R\left(s_t, \pi\left(s_t, t\right), s_{t+1}\right) \right] \tag{4.1}$$

where $\gamma \in (0, 1]$ is the *discount factor* and H is an integer number (or infinity) called a *horizon*. In the case of an infinite horizon, $\gamma < 1$, but in the case of a finite one (or a transition model having terminal states), we can set $\gamma = 1$.

In RL practice, one groups steps into *episodes*, each beginning in a possible starting state of the environment and ending either in a terminal state of the environment (so-called *termination* of the episode) or because of the exhaustion of the agent resources (e.g. a maximal possible number of steps in a row) which situation is called a *truncation* of an episode. At the end of each episode (or at each step), an agent can adjust its policy of selecting the actions. A sequence of *transitions* (s_t, a_t, r_t, s_{t+1}) (where $r_t = R(s_t, a_t, s_{t+1})$ and $a_t = \pi(s_t, t)$) happened during a particular episode is called a *trajectory*.

4.2.2 Observation as state representation

In real RL applications, an agent rarely has access to the complete environment state or processes it in its original format. In the case of deep reinforcement learning, we model a policy by a deep neural network, so we expect the observation to be a tensor. For example, in simple video games [77], an input can be a tensor of screen pixel colour channels. But then, while training a deep learning model for the policy, we perform two tasks simultaneously:

- extracting valuable information from raw visual input (*representation learning*)
- learning how to win a game based on the information extracted (reinforcement learning per se)

The network architecture in [77] even mirrored these two stages: two convolutional layers first and then two fully-connected ones. Current advances in RL rely on decoupling representation learning from reinforcement learning [127] or the so-called *pre-training and fine-tuning* paradigm. In such an approach, instead of building one large model for a particular task, we:

- start by building an unsupervised or self-supervised generic (task-independent) model learning representations of data points from a huge dataset
- fine-tune the model built at the previous step (i.e. pre-trained) as a part of a supervised or RL model solving a particular task

For example, this paradigm became a cornerstone of the InstructGPT [84] (a "sibling model" of the world-famous ChatGPT [82]) produced by fine-tuning the GPT-3 [19], an enormous (175 billion parameters) language model.

4.2.3 Sparse rewards and parametric actions

Apart from the partially observed state discussed in the previous subsection, real-world RL problems differ from an MDP model because the reward can be *sparse*, i.e. there is no reward (or it equals zero) for nearly all the steps of an episode (often except the last one). For example, let us set a reward to be 1 if a chess player wins, -1 if they lose,

and 0 in case of a draw. Then, during a game of, e.g. 40 moves after each of them but the last one, we can not say who will win with certainty, and thus there is no reward. Setting it to 0 for each non-final move does not change much: we still do not get any information on how the current game move differs from the previous one (it means we can not learn at each step). Contemporary RL algorithms usually work with such situations out of the box. However, sometimes the reward is so sparse that an agent can go through episode after episode without seeing anything but zero. For example, imagine a robotic arm supposed to put a box at a particular spot on the table with a binary reward which equals 1 only when the task is accomplished and is 0 otherwise, no matter how close the box is to the target circle. Standard approaches can not even initialise learning in such cases, and no learning means no change in the policy, which forms a vicious circle without rewards. Nevertheless, the research community recently developed practical RL approaches even for such cases [4].

When the number of actions becomes too large (e.g. around 80000 for Dota 2 [13]), their space being discrete stops making much sense in practice. Nonetheless, moving to full-blown continuous action RL can still be an unnecessary complication. In such situations, practitioners represent observations and actions by embeddings (real-valued vectors) stored in an embedding dictionary (thus keeping the action space finite). Of course, if a policy returns an embedding of non-existent action, an agent chooses one with the nearest embedding instead. One can either use pre-trained static embeddings representing action space topology or train action embeddings from scratch simultaneously with the RL agent policy (or even pre-train and then fine-tune as with observations). We will call such action representations *parametric actions*, following [38] (where authors report "more than millions of possible actions" in RL applications for recommender systems).

4.2.4 Multi-armed bandits

Definition 4.6 (Contextual bandit). [2] A *contextual bandit* is an MDP (see Definition 4.5) for which

- the state space $S \subseteq \left(\mathbb{R}^d\right)^N$ consists of N-tuples of d-dimensional real vectors $b_i, 1 \leq i \leq N$. Each b_i is called a *context*
- the action space $A = \{1, 2, \ldots, N\}$ is a finite set. The set of available actions for every state s is the same $A_s = A$. Each action $a \in A$ is called an *arm*. Thus we talk about an N-armed contextual bandit

Remark 4.7. The simplest case of a contextual bandit is when all the contexts are constant. One can assume $d = N$ and S having only one item, namely $(b_1, b_2, \ldots, b_N)$ where $\{b_i\}_{i=1}^N$ form a basis in $\mathbb{R}^N$. Then we talk about a *multi-armed bandit (MAB)* (thus not a contextual one). The name comes from a hypothetical situation of an agent playing simultaneously several one-armed bandits (slot machines standing in a casino)

and striving to learn which ones have better potential pay-offs. Notice that we also can talk about parametric actions in this case (see Subsection 4.2.3) since our action space is discrete and finite, but every action has a constant vector embedding.

4.3 Machine learning guided automated reasoning

4.3.1 Related work and software architecture choices

ML is applied widely in the automated reasoning domain. There are several projects using reinforcement or supervised learning to guide ITPs: HOList [10] for HOL Light, ASTactic [148] for Coq, TacticZero [146] for HOL4, and another one for Lean [92] to name a few. In these projects, a task for a learning agent is to make the next step in the proof. To achieve that, one gathers a database of existing human-written proof steps, which agents then try to memorise and mimic. As a result, each of these projects comes with its distinctive benchmark, which renders agents guiding different ITPs incomparable. We want to avoid such uniqueness, and keep our research results easier to scrutinise by other research groups. In addition, a benchmarking dataset built from human-written proof formalisations is prohibitively expensive to scale (one needs a graduate student level worker who completed an additional training in the ITP at hand to perform a formalisation, and even then, it is never fast) and biased (depending on the style of thinking of a person who did the formalisation). We want to get tons of cheap data for our ML models, and we also believe a computer can discover proofs inherently different from known ones. After all, we do not want to simulate a working mathematician for the sake of doing it, but we want more theorems proved. So, in our work, we decided not to guide an ITP.

Among ATPs, there are also many projects applying supervised and reinforcement learning techniques. They exist for both saturation (e.g. Deepire [128] for Vampire, ENIGMA [53] for E, and TRAIL [1]) and connection provers (e.g. rlCoP [57] for mlCoP [58], FLoP (Finding Longer Proofs) [152] for fCoP [59], and another one for lazyCoP [98]). Unfortunately, even these projects often are evaluated using different benchmarks, and hardly ever an ML-guided prover enters the CASC competition. Nevertheless, they all can work with theorems from the TPTP library, so, at least in principle, one can objectively compare existing ML-guided ATPs with newly created ones. In addition, since the TPTP does not contain proofs, the agents learn from their proof attempts rather than human-style proofs, which can help us to generate much more training data than we can extract from existing libraries of formalised mathematics.

Among ATPs, one can consider saturation provers less suitable for the RL (e.g., see design considerations from [96]), but the projects we mentioned (ENIGMA, Deepire, and TRAIL) show encouraging results. For example, ENIGMA beat all participants except Vampire in the FOL division of the CASC-J10 [132]. So, keeping possible risks in mind, we decided to concentrate on guiding clause selection in the given clause

algorithm by RL.

Despite the community-accepted standard for implementing RL environments for reproducible research (OpenAI Gym [18]), only the FLoP system followed it from all the projects we mention here. Nevertheless, the FLoP guides a closed-source prover called fCoP, an OCaml reimplementation of leanCoP [83] (GPL licensed prover in Prolog). In our opinion, relying on closed-source software reduces freedom of experimentation and research reproducibility. RL source code for guiding lazyCoP (itself open-sourced) and the source of the rlCoP were never released (to our best knowledge), and the TRAIL is completely proprietary software developed by IBM. Deepire and ENIGMA exist as patched versions of Vampire and E which means that separating the environment from the agent code demands proficiency in the programming languages used to code the provers (C++ and C, respectively) and also a solid understanding of the respective project codebase. One might suppose that such architectural design choices might be among the reasons why ideas contributed by Deepire were never implemented (to our best knowledge) in ENIGMA despite the published evidence of a potential performance boost. Also, in the status quo, one can not judge with certainty whether a particular ML algorithm perks or the underlying ATP properties were the main reason for an ML-guided ATP performance improvement. We want more free idea flow between different research groups (not limited to automated deduction community) for faster scientific progress. Thus, we decided to, first of all, create an RL environment for saturation provers.

4.3.2 A saturation prover as an RL task

For practical implementations of RL, we use Gymnasium[2], a maintained fork of OpenAI Gym [18].

```
import gymnasium as gym

env = gym.make("Environment-v1")
observation, info = env.reset()
terminated, truncated = False, False
while not (terminated or truncated):
    action = ...  # agent's policy call
    observation, reward, terminated, truncated, info = env.step(action)
env.close()
```

Listing 2: A typical use-case for Gymnasium environment

If we look at the Listing 2 and Algorithm 1 on page 36, we will see lots of similarities, ending up in the following translation from ATP to RL parlance (we ignore satisfiable sets of clauses for the rest of the chapter):

[2] https://github.com/Farama-Foundation/Gymnasium

RL term	saturation ATP term
available actions	set of unprocessed clauses
action	given clause
policy	heuristics for given clause selection
observation	sets of processed and unprocessed clauses
episode	proof attempt
termination	refutation found
truncation	timeout or out-of-memory
environment	deduction system (inference rules)
reward	1 if refutation found, 0 otherwise

Things to notice and challenges are:

1. action space and observation space are countably infinite. Thus, one will have to rely on some representations (embeddings) for both, i.e. we are in the situation of parametric actions. We will discuss clause representations in Chapter 5 in more detail

2. available actions set can grow with each step. Nevertheless, since an ATP usually store clauses in RAM, one can set a maximum possible number of clauses as a 'soft' version of the memory limit

3. One can not repeat actions. After an agent selects a given clause, it moves from the unprocessed set from which all future given clauses will come. A typical solution for action availability is defining an action mask — an array `action_mask` of size N (the maximal number of actions mentioned in the previous point) of zeros and ones where `action_mask[i] == 1.0` if and only if the option with order number i is available. Such representation also helps to deal with the fact that in the beginning, we have a much smaller number of actions than N (all the rest are zeroes in `action_mask`). This property is somewhat peculiar for an RL problem, although not unique and shared with the Travelling Salesman Problem (TSP). An action in the TSP is the city to visit next on the condition that the agent must visit each city exactly once. See [73] for a survey of RL for combinatorial optimisation, including the TSP.

4. reward is binary and exceedingly sparse. It is related to the fact that even sub-human performance in automated deduction still seems out of reach. Most proof attempts finish without proof found, so they give no positive reward for an agent to learn from. As mentioned in Subsection 4.2.3, it is not a completely unknown problem in RL, and we will discuss more possible solutions in Chapter 5.

5. if we fix the starting state of the environment (a theorem to prove), it becomes completely deterministic (i.e. we do not have a real MDP here). Nevertheless, we can randomise the environment reset function by letting it use different theorems from some (potentially infinite) pool. This situation makes ATP similar to

the environment built with Box2D [18] physics simulator with landscapes (maps) changing randomly from episode to episode, and we expect the agent to generalise its experience to different conditions. A distinctive property of ATPs here is that some theorems are inherently more complex to prove than others. Curriculum learning [79] is a well-known approach for training a multi-task agent in case of inhomogeneous tasks.

4.3.3 A saturation prover as a multi-armed bandit

Instead of deciding which clause to choose, one can determine from which priority queue of clauses to draw (see Remark 4.2). Such an approach makes the number of possible actions small and fixed at each episode step, thus reformulating the given clause choice problem into a MAB (see Definition 4.7). In addition, one can attach the clause representation for the clauses coming from different priority queues and regard the problem as a contextual bandit (see Definition 4.6). The main hindrance, however, will be that the bandit algorithms like Thompson sampling [2] expect the pay-off to be tied to a particular arm, even if not received immediately after playing it. In the case of proofs, we do not get the reward for steps but rather for episodes (like in the game of chess). So we do not assume bandit formulation to be suitable for guiding provers, although we highlight its deceiving similarities.

4.4 gym-saturation

4.4.1 General description

`gym-saturation` is a collection of OpenAI Gym [18] environments for RL agents guiding the selection of a given clause in saturation provers. It includes two environments: one for Vampire and the other for iProver. Its main features include:

- `gym-saturation` is a free software. All its code is publicly available, as well as the code of the provers it relies on. A permissive licence (Apache 2.0 [3]) lets anyone modify the code for their experiments

- `gym-saturation` is maintained. It is not a public archive of an accompanying code for a paper, but a full-blown Python package, distributed through both PyPI [4] and Conda [5]. It has been receiving updates nearly monthly since July 2021. The maintainer is committed to going on with the compatibility releases and bug fixes after the thesis defence

[3] `https://www.apache.org/licenses/LICENSE-2.0.html`
[4] `https://pypi.org/project/gym-saturation/`
[5] `https://anaconda.org/conda-forge/gym-saturation`

- `gym-saturation` adopts the best practices of software development such as continuous integration [6], automatic generation of API documentation [7], near 100% code coverage [8] by unit-tests and providing a complete environment (a Docker container) to run the code on any machine
- an end-user does not need to be a Vampire or iProver developer, nor they have to have any knowledge of C++ or OCaml. Only Python programming skills and knowledge of OpenAI Gym standard are pre-requisites
- one can use an agent guiding Vampire prover through `gym-saturation` for guiding iProver (and vice versa) with minimal code edits (see Listing 3 for details)
- `gym-saturation` treats an underlying prover as a black box, so it is independent on a particular inference rules system

All these features make `gym-saturation` a unique piece of software, and we hope that it can bring much additional value when applied by researchers from different research groups and diverse backgrounds.

4.4.2 Usage examples

```python
import gym_saturation
import gymnasium

env = gymnasium.make("Vampire-v0")  # or "iProver-v0"
# skip this line to use the default problem
env.set_task("a-TPTP-problem-filename")
observation, info = env.reset()
terminated, truncated = False, False
while not (terminated or truncated):
    # apply policy (a valid random action here)
    action = env.action_space.sample(mask=observation["action_mask"])
    observation, reward, terminated, truncated, info = env.step(action)
env.close()
```

Listing 3: How to use `gym-saturation`

When combined with an agent, `gym-saturation` can work as an ATP. See Listing 3 for an example of a random prover. Notice that to guide iProver instead of Vampire, we have to change only the name of a prover.

For an example of age-weight agent implementation, please look at `agent_testing` module in the `gym-saturation` package. Notice that the agent remains external towards the prover and thus independent from it.

[6] https://app.circleci.com/pipelines/github/inpefess/gym-saturation
[7] https://gym-saturation.readthedocs.io/en/latest/api/utils.html
[8] https://app.codecov.io/gh/inpefess/gym-saturation

Of course, guiding a prover with an external wrapper introduces significant overhead, so for better efficiency, after we get the best possible ML model, we should extract or reimplement machine learning guidance in the prover, avoiding slower languages like Python. In the case of C++ and Torch, one can transform a Python object of a trained model into C++-compatible TorchScript [9] artefact as in [71]. Nevertheless, we argue that ML model development progresses faster in a dedicated environment separated from the prover's implementation technical details.

4.4.3 Architecture

Although the `gym-saturation` user communicates with both iProver and Vampire in the same manner, under the hood, they use different protocols. For Vampire, we relied on the so-called manual (interactive) clause selection mode implemented several years ago for an unrelated task [39]. In this mode, Vampire interrupts the saturation loop and listens to standard input for a number of a given clause instead of applying heuristics. Independent of this mode, Vampire writes (or not, depending on the option `show_all`) newly inferred clauses to its standard output. Using Python package `pexpect`, we attach to Vampire's standard input and output, pass the action chosen by the agent to the former and read observations from the latter. In manual clause selection mode, Vampire works like a server awaiting a request with an action to which it replies with observation (exactly what an environment typically does).

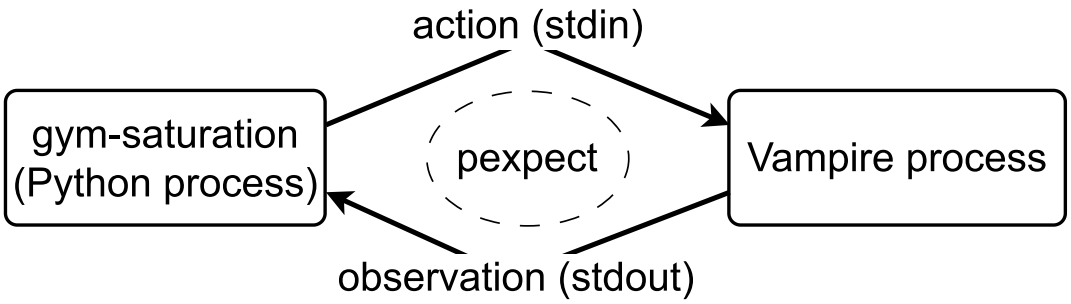

Figure 13: `gym-saturation` wrapping Vampire

In the case of iProver, there existed a way for it to communicate with an external TCP server providing it with guidance. See, for example, the experimental release [10] (iProver did not ship official binary releases at the time of writing). So, iProver behaves as a client which sends a request with observations to some server and awaits a reply containing an action. To make it work with `gym-saturation`, we implemented a tiny *relay server*. It accepts a long-running TCP connection from a running iProver thread, stores its requests to a thread-safe queue [11], and sends responses to it from another such queue filled by `gym-saturation` thread.

[9] `https://pytorch.org/docs/stable/jit.html`
[10] `https://gitlab.com/inpefess/iprover/-/releases/2022.11.03`
[11] `https://docs.python.org/3/library/queue.html#queue.Queue`

See Figure 13 and Figure 14 for a comparison of different communication schemes under the hood of `gym-saturation`.

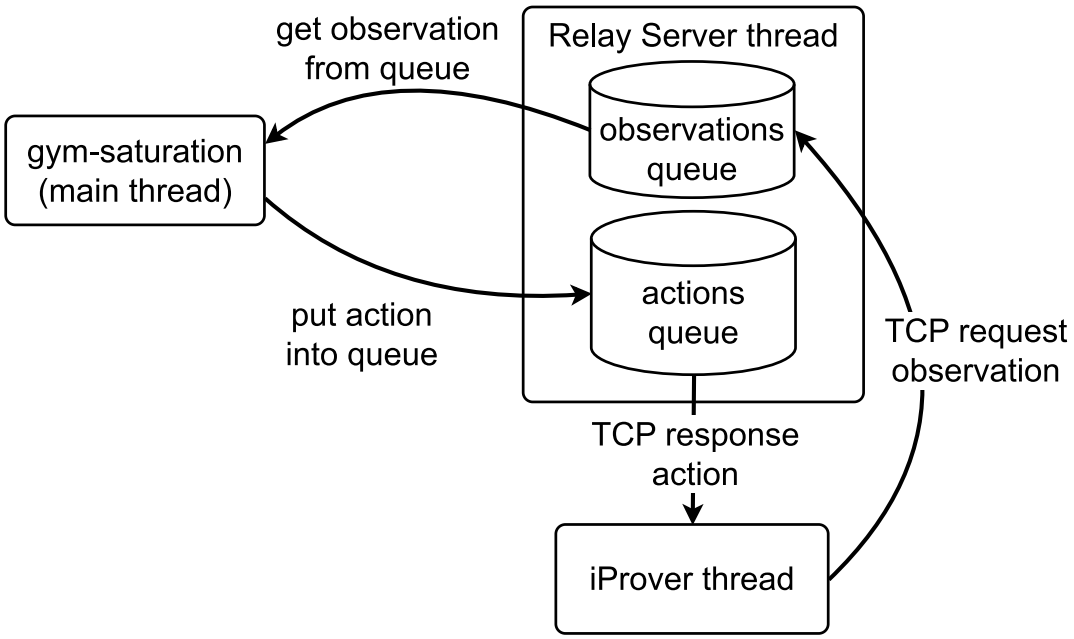

Figure 14: `gym-saturation` interacting with iProver

4.4.4 Implementation details

Clause is a Python dictionary having the following keys and respective values:

- `literals` — a string of clause literals in the TPTP format, e.g. `'member(X0,bb) | member(X0,b)'`
- `label` — a string label of a clause, e.g. '21'. Some provers (e.g. Vampire) use integer numbers for labelling clauses, but others (e.g. iProver) use an alphanumeric mixture (e.g. 'c_54')
- `role` — a string description of a clause role in a proof (hypothesis, negated conjecture, axiom, et cetera)
- `inference_rule` — a string name of an inference rule used to produce the clause. It includes not only resolution and superposition but also values like 'axiom' and 'input' (for theorem assumptions)
- `inference_parents` — a tuple of clause labels if needed by the inference rule ('axiom' does not need any, 'factoring' expects only one, 'resolution' — two, et cetera)
- `birth_step` — an integer step number when the clause appeared in the proof state. Axioms, assumptions, and the negated conjecture have birth step zero.

All the fields except the `birth_step` exist in the TPTP format.

Observation is a Python dictionary with several keys:

- `real_obs` is a tuple of clauses. It can be transformed to tensor representation by so-called observation wrappers
- `action_mask` is a numpy [46] array of the size `max_clauses` (a parameter which one can set during the environment object instantiation) having a value 1.0 at index i if and only if a clause with a zero-based order number i currently exists and is not yet processed. All other values of `action_mask` are zeros. This array simplifies tensor operations on observation representations.

Action is a zero-based order number of a clause from `real_obs`. If a respective `action_mask` is zero, a prover throws an exception during execution of the `step` method.

Reward is 1.0 after a step if we found the refutation at this step and 0.0 otherwise. We discuss different options for post-processing rewards of completed episodes in Chapter 5

Render modes of the environment include a human one which is the same as ANSI one and is the TPTP formatted string. For example, a clause dictionary

```
{
    "literals": "subset(X0,X0)",
    "label": "11",
    "role": "lemma",
    "inference_parents": ("10",),
    "inference_rule": "duplicate_literal_removal",
    "birth_step": 3
}
```

becomes

```
'cnf(11, lemma, subset(X0,X0), inference(duplicate_literal_removal, [], [10])).'
```

4.4.5 Release history and lessons learned

According to the `git` log [12], we have been developing `gym-saturation` since 2021-07-26, making over 500 commits at more than 130 days of activity. It accumulates to about six working months over two years, making it the most labour-intensive project of this thesis. Here we present a list of development milestones with notes and remarks highlighting our findings relevant to the automated provers research.

[12] `https://github.com/inpefess/gym-saturation/commits/master`

- 2021-07-26. The first public version of `gym-saturation`. It did not rely on existing provers like Vampire or iProver but implemented a resolution-based calculus in Python. The inspiration for this work came from PyRes [109], a simplistic ATP in Python.
- 2021-10-01. The first version was submitted for external review (later published as [111]). It added paramodulation to our pure Python prover and its evaluation on the Jean Zay super-computer [13].
- 2022-04-07. The first version with two different provers using the same API. We added a Vampire wrapper and realised that our paramodulation implementation was tremendously inefficient and that Python, in general, was prohibitively slow for implementing a prover
- 2022-05-07. `gym-saturation` appeared in the curated list of the third-party OpenAI Gym environments [14]
- 2022-05-21. We moved the TPTP parser employed by our pure Python prover to a separate project [115]. To our best knowledge, it was the first Python package for parsing the TPTP language
- 2022-10-16. We replaced our pure Python prover with iProver, keeping two different ATPs in the environment collection.
- 2023-02-25. We moved from the original OpenAI Gym to the updated Gymnasium API because of the discontinuance of the general maintenance of the former.

We learned several lessons during our work on `gym-saturation`:

- it is practically impossible to do equally well in training ML models and building ATPs. We implemented paramodulation calculus, for example, but to no surprise, it worked worse than in Vampire, being developed for nearly three decades by a professional team. This sentiment is not only ours [15], and we suppose that more thoughtful design of ML and ATP systems (and their interactions) can facilitate the collaboration of specialists from two respective domains which we see as a remedy
- the more community verified components you reuse, the better. Even a task of parsing the TPTP language, however trivially sounding in the twenty-first century, can bring numerous singularities to debug and performance tweaks to implement

[13] `http://www.idris.fr/eng/jean-zay/`

[14] `https://gymnasium.farama.org/environments/third_party_environments/#gym-saturation-environments-used-to-prove-theorems`

[15] "If you do a doctorate in ML for ATP, then ATP people are intrigued but suspicious of you (they also expect you to apply ML to their obscure ATP stuff); ML people are pleased but have no idea what this ATP thing is (they also expect you to use their obscure ML techniques). Problem one: you know neither...", Michael Rawson at the 7th Conference on Artificial Intelligence and Theorem Proving on September 6, 2022)

- running a prover at scale (on a Slurm [16] cluster, or StarExec [17]) can be technically complex and demanding skills not necessary to develop it. For this matter, we assume that keeping a research team more diversified (not only researchers, but DevOps engineers) might help
- maintaining a research-support system is costly (dependencies change, and one needs to work to keep the system compatible and runnable) and gratifying, although not necessarily expressing itself in papers published and cited

4.5 Conclusion and future work

We briefly overviewed existing attempts of supervised ML and RL applications to automated deduction and justified our focus on saturation provers. Then we analysed a saturation prover guidance as an RL task and identified its three main components: the environment (deductive system/inference rules/calculus), the state representation (encoding logic clauses to real vectors), and the agent (using RL algorithms instead of heuristics). Finally, we contributed a collection of RL environments in standard format (OpenAI Gym) working with two provers: Vampire, written in C++, and iProver, written in OCaml). We showed that when given a policy (a mapping from the state representation to a proposed given clause), an agent can guide both provers seamlessly. We hope that using such standard environments will help:

- ATP developers — to easier transfer policies (including heuristics) from one prover to another without mastering several programming languages and large code bases at a time
- RL practitioners — to apply their experience to automated theorem proving study without the need to deeply specialise in it before they could even start doing what they do best

If the future, we would be happy to add other popular provers to `gym-saturation`: first of all, E, but also other CASC top competitors like Twee [120], Zipperposition [141], and Leo-III (read more on its given clause algorithm version in Chapter 4 of [123]). We also hope that `gym-saturation` might get contributions from someone except its original author. Or at least it will inspire a more successful project that will supersede it, but the community will adopt the RL best practices nonetheless.

Now, we have an RL environment that works, but can an RL agent learn to prove theorems in it? We investigate it in Chapter 5.

[16] `https://slurm.schedmd.com/overview.html`
[17] `https://starexec.ccs.miami.edu/starexec/public/about.jsp`

CHAPTER 5

Generic Reinforcement Learning Prover

The thesis author conducted the research presented in this chapter alone and did not previously publish it elsewhere.

This thesis author started creating an RL-guided prover at the end of the first year of his studies. After running the first makeshift version, it became evident that it was not a coincidence that many publications on attempts to guide ATPs with ML had many co-authors or were parts of many-years projects. Combining graph neural networks, paramodulation calculus, TPTP parsing, and RL in one code repository was too much. Dependencies quickly started conflicting, and the code became undebuggable. Then the idea of a separate environment appeared (described in Chapter 4). A pure Python saturation prover and TPTP parser saw the light of day only to be abandoned as too slow and inefficient shortly afterwards. Guiding Vampire came to the rescue, but the question of state representation stayed impenetrable. Eventually, the architecture using a representation server based on a pre-trained model gave birth to the proof of concept we present here.

Outline of the chapter:

In Section 5.1, we remind architectures of existing ML-guided projects.

In Section 5.2, we introduce a pre-trained model for Python code snippets' embeddings and our technical improvements to it.

In Section 5.3, we talk about the RL algorithm we used in our experiments and why.

In Section 5.4, we look in more detail at how one evaluates RL algorithms in the ATP context.

In Section 5.5, we detail the experiment setup and report the results.

In Section 5.6, we examine approaches to multi-task RL and more experimental data pertinent to our case.

In Section 5.7, we discuss the possible value and risks of using non-monolith architecture for ATP development.

In Section 5.8, we overview lessons learned, problems encountered, and further research directions worth pursuing.

5.1 RL-guided prover architecture

5.1.1 Short overview of existing solutions

In Chapter 4, we noted that to guide a saturation prover with RL, we need at least three components: the environment (`gym-saturation` described in detail in the same chapter), the observation representation, and the RL algorithm per se. We list several projects using RL to guide ATPs and detail their components: base ATP (corresponding to the environment), clause representation, and the RL algorithm.

FLoP [152]

- complete rewrite of an existing prover
- existing clause representation (but depending on the prover)
- standard training algorithm implementation

lazyCoP-based [98]

- an original prover
- original clause representation
- original training algorithm implementation

TRAIL [1]

- claims to be prover-agnostic (but does not publish any code)
- original clause representation
- original training algorithm implementation

The following projects do not claim using RL, but rather *iterative learning* or supervised learning applied in a loop.

Hindsight experience replay-based [5]

- new prover (resolution-only, which effectively means inability to handle equality)
- original clause representation
- original training algorithm

ENIGMA [53]

- uses a patched version of E prover
- original clause representation
- original training algorithm implementation

Deepire [128]

- uses a patched version of Vampire
- original clause representation
- original training algorithm implementation

5.1.2 Prover-agnosticity

As we can see, in all projects we mentioned, authors either:

- write a prover from scratch
- uses an experimental (often heavily modified) version of an existing prover
- do not publish code

Since we also tried implementing a pure Python prover, we can suggest arguments for writing a new ATP for its ML guidance. Existing ATPs are highly competitive, thus (rightfully) sacrificing the readability and maintainability of the code for computational efficiency. It makes navigating and changing their codebases (even when assisted by their developers) sufficiently demanding. For example, during their work on [69], one of Tensorflow core developers proposed a change-set to E code base, noting that "it does not build" [1]. E prover maintainer tried to merge the experimental branch only to revert it later [2]. We think creating new ATPs is not something the community should abandon, but we also believe that ML-guidance should be prover-agnostic. For example, `gym-saturation` can, in principle, work with any saturation prover, and it works with a stable Vampire version and an experimental version of iProver (but the maintainer merged this experimental branch to the main one since then, and iProver does not publish stable binaries).

5.1.3 On representations

To our best knowledge, there were no attempts to compare clause representations published by different research teams, despite such embeddings being an object of dedicated study [94]. Also, we are unaware of attempts to use generic *abstract syntax tree (AST)* embeddings to guide ATPs. One probable reason for it is that researchers creating embeddings either:

- do not publish embedding models training code rendering their research irreproducible (rare, but unfortunate case of some projects we cite in our work)
- do not publish digital artefacts of pre-trained embedding models they successfully applied for guiding ATPs or other tasks (nearly all of the published ATP papers we cite in this thesis)
- implement embedding model inference procedures in an arguably suboptimal way (e.g. in `code2vec`, one has to write a Java code snippet to a file on a disk before calling the embedding function — an unnecessary operation wasting precious time)

We do not claim such practices to be detrimental, especially given the latency considerations from Subsection 5.2.3. Nevertheless, we argue that making an efficient pre-trained FOL embedding model available for free might help RL and ML practitioners to focus more on applying artificial intelligence techniques for theorem proving and make ideas flow between different research groups more fluidly.

[1] `https://github.com/eprover/eprover/pull/2#issue-205749518`

[2] `https://github.com/eprover/eprover/commit/ce581f869932ac98e3c623b48293a8ab12e88dcb`

5.1.4 Original RL algorithm implementations

RL training algorithms are notorious for the number of details that can differ from one implementation to another [51]. It is a well-known problem in the RL community and relates to the general problem of machine learning research reproducibility. We can not imagine any reasonable explanation for not comparing novel RL algorithms with community-verified implementations of well-known ones in a scholarly work. Nevertheless, we acknowledge existing pressure for swift and breathtaking (but irreproducible) results. For example, an anonymous reviewer [151] criticised the FLoP for not being "methodologically new" because it relied on a standard RL algorithm (PPO).

5.2 Representation subsystem

5.2.1 Existing first-order formulae representations and related projects

As discussed in Subsection 4.2.2, to apply any deep reinforcement learning algorithm, one needs a representation of the environment state in a tensor form first. In the case of ML-empowered ATPs (for each project mentioned in this thesis), the authors proposed feature engineering procedures. It can be as simple as clause age and weight (see Remark 4.2), or information extracted from a clause syntax tree [81] or an inference lineage of a clause (Deepire). Representing logic formulae as such is an active research domain: for example, in [95], the authors proposed more than a dozen different embedding techniques based on formulae syntax. In communities other than automated deduction, researchers also study first-order formulae representation: for example, in [7], the authors use semantics representation rather than syntax. One can also notice that first-order logic is nothing more than a formal language, so abstract syntax trees of FOL are not, in principle, that different from those of programming language statements. And of course, encoding models for programming languages (like `code2vec` [3] for Java) exist, and solutions as GPT-3 [19] code embeddings are even commercially available on cloud platforms [3].

To make the first step in this direction, we took advantage of existing pre-trained embedding models for programming languages and tried to apply them to a seemingly disconnected domain of ATPs.

5.2.2 `ast2vec` and our contributions to it

In [85], the authors proposed a particular neural network architecture they called *Recursive Tree Grammar Autoencoders (RTG-AE)*, which encodes ASTs produced by a programming language parser into real vectors. Being interested in education applications, they also published the pre-trained model for Python [86].

To make use of it for our purpose, we furnished several technical improvements to their code:

- a TorchServe [4] handler for HTTP POST requests for embeddings
- request caching with the Memcached server [5]

[3] for example, OpenAI code embeddings delivered by Microsoft: `https://learn.microsoft.com/en-us/azure/cognitive-services/openai/concepts/understand-embeddings#embedding-models`

[4] `https://github.com/pytorch/serve`

[5] `https://www.memcached.org/`

- Docker container to start the whole subsystem easily on any operating system

Our code contribution is freely available [6].

To integrate the `ast2vec` server with `gym-saturation` environments, we added several Gymnasium observation wrappers, transforming a clause in the TPTP language to a Python script. See Figure 15 for a communication diagram and Appendix A for more details.

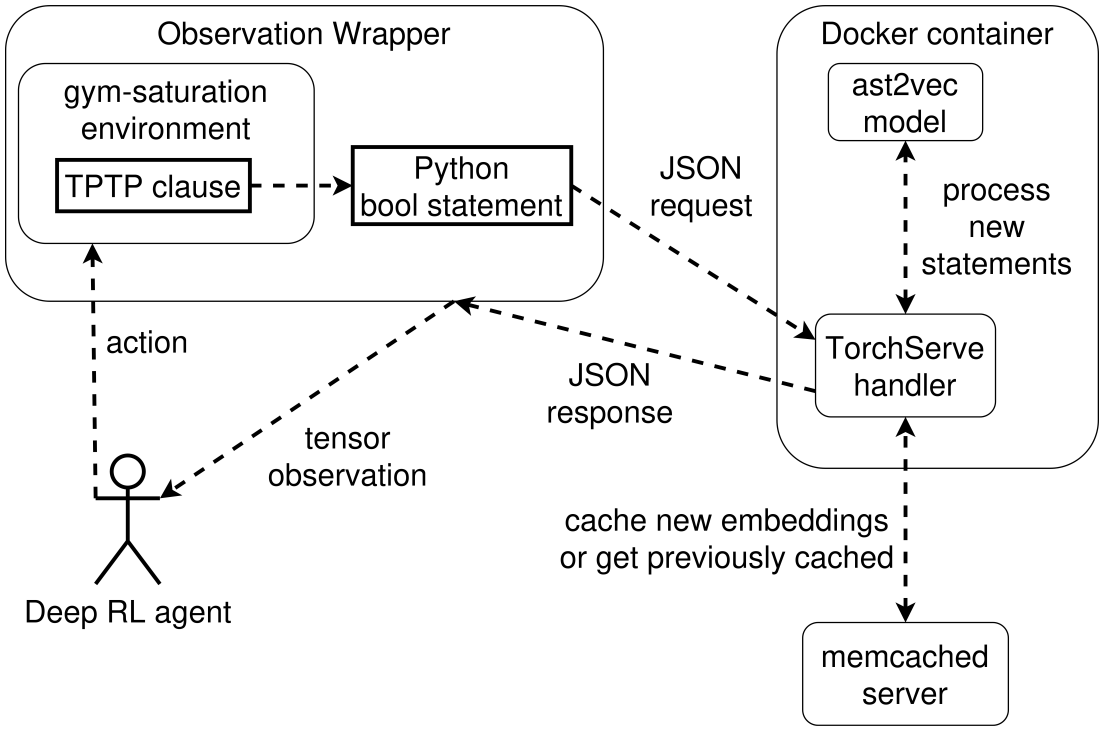

Figure 15: gym-saturation communication with ast2vec

5.2.3 Latency considerations

Looking at Figure 15, one might wonder how efficient is such an architecture. The average response time observed in our experiments was $2ms$ (with a $150ms$ maximum). A typical natural language processing model which embeds whole texts has a latency from $40ms$ to more than $600ms$ [7] (depending on the model complexity and the length of a text to embed) when run on CPU, so there is no reason to believe that `ast2vec` is too slow. When evaluating an ATP, one usually fixes the time limit: for example, $60s$ is the default value for Vampire. Being written in C++ and with a cornucopia of optimisation tweaks, Vampire can generate around a million clauses during this relatively short timeframe. Thus, to be on par with Vampire, a representation service must have latency around $60\mu s$ (orders of magnitude faster than we have). There can be several ways to lower the latency:

- inference in batches (one should train the embedding model to do it; `ast2vec` does not do it out of the box). The improvement may vary

[6] `https://gitlab.com/inpefess/ast2vec`

[7] `https://developer.nvidia.com/blog/optimizing-t5-and-gpt-2-for-real-time-inference-with-tensorrt/`

- use GPU. NVIDIA reports around 20x improvement vs CPU [8]. However, throwing more GPUs will not be as efficient without batch inference from the previous point
- request an embedding for a binary object of an already parsed clause instead of a TPTP string. It means not repeating parsing already done by an ATP, which might lower the latency substantially. To do this, one will have to patch an underlying ATP to return binary objects instead of TPTP strings
- use RPC (remote procedure call) instead of REST protocol. TorchServe relies on REST and parcels in JSON format, and in gRPC [9], they prefer the binary `protobuf` format. One rarely expects sub-millisecond latency from REST, although for RPC, $150\mu s$ is not unusual. This point does not make much sense without the previous one

Since we wanted to build our system without hacking existing ATPs and with an off-the-shelf representation model, we limited the maximal number of clauses for each proof attempt instead. Of course, by trying to solve only problems with 1000 clauses instead of 1000000, we will not create a new ATP that will beat Vampire. But our goal is to understand how RL techniques can help automated theorem proving in principle and provide future researchers with a framework for more fruitfully doing collaborative research. And we hope that the experiments' results presented in the following sections of this chapter show that we achieved this goal.

5.3 RL algorithm

5.3.1 Proximal Policy Optimisation

Definition 5.1 (Q-function)**.** A *quality function* (or *Q-function*, sometimes also called *action-value function*) is the expectation of discounted cumulative rewards conditional on the current state s, the action a chosen in this state, and the policy with parameters θ, according to which the agent selects all the following actions:

$$Q(s,a;\theta) = \mathbb{E}\left[\sum_{j=t}^{H} \gamma^j R(s_j, a_j, s_{j+1}) \,|s_t = s, a_t = a, a_{j>t} = \pi(s_j, j; \theta)\right] \tag{5.1}$$

One can observe that (see Chapter 3 of [133]):

$$Q(s,a;\theta) = \mathbb{E}_{a'=\pi(s',t;\theta),s'}\left[R(s,a,s') + \gamma Q(s',a';\theta)\right] \tag{5.2}$$

Definition 5.2 (Value function)**.** A *value function* [106] $V(s)$ is the expectation of discounted cumulative rewards conditional on the current state s and the policy with parameters θ:

$$V(s;\theta) = \mathbb{E}\left[\sum_{j=t}^{H} \gamma^j R(s_j, a_j, s_{j+1}) \,|s_t = s, a_j = \pi(s_j, j; \theta)\right] \tag{5.3}$$

Remark 5.3. One can notice that Definition 5.2 looks very similar to Definition 5.1 of Q-function. Indeed,

$$V(s;\theta) = \mathbb{E}_{a=\pi(s,t)}\left[Q(s,a;\theta)\right] \tag{5.4}$$

[8] `https://developer.nvidia.com/blog/nlu-with-tensorrt-bert/`
[9] `https://grpc.io/`

Moreover, we can rewrite 5.2 to get:

$$Q(s,a;\theta) = \mathbb{E}_{a'=\pi(s',t;\theta),s'}\left[R(s,a,s')\right] + \gamma\mathbb{E}_{s'}\left[V(s';\theta)\right] \tag{5.5}$$

Definition 5.4 (Advantage function)**.** An *advantage function*

$$A(s,a;\theta) = Q(s,a;\theta) - V(s;\theta) \tag{5.6}$$

shows how much better, on average, is taking action a in the state s than sticking to a chosen policy with parameters θ.

If we use a neural network $\hat{V}(s_t)$ to estimate $V(s_t;\theta)$ and observe a reward r_t at the step t of an episode during the agent training, we can use an estimator of the advantage function based on 5.5 and called a *temporal difference (TD)*:

$$\delta_t = r_t + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t) \tag{5.7}$$

In practice, one uses a truncated *generalised advantage estimation* with a parameter λ:

$$\hat{A}_t = \sum_{i=0}^{T-t-1} (\gamma\lambda)^i \delta_{t+i} \tag{5.8}$$

where T is much less than the episode's length.

Algorithm 2 Proximal policy optimisation

```
for iteration= 1, 2, ... do
  for agent= 1, 2 ..., N do
    Run policy π(., t; θ) in an environment for T times
    Compute advantage estimates Â_1, Â_2, ..., Â_T
  end for
  Optimise L 5.9 wrt θ, with K iterations and minibatch size M ≤ NT
  Update θ with the results of optimisation
end for
```

In *proximal policy optimisation (PPO)* algorithm (see Algorithm 2 and [107] for more details) one optimises the loss function

$$L = \hat{\mathbb{E}}_t\left[-L^{CLIP} + \beta L^{KLPEN} + c_1 L^{VF} - c_2 H_t\right] \tag{5.9}$$

where β, c_1, c_2 are non-negative meta-parameters for balancing different aspects of the loss, $\hat{\mathbb{E}}_t$ is an expectation's estimation (mini-batch average), and the addends are:

1. *clipped surrogate objective* $L^{CLIP} = \min\left(\rho_t\hat{A}_t, \text{clip}(\rho_t, 1-\epsilon, 1+\epsilon)\hat{A}_t\right)$ where
   - $\text{clip}(x,a,b) = \max(a, \min(x,b))$ is a clipping function
   - $\rho_t = \frac{\pi(s_t,t;\theta)}{\pi(s_t,t;\theta_{old})}$ is a probability ratio of two action distributions: $\pi(s_t,t;\theta)$ — the policy which we are training at the moment, and $\pi(s_t,t;\theta_{old})$ — the old policy (used to collect trajectories for the current training phase)
   - $0 < \epsilon < 1$ is a meta-parameter
2. *penalty on KL divergence* $L^{KLPEN} = \text{KL}\left[\pi(s_t,t;\theta_{old}), \pi(s_t,t;\theta)\right]$ where KL is a Kullback-Leibler divergence 3.5 for the old and current policies as discrete probability distributions on possible actions

3. *squared error loss of the value function* $L_{VF} = \left(\hat{V}_t - \hat{G}_t\right)^2$ where $\hat{G}_t = \sum_{i=0}^{T-t-1} \gamma^i r_{t+i}$ is a target value function

4. *entropy bonus* $H_t = -\mathbb{E}\left[\pi\left(s_t, t; \theta\right) \log \pi\left(s_t, t; \theta\right)\right]$ is an entropy of the distribution of the action according to the trained policy at the moment t. Note that the expectation operator is computable, so one does not need an estimation.

In some sense, the "main" part of the loss 5.9 is L^{VF}, and L^{CLIP} and L^{KLPEN} are "only" regularisations. In reality, the clipping and KL penalties constitute the principal PPO contribution making the whole value function optimisation process stable enough for practical applications. Both L^{CLIP} and L^{KLPEN} try to prevent the policy from changing too much based on observed trajectories.

In PPO, the policy network (architecturally incorporated into $\hat{V}\left(s\right)$) returns, not the action probabilities but the normal distribution parameters to sample the actions. Such an approach incorporates the exploration strategy, so one needs to account for it explicitly. The general weight of the exploration in the training process is regulated by the c_2 coefficient (for the policy entropy).

5.3.2 Motivation for choosing PPO

Apart from PPO being a go-to algorithm to try out first new problems among RL practitioners, FLoP [152] successfully used a 'vanilla' PPO for guiding a tableaux prover. An anonymous reviewer asked [151] how techniques applied could help for saturation provers (namely, they cited a project relying on E [69]). We are unaware of papers employing 'vanilla' PPO to saturation provers.

Indeed, TRAIL uses a policy gradient algorithm which belongs to the same family as PPO but is not as sophisticated. For example, its loss includes two terms: the entropy from 5.9 and a much simpler gain estimator. A more significant difference from how PPO works in FLoP is trajectory post-processing. After a successful proof attempt, we know which given clauses became part of refutation proof. So, at step t of a successful proof attempt where the action a_i is selected, the reward r_t is 0 if a_i is not part of the refutation proof; otherwise, r_t is inversely proportional to the time spent proving the conjecture. In FLoP, all the rewards are 0 except for the final step of a successful episode (then it is 1).

ENIGMA does not use a proper RL algorithm but applies Algorithm 3 (see [41] for details).

Algorithm 3 ENIGMA training/evaluation loop

```
fix a set of tasks 𝕋
𝕋₀ ← ∅
for 𝒯 ∈ 𝕋 do
  if non-guided ATP solves 𝒯 then
    add clauses generated while solving 𝒯 to 𝕋₀
  end if
end for
for i = 0, 1, 2, ... do
  train a classifier (clause in proof or not) ℳᵢ on 𝕋ᵢ
  𝕋ᵢ₊₁ ← 𝕋ᵢ
  for 𝒯 ∈ 𝕋 do
    if ATP guided by ℳᵢ solves 𝒯 then
      add clauses generated while solving 𝒯 to 𝕋ᵢ₊₁
    end if
  end for
end for
```

Although one might notice similarities between Algorithm 3 and Algorithm 2, there are enough differences:

- one gathers the initial training data not with a random policy but using a state-of-the-art ATP (with no ML)
- trajectory post-processing (similar to TRAIL: given clauses which ended up as parts of proof get 1 as a reward, and others — 0)
- one discard unsuccessful proof attempts (not the case for the TRAIL)

An analogous loop (called *incremental learning*) appears in another recent work [5] for guiding an unpublished saturation prover. However, there is no pre-generation of training data, and instead of discarding unsuccessful proof attempts, authors produce new tasks from them.

We decided to rely on an existing RL framework containing tested implementations of well-known baselines to eliminate the risk of abandoning an RL algorithm as unsuitable for guiding an ATP only because of flaws in our implementation of it. We have chosen Ray `RLlib` [67] as a library claiming both deep learning framework independence and extendability. Solutions like Tensorflow Agents [45] or Catalyst.RL [65] tend to support only one framework, which we wanted to avoid for greater generality. Since `gym-saturation` implements the standard OpenAI Gym API, it is relatively easy to integrate with libraries like Ray RLlib.

Saying all that, we propose the following research question: can we successfully guide a saturation prover with RL while

- using an out-of-the-box implementation of PPO
- not applying any trajectory post-processing
- not discarding and not transforming unsuccessful proof attempts
- relying on a pre-trained clause representation without fine-tuning it

5.4 RL-guided ATP evaluation

5.4.1 Episode truncation conditions

When we guide the given clause selection in a saturation algorithm over the refutation-complete deductive system and do not limit time and RAM usage, even a random policy will eventually arrive at a proof if only it exists. So the question is not whether an agent can prove a theorem in principle but whether it can succeed within its resource limit. There are several points of view on what constitutes the most valuable resource for an ATP:

- wall-clock time. A standard measure used in CASC competition. It makes perfect sense for "hammers" since they should work fast enough for a human proving a theorem not to get bored
- number of saturation loop iterations (or *activations* since one sometimes calls processed clauses *active* ones, thus choosing a clause as a given one "activates" it) as in [97]. It is a particularly intuitive measure if we try to discern which given clauses are parts of a proof
- number of processor instructions (used in a massive study of random age-weight policy [129]) as a more robust alternative for wall-clock time
- memory (mentioned as a hypothetical bottleneck instead of running time in [130]). Even if not the principal bottleneck for saturation provers, it certainly can be one for some problems. For example, in our experiments with problems from outside the TPTP library (on residuated binars), we saw that Vampire could allocate more than 32GB in less than 15 minutes
- number of clauses (processed and unprocessed combined). We consider this metric a compromise between too imprecise wall-clock time and a much harder-to-evaluate number of processor instructions. We also notice that the number of clauses correlates to RAM allocated, thus having a mixture of different aspects of ATP limitations. In `gym-saturation`, it is a mandatory environment parameter, and we will stick to it in our experiments

We also argue that using time-limit or number of processor instructions metrics entangle two unrelated properties of policies: how effective are the actions they propose in traversing the search space and how computationally efficient they are. Sometimes ML models can produce spectacular results, but their latency or cost of engineering into their target platforms would prevent any practical applications. For example, the Netflix Prize competition [12] harvested dozens of research papers with influential ideas. Nevertheless, the winning model never made it to serve recommendations to the company customers [10]. Nevertheless, if we keep agent's ability to generate proofs separate from how long it takes, we can focus on the study of prospective RL algorithms instead of acing C++ programming. We will need the latter but only for the best candidate model.

In reinforcement learning research, there is an established separation between theoretical benchmarks and practical competitions. For example, the Meta-World [149] benchmark includes object picking and placing task simulations, but there also was a

[10] `https://netflixtechblog.com/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429`

competition [24], in which robots had 20 minutes to pick twelve specified items from shelves in a warehouse. Not every RL algorithm tested with the Meta-World benchmark controls working robots, but this fact can not undermine their research value. To sum up, we suggest evaluating ML guidance of provers per se, not the resulting provers, which often are complex enough even without ML. For that matter, we propose to limit the number of generated clauses in a proof attempt.

5.4.2 What to expect from ML guidance

Running ML models can be slower than computing simple heuristics and not economically viable. Statistics gathered with Deepire [128] show that the less time an ATP spends applying ML, the more problems it solves in a given timeframe. In [5], authors built a system finding proofs 25 times slower than E. Our consideration for representation latency (see Subsection 5.2.3) suggests a similar problem. What do we expect from machine learning models guiding ATPs, then? One can come up with several answers:

- ML guidance is so effective that it gets us across the search space further in one-second step than dozens of super-fast heuristics steps done within the same second
- ML guidance can not outsmart heuristics on shorter time limits, but when provided with more time, an ML-enhanced prover can solve problems which heuristics can not reach even in a whole day of computations

Or we can focus on wallclock-time-independent features of proofs. For example, in Table 5.1, we took all unsatisfiable CNF problems in set theory from TPTP using a particular axiomatisation. We then ran Vampire on each task using a $1:5$ age-to-weight ratio and observed how many steps an attempt took, how many clauses the prover generated, and how many characters there were in these clauses. We then ran Vampire in a manual clause selection mode feeding it only with clauses from proofs found. Not only it reduced the number of steps (more than ten times in some cases), but also the prover generated much fewer clauses in total (more than 40 times in some cases), and the total length of these clauses could be up to 50 times smaller.

Table 5.1: Basic set theory problems from the TPTP. Vampire solves all of them in different number of steps from which only some make part of proof

Problem name	Steps attempted	Steps needed	Clauses generated	Clauses needed	Total chars	Chars needed
SET001-1	9	9	14	13	459	407
SET002-1	35	19	116	30	5613	1216
SET003-1	13	10	22	20	893	850
SET004-1	14	10	23	20	946	850
SET005-1	271	57	2179	202	176314	14755
SET006-1	16	10	28	20	1077	885
SET007-1	1075	78	14721	311	1277271	23898
SET008-1	87	13	345	40	21228	2179
SET009-1	70	23	215	36	11720	1638
SET010-1	1282	90	12675	458	1114679	35827
SET011-1	257	38	2246	116	190038	9030

So, when using ML models (or more powerful heuristics), we essentially fight for successful proof attempts, which contain fewer steps, but also for less numerous and shorter clauses (which translates to both working faster and requiring less memory). RAM is a much more problematic resource to deal with since there is no principal problem in running a prover for a bit longer when we reach the limit, but one can not add more RAM to a server as easily.

We thus propose to look not only (and not mainly) at the number of problems from a particular pool solved but at how frugal the ML model is.

5.5 Experiments

5.5.1 Software and hardware

We use `gym-saturation` as an environment, PPO implementation from Ray RLlib as an agent training algorithm, and the modified version of `ast2vec` model server for clause representation.

We ran all the experiments in this chapter using a workstation with 32GB of RAM and Intel® Core™ i7-10850H CPU @ 2.70GHz.

5.5.2 Data

We have chosen a trivial problem from a fragment of set theory to start [11]:

```
include('Axioms/SET001-0.ax').

cnf(b_equals_bb,hypothesis,
    equal_sets(b,bb) ).

cnf(element_of_b,hypothesis,
    member(element_of_b,b) ).

cnf(prove_element_of_bb,negated_conjecture,
    ~ member(element_of_b,bb) ).
```

which means

$$\frac{A \doteq B, x \in A}{x \in B} \tag{5.10}$$

Where $\doteq$ (`equal_sets` in TPTP) denotes sets equality defined by the following axioms [12]:

```
cnf(membership_in_subsets,axiom,
    ( ~ member(Element,Subset)
    | ~ subset(Subset,Superset)
    | member(Element,Superset) ) ).

cnf(subsets_axiom1,axiom,
    ( subset(Subset,Superset)
    | member(member_of_1_not_of_2(Subset,Superset),Subset) ) ).
```

[11] https://tptp.org/cgi-bin/SeeTPTP?Category=Problems&Domain=SET&File=SET001-1.p

[12] https://www.tptp.org/cgi-bin/SeeTPTP?Category=Axioms&File=SET001-0.ax

```
cnf(subsets_axiom2,axiom,
    ( ~ member(member_of_1_not_of_2(Subset,Superset),Superset)
    | subset(Subset,Superset) ) ).

cnf(set_equal_sets_are_subsets1,axiom,
    ( ~ equal_sets(Subset,Superset)
    | subset(Subset,Superset) ) ).

cnf(set_equal_sets_are_subsets2,axiom,
    ( ~ equal_sets(Superset,Subset)
    | subset(Subset,Superset) ) ).

cnf(subsets_are_set_equal_sets,axiom,
    ( ~ subset(Set1,Set2)
    | ~ subset(Set2,Set1)
    | equal_sets(Set2,Set1) ) ).
```

The first axiom ties $\in$ (`member`) and $\subset$ (`subset`) together (both having their common meaning: set membership and subset relation):

$$\neg (x \in A) \vee \neg (A \subset B) \vee (x \in B) \tag{5.11}$$

Then there are two axioms defining a function $m_0(A, B)$ (`member_of_1_not_of_2`) which returns an element which is in A, but not in B (and is undefined if $A \subset B$):

$$(A \subset B) \vee (m_0(A, B) \in A) \tag{5.12}$$

$$\neg (m_0(A, B) \in B) \vee (A \subset B) \tag{5.13}$$

These two axioms are irrelevant for 5.10, but the prover does not know it in advance.

And finally, `SET001-0.ax` contains three axioms translating to the following statements in common mathematical language:

$$\neg (A \doteq B) \vee (A \subset B) \tag{5.14}$$

$$\neg (A \doteq B) \vee (B \subset A) \tag{5.15}$$

$$\neg (A \subset B) \vee \neg (B \subset A) \vee (A \doteq B) \tag{5.16}$$

where $\neg$ and $\vee$ are logic negation and disjunction respectively. They serve as a definition for set equality relation $\doteq$. Notice that we use a different symbol for this equality since $=$ (`=`) is an equality of terms interpreted by provers as the possibility of substitution of one term for another.

There are only fifteen problems using the axiom set `SET001-0.ax` in the TPTP. We take only the unsatisfiable ones. Vampire can find refutations for all of them using the $1 : 5$ age-weight ratio but generates different numbers of clauses. We sort these theorems by the ascending number of clauses in the final proof state (reflecting how much time Vampire needed to produce the clauses). Some problems use additional axiom sets.

`SET001-1.ax` defines the set union:

```
cnf(member_of_union_is_member_of_one_set,axiom,
    ( ~ union(Set1,Set2,Union)
    | ~ member(Element,Union)
    | member(Element,Set1)
    | member(Element,Set2) ) ).

cnf(member_of_set1_is_member_of_union,axiom,
    ( ~ union(Set1,Set2,Union)
    | ~ member(Element,Set1)
    | member(Element,Union) ) ).

cnf(member_of_set2_is_member_of_union,axiom,
    ( ~ union(Set1,Set2,Union)
    | ~ member(Element,Set2)
    | member(Element,Union) ) ).

cnf(union_axiom1,axiom,
    ( union(Set1,Set2,Union)
    | member(g(Set1,Set2,Union),Set1)
    | member(g(Set1,Set2,Union),Set2)
    | member(g(Set1,Set2,Union),Union) ) ).

cnf(union_axiom2,axiom,
    ( ~ member(g(Set1,Set2,Union),Set1)
    | ~ member(g(Set1,Set2,Union),Union)
    | union(Set1,Set2,Union) ) ).

cnf(union_axiom3,axiom,
    ( ~ member(g(Set1,Set2,Union),Set2)
    | ~ member(g(Set1,Set2,Union),Union)
    | union(Set1,Set2,Union) ) ).
```

The first three axioms introduce a predicate `union(Set1,Set2,Union)` meaning that the set `Union` is a union of two sets `Set1` and `Set2`. These three formulae are CNF of the following statement

$$(A \cup B = C) \Rightarrow (x \in C \Leftrightarrow (x \in A \vee x \in B)) \tag{5.17}$$

The last three axioms define a function $g(A, B, C)$ returning an element from C, but not from A and B, and not defined when $C = A \cup B$. They are CNF of the following statement:

$$C = A \cup B \vee (g(A, B, C) \in C \Leftrightarrow (g(A, B, C) \notin A \wedge g(A, B, C) \notin B))$$

Notice the similarity to 5.12.

Analogously, `SET001-2.ax` introduces intersection, and `SET001-3.ax` — the set difference.

For example, the `SET008-1` is:

```
%----Include the member and subset axioms
```

```
include('Axioms/SET001-0.ax').
%----Include the member and intersection axioms
include('Axioms/SET001-2.ax').
%----Include the member and difference axioms
include('Axioms/SET001-3.ax').

cnf(b_minus_a,hypothesis,
    difference(b,a,bDa) ).

cnf(a_intersection_bDa,negated_conjecture,
    ~ intersection(a,bDa,aI_bDa) ).

cnf(prove_aI_bDa_is_empty,negated_conjecture,
    ~ member(A,aI_bDa) ).
```

It is the CNF of the following statement:

$$\forall A, B, C, D \quad C = A \backslash B \Rightarrow (D = B \cap C \vee \exists x : x \in D) \tag{5.18}$$

where $\backslash$ is a set difference, and $\cap$ is intersection. It means that the intersection of a set difference with its second operand is empty. To compare, the `SET009-1` is:

```
%----Include the member and subset axioms
include('Axioms/SET001-0.ax').
%----Include the member and difference axioms
include('Axioms/SET001-3.ax').

cnf(d_is_a_subset_of_a,hypothesis,
    subset(d,a) ).

cnf(b_minus_a,hypothesis,
    difference(b,a,bDa) ).

cnf(b_minus_d,hypothesis,
    difference(b,d,bDd) ).

cnf(prove_bDa_is_a_subset_of_bDd,negated_conjecture,
    ~ subset(bDa,bDd) ).
```

It is the CNF of the following statement:

$$\forall A, B, C, D, E \quad (A \subset B \wedge D = C \backslash B \wedge E = C \backslash A) \Rightarrow D \subset E \tag{5.19}$$

It is a somewhat verbose form of set difference anti-monotonicity:

$$X \subset Y \Rightarrow (Z \backslash Y) \subset (Z \backslash X) \tag{5.20}$$

5.5.3 Algorithm meta-parameters and random baseline

PPO works with any tensor input that, in our case, are 500×256 since we have at maximum 500 clauses in a proof state, and `ast2vec` returns embeddings of size 256. We

limited the number of clauses to 500 based on Table 5.1 (the maximal number of clauses needed equals 458).

We approximate the value function by a dense neural network with two hidden layers of size 256. Discount factor $\gamma = 0.99$ and the learning rate equals 0.0005. The truncation parameter in 5.8 $\lambda = 1.0$. The amount of transitions collected by two agents in parallel ($N = 2$) before starting the training phase (denoted as NT in Algorithm 2) is 1024, the training batch size (M in Algorithm 2) is 128, and the number of training iterations $K = 8$ (one epoch). Clipping parameter from the loss function 5.9 $\epsilon = 0.3$, the value function target impact $c_1 = 1.0$, the KL penalty weight $\beta = 0.2$, and the entropy part is missing ($c_2 = 0.0$). All these are default parameter values in Ray RLlib, except for N, NT, and K. We reduced the number of parallel agents and the size of the training buffer to fit RAM.

We trained PPO until reaching 0.99 successful episodes at each problem. We also stopped training when having less than 0.01 successful episodes after more than 10000 steps sampled. PPO training started from scratch for each task.

In addition to the PPO, we ran a random agent for 100 episodes at each task. This agent chooses given clauses uniformly at random, so it can perform a bit differently than the first experience collection phase of the PPO, which samples from randomly initialised normal distributions.

5.5.4 Experiment results

We present results in the following tables having similar structures. Table 5.2 shows how many steps were in the best proof found by PPO and random agent, compared to heuristics results from Table 5.1.

Table 5.2: Number of steps attempted to find a proof guiding Vampire by heuristics, best result of a random agent, and best result by PPO

Problem	1:5 Heuristic	Best Random	Best PPO	PPO iterations
SET001-1	9	9	10	1
SET002-1	35	19	36	1
SET003-1	13	9	12	3
SET004-1	14	17	8	8
SET006-1	16	15	14	1
SET008-1	87	23	16	27
SET009-1	70	-	33	14

We can see that sometimes (when the "PPO iterations" column is 1), a problem is so simple that we do not need to train any ML, and a random agent finds proof in fewer steps than a heuristics-based one. That happens because we report only the best result over 100 episodes. We also notice that the random agent can not solve some problems (e.g. `SET009-1`), but the PPO can. Since the PPO starts with random exploration, it can not solve the tasks not present in the table (`SET005-1`, `SET007-1`, `SET010-1`, `SET011-1`), which the random agent can not solve either. Nevertheless, after getting the first negative feedback for the `SET009-1`, the PPO adjusts exploration to find a solution. For the problems where PPO did some training ("PPO iterations" column is more than 1), we observe solutions with fewer unnecessary steps, although not necessarily without them.

We can see similar patterns in Table 5.3 and Table 5.4 for the numbers of clauses and characters generated, respectively.

Table 5.3: Number of clauses Vampire needs to generate to find a proof when guided by heuristics, best result of a random agent, and best result by PPO

Problem	1:5 Heuristics	Best Random	Best PPO	PPO iterations
SET001-1	14	14	15	1
SET002-1	116	32	191	1
SET003-1	22	21	22	3
SET004-1	23	26	21	8
SET006-1	28	30	30	1
SET008-1	345	82	53	27
SET009-1	215	-	55	14

Table 5.4: Vampire characters

Total number of characters in clauses generated by Vampire to find a proof when guided by heuristics, best result of a random agent, and best result by PPO

Problem	1:5 Heuristics	Best Random	Best PPO	PPO iterations
SET001-1	459	419	451	1
SET002-1	5613	1255	12482	1
SET003-1	893	863	893	3
SET004-1	946	1177	863	8
SET006-1	1077	1343	1599	1
SET008-1	21228	5054	3148	27
SET009-1	11720	-	2345	14

5.5.5 Experiment results: answers

The experimental evidence helps us to answer positively to our research question. We can successfully guide a saturation prover with RL while

- using an out-of-the-box implementation of PPO
- not applying any trajectory post-processing
- not discarding and not transforming unsuccessful proof attempts
- relying on a pre-trained clause representation without fine-tuning it

We consider this finding a foundation stone for future ablation studies in research applying RL to ATPs. First, we did not tweak PPO from Ray RLlib in any tangible way. The PPO itself is a well-established algorithm in the RL practice. So, we propose that whoever uses an original RL algorithm should compare it with classical ones and quantify the effect of introducing more complexity.

Second, we did not use any particular neural network architecture for the task. Of course, the dense layers have the annoying property of having too many weights. If we want to scale our networks to millions of clauses, we will stop processing the proof state as a monolithic array of clause embeddings. We can, for example, notice that

we have to separate tasks: fine-tuning clause embeddings and then operating on collections of clauses. If we view these collections as unordered sets, we can use Deep Set architecture [150]. If we think they are sequences, the recurrent neural networks (see Chapter 10 of [42]) are the first option to try. We can treat them as trees or graphs (clauses are nodes, and edges are relations of being an inference parent) and use recurrent or graph neural networks. As always with deep neural networks, options are endless here, so we expect future research to quantify how any architecture complication helps. We also want to highlight that seeking a better neural network architecture is out of the question without a fast enough representation scalable to millions of clauses.

Third, we succeed in not dealing with the reward sparsity. Many saturation ATP guiding projects spread the final reward to all the steps in a finished trajectory. Namely, we can assign a positive reward to the steps where we selected a given clause encountered in a proof, leaving everything else zero. In the PPO, we train using a local buffer containing complete episodes. It shifts the perspective of our agents from proof steps to complete proofs. And as we saw, they can even discover shorter ones because of it. We do not claim such an approach to be better or worse per se, but we suggest future research should quantify the effect of trajectory post-processing.

Finally, we saw that the representation does not matter that much. In many projects cited in this thesis, the authors trained clause embeddings based on previous proof attempts. In our case, we did not learn representation. Moreover, we did not even use a specialised one (pre-trained on first-order logic). `ast2vec` are embeddings of Python scripts. Of course, Python, like nearly any contemporary programming language, have Boolean values and Boolean-valued functions (predicates). So, in a sense, `ast2vec` can encode first-order logic formulae, but it is not specialised in this task. One can say it is not that surprising since contemporary ATPs work well with much simpler representations (only two-dimensional: formula's 'age' and 'weight'). Nevertheless, we expect future research to quantify the effect of introducing and tweaking sophisticated representations.

5.5.6 Experiment results: questions

First of all, the representation latency is crucial, and to play on par with contemporary ATPs, we can not rely on modern code embedding models if we want to work with gigantic proof states (with millions or more clauses). Of course, we can try making representation faster (see Subsection 5.2.3). Another way will be to use a more fine-grained saturation algorithm. E.g., in the TRAIL project [1], they define an action as a pair comprising an inference rule and the given clause. We can make it even more detailed, saying it is a pair of clauses (for the resolution rule). Then we will not generate multiple new clauses at each step which will substantially reduce the number of representation requests. Such an approach can help us tame the representation problem while keeping its latency as it is. Of course, the downside will be the quadratic size of the action space, but ours is already large enough. One can also argue that managing a huge action space of pairs could be easier than an enormous space of a more complex structure. One also should remember that most clauses generated during saturation are useless and not even considered potential candidates for a given clause. So, although the action space will be larger, it will not grow as fast as in a classical given clause algorithm. And finally, moving to non-saturation (e.g. tableaux) provers will spare us of this problem altogether (and undoubtedly bring other headaches instead).

Second, we confirmed that random search does not always work to initialise train-

ing with sparse rewards. There is a plethora of possible solutions to try here. Some of them appeared in previous works (*curriculum learning* in [152] and *hindsight experience replay* in [5]) while others including classical *curiosity-driven exploration* [89] and bleeding-edge *Learning Online with Guidance Offline* [100] are waiting for the time.

Another thing to notice is that we trained the PPO on each problem separately and evaluated them in the same way. It is not typical for ATPs, so we devote the following section to this topic development.

5.6 Multi-task RL

5.6.1 Existing evaluation protocols

In the CASC competition, a prover managing to solve more tasks under fixed constraints wins. To model this situation, we can count the problems solved by a trained RL agent limited, as discussed in the previous subsection, by the maximal clause's number in the proof state. Unfortunately, the same agent can succeed or fail on the same problem (depending on the random seed) since its policy does not often return a single action but rather a probability distribution over them. Of course, one can "fix" such behaviour for evaluation by always choosing the most probable choice instead of sampling them.

We argue that such a CASC-like evaluation outside the competition does not have the same logic. Even if one shows that a system solves more problems from a subset (mentioned in several previously published articles) than a particular ATP, there are still free parameters in such a scheme (fixed in the CASC):

- exact base ATP version. Usually, they become better with time
- time and RAM limits. One can fix them in one research article but not across all of them
- of hardware. Even a random policy can do better on faster CPUs

Another question is whether we should evaluate the agent on the same TPTP problems on which we trained it or on a hold-out set never exposed to the agent during the training time. ENIGMA has used a hold-out set since its appearance [53]. Deepire [128] does not explicitly define any hold-out, but since it trains only on successful proof attempts and counts the problems solved by a trained model in addition to training ones, the absence of an explicit hold-out does not represent a data leak. Indeed, the authors of [5] give a similar justification for not having a hold-out task set.

5.6.2 Multi-task and meta-reinforcement learning

In the RL setting, an agent can interact with MDPs varying from episode to episode (but the MDP stays the same at each step of an episode). In this case, we speak about *multi-task reinforcement learning* [149]. Tasks can be similar (a robotic arm moving a box to the corners or the sides of a table) or strongly inhomogeneous (a robotic arm moving a box around the table or the same arm opening a window). Theorem proving belongs more to the latter case, since

- Some theorems have compact proofs, and others — enormous ones

- Some theorems have concise statements, and others need a whole theory developed and numerous definitions introduced before one even can formulate it (and the complexity of the theorem statement does not necessarily correlate with its proof size)
- theorems belong to various subdomains or use different (even if equivalent) axiom systems
- properties of complex objects can depend on only a subset of defining axioms (e.g. residuated binars from Definition 2.13 are lattices from Definition 2.6, thus some theorems about residuated binars need only lattice axioms)

So, we claim that proving theorems is essentially a multi-task RL problem, and one should evaluate it accordingly. The standard [149] quality metric for a multi-task RL problem is the following.

Definition 5.5 (Mult-task reinforcement learning)**.** A task $\mathcal{T}$ comes from a task distribution $p(\mathcal{T})$, and each task corresponds to a different MDP (see Definition 4.5). We try to build a task-conditioned policy $\pi(s, t, z)$ (where z denotes the task's $\mathcal{T}$ real vector encoding) to maximize

$$\mathbb{E}_{\mathcal{T}\sim p(\mathcal{T})}\left[\mathbb{E}_{\pi}\left[\sum_{t=0}^{H}\gamma^{t}R\left(s_{t},\pi\left(s_{t},t,z\right),s_{t+1}\right)\right]\right] \tag{5.21}$$

This evaluation scheme translates well to the ones discussed in the previous subsection, and we will stick to it.

However, there exists so-called *meta-reinforcement learning* [149]. We take M *meta-training* tasks $\{\mathcal{T}\}_{i=1}^{M}$ from the distribution $p(\mathcal{T})$. We then train a policy $\pi(s, t, z)$ to solve these tasks. We then take another subset of *meta-testing* tasks coming from the same distribution $p(\mathcal{T})$ and continue training the same policy $\pi(s, t, z)$ on them. The meta-learning is successful if policy training achieves better gains on meta-testing tasks in fewer steps than it would without being pre-trained on meta-training ones.

Meta-learning evaluation is somewhat vague, and in [121], the authors highlighted several ambiguities inherent to binary reward environments. For example, the more one runs an agent on a given task $\mathcal{T}$, the better gain one observes on $\mathcal{T}$ (not only because the agent learns, but also because it randomly explores). We saw it in our single-problem experiments (e.g. see Table 5.2). So, when applying meta-RL, we will focus on the number of steps needed to achieve a particular average episode reward level rather than the level itself.

If we plan to use an ML-guided ATP as a "hammer" (part of an ITP), then its application time is often limited by seconds, and continuing training it during this timeframe seems to be an unfit solution. But if we see an ML-guided ATP as a prover we want to use to solve open problems in mathematics, then whatever it does (uses a pre-trained policy or continues training it) is all right if it finds proof eventually. So we argue that the meta-RL approach can be reasonable for ML-guided provers depending on the final goal. For the same reason, we do not evaluate only once (as in Algorithm 3) while in a multi-task RL setting.

5.6.3 Meta-learning in pairs experiment

We repeat the experiment from the previous sections, but instead of training the PPO on a single problem until it reaches a 0.99 success rate, we do the following:

- train the PPO on one problem until it reaches a 0.99 success rate
- continue training the PPO, but using a different problem until it reaches 0.99 success rate or stagnates at less than 0.01 success rate during more than 10000 steps sampled

We do it for all possible pairs of different problems using 7 problems (which the PPO can solve) from the same pool.

5.6.4 Meta-learning experiment results

We present the results in Table 5.5. Sometimes solving another problem before helps to crack the second one faster (each PPO iteration is a sampling of 1024 given clause algorithm steps; it takes around three minutes together with training). In other cases, the effect is the opposite (in RL, they call it *negative transfer*). We have not noticed any distinctive features explaining why sometimes the transfer is positive. It seems to be related to the axiom sets used by the problems but not with 100% certainty.

Table 5.5: The number of the PPO iterations needed to achieve 0.99 success rate for the second problem after solving the first problem.

1st \ 2nd	SET001-1	SET002-1	SET003-1	SET004-1	SET006-1	SET008-1	SET009-1
None	1	1	3	8	1	27	14
SET001-1		4	1	7	1	21	19
SET002-1	1		1	6	1	34	19
SET003-1	1	1		7	1	22	20
SET004-1	1	1	1		1	19	19
SET006-1	1	5	1	6		23	-
SET008-1	1	1	1	5	1		-
SET009-1	1	1	1	1	1	65	

Because of the high frequency of negative transfer cases, our experiments in the multi-task setting (5.21) did not show any encouraging results. That is not to say it can not work in principle, but it means that how successful meta- or multi-task-learning might be, depends heavily on the problems at hand and, probably, on other components. These experiments show that even the simplest multi-layer perceptron model can generalise and transfer what it learned from solving one problem to others. Generating and ordering training data for an RL-based ATP seems to be a rewarding and highly non-trivial task.

5.7 Why have so many moving parts

A typical ATP is a single binary: one can download and run it as is. Having a representation server, multiple agents and environment copies running in parallel and communicating with base prover processes might seem a bit of a mess. There are, of course, many situations when a monolith architecture is preferable. Let us discuss several well-known ones [76]:

- unclear domain. When potential micro-services boundaries are vague, haphazardly drawing them might be the worst solution. It is certainly not the case with

guiding an ATP by ML. Parsing TPTP input is one thing, and making deduction inferences is another. And they both have nothing to do with calculating given clause selection heuristics, let alone the process of learning the best algorithms for computing these heuristics.

- start-ups (since they need to focus on finding the right fit for their products). Every research project is, of course, a start-up. But the mature ATPs like Vampire, E, and especially Prover 9 are already not. And we already know what the ATPs are fit for. They can solve open problems in mathematics, for example. And if mathematicians collaborate with computer scientists, they can afford to run all the micro-services on HPC servers without wasting their time with the tasks they do not excel in

- customer-installed and managed software. Of course, if an ATP is a "hammer" or a mathematician uses it, it is better not to have any moving parts. But for larger-scale (and collaborative) projects, the resulting ML-guided ATP is never (to our best knowledge) a customer-installed software. In our experience, such systems are too often hard to install even while working in pair with its creator

- not having a good reason for using micro-services. And we have the reasons we already mentioned. We believe that ATPs are already complex enough: written in specialised languages (e.g. functional ones like Haskell, OCaml, or Scala), optimised with readability sacrificed, bearing a legacy of past design decisions. Mixing ML in will not make them any more manageable. Moreover, a person well-versed in formal logic and software engineering was not the most common hire for an automated deduction research team. Asking such a candidate to be (in addition) an ML or RL professional might become a unicorn hunt.

We believe that ease of experimentation is a prerequisite for fruitful research. And we argue that searching for the best first-order logic representation and the most optimal implementation of logic calculus could and should be done by specialists in different domains. And finally, we advocate for the collaboration of mentioned specialists and hope that the architecture proposed will serve that purpose.

Another peculiar point we noticed is that parsers of the TPTP language are also inseparable parts of many ATPs. There are exceptions, of course, for example, the `scala-tptp-parser` [125] used by the Leo III [124] prover. We think that the low-latency first-order formulae representation project should start with such a detachable parser. And later, we propose there should be "light" versions of contemporary ATPs containing only deductive inference rules and the given clause loop but no heuristics or clause evaluation procedures. One could link them as separate modules to build a "customer-installed" version of an ATP. To reiterate, we do not propose to ship an ATP with ML guidance as a bunch of micro-services but to develop an ATP as a collection of loosely coupled sub-projects, each of which can be a microservice when in active research.

5.8 Conclusion and future work

In this chapter, we presented the results of our experiments demonstrating the viability of a generic RL prover architecture we proposed. We built it from several ideas to each of which we contributed our implementations:

- an RL environment giving access to different saturation provers through the same API (100% compatible with the OpenAI Gym standard) and decoupling what we guide from how: `gym-saturation` of Chapter 4
- external clause representation service decoupling representation learning from reinforcement learning: `ast2vec` with our technical improvements
- an agent built from off-the-shelf components (easy to test and modify): code for this chapter using the Ray RLlib

We argue that such an approach can facilitate collaboration between machine learning and automated deduction research communities and consider the software we contributed a proof of concept.

Unfortunately, the representation service latency seems to be a bottleneck of our architecture. We identify training a faster first-order formulae embedding model as the best next research goal before improving other architectural parts. Learning first-order formula representations will not be interesting only for guiding saturation provers but also tableaux-based, and even outside of the ATP community per se, e.g. for SMT-solvers.

APPENDIX A

Details of ast2vec representation

Let us take one of the set theory axioms defined in the TPTP file `Axioms/SET001-0.ax`:

```
cnf(membership_in_subsets,axiom,
    ( ~ member(Element,Subset)
    | ~ subset(Subset,Superset)
    | member(Element,Superset) ) ).
```

In commonly-used mathematical notation, it means

$$\forall x \forall A \forall B \, (x \in A \wedge A \subset B) \Longrightarrow x \in B \tag{A.1}$$

The formula

```
(~ member(Element,Subset)
| ~ subset(Subset,Superset)
| member(Element,Superset))
```

is itself a syntactically correct expression in Python. So, when passed to the pre-trained `ast2vec` model, it is first parsed to the abstract syntax tree (AST) depicted in Figure 16.

First, one can notice that the tree in Figure 16 is not a graph since one presume the order of nodes, e.g. a `BinOp` node always has three child nodes in exactly the following order: first operand, binary operation name, second operand. It is not a typical way of encoding graphs in graph neural networks (GNNs). But since `ast2vec` relies on recursive neural networks, it does not matter. As a remedy for using GNNs, one can draw additional arrows in Figure 16 representing the argument order. To distinguish between different types of edges, one can assign values to them (a widespread practice in GNN training).

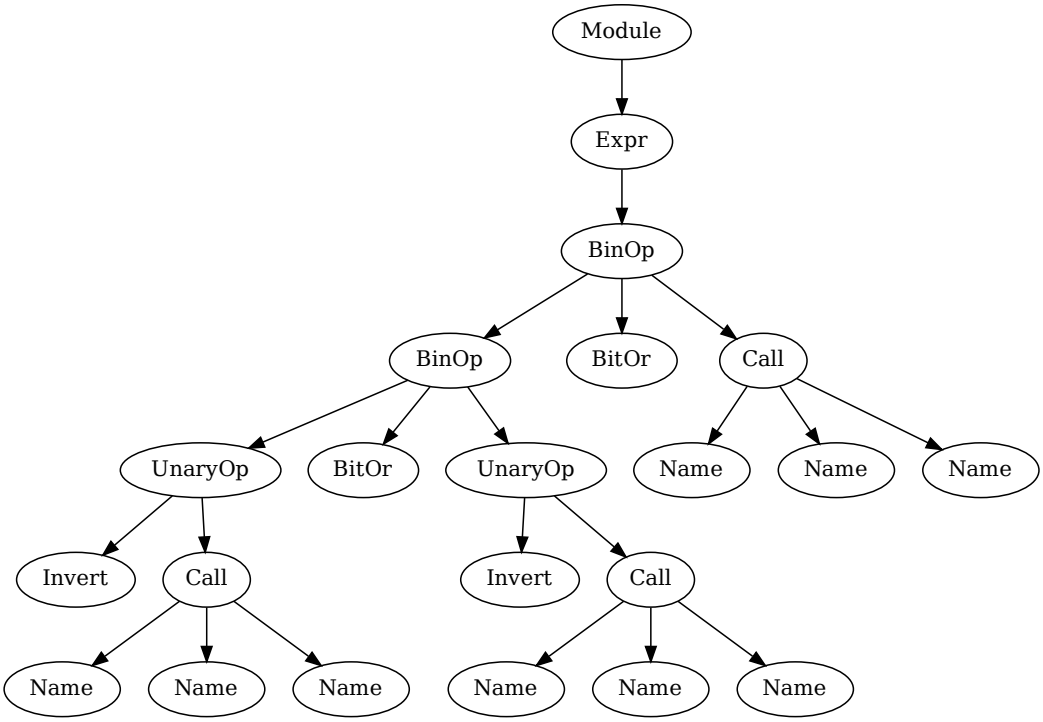

Figure 16: An example of a Python AST

Second, we have lost nearly all the information about symbols: variable names, functions, and predicates are gone. On the one hand, it reflects an inherent renaming invariance of formal languages: we can rename `Element` to `Element_1` or `FirstElement` without changing the semantics and validity of our statement. On the other hand, we are missing that some objects called `Name` in Figure 16 refer to the same variable, while others denote different symbols. It leads to `ast2vec` representation being too 'forgetful', e.g. the following axiom.

```
cnf(subsets_are_set_equal_sets,axiom,
    ( ~ subset(Set1,Set2)
    | ~ subset(Set2,Set1)
    | equal_sets(Set2,Set1) ) ).
```

will have the AST depicted in Figure 16. As a result, A.1 and

$$\forall A \forall B \, (A \subset B \wedge B \subset A) \Longrightarrow A \doteq B$$

will have the same 256-dimensional embedding produced by `ast2vec`. Such a situation might prevent a machine learning model from preferring one of them over the other as a candidate for a given clause and result in failing a proof attempt. One can transform the graph in Figure 16 by adding new nodes and edges representing the relation 'are instances of the same symbol'.

These drawbacks of ASTs are well-known in code analysis research, and enriching them with additional edges is a typical solution (see, for example, [147]).

Bibliography

[1] Ibrahim Abdelaziz, Maxwell Crouse, Bassem Makni, Vernon Austel, Cristina Cornelio, Shajith Ikbal, Pavan Kapanipathi, Ndivhuwo Makondo, Kavitha Srinivas, Michael Witbrock, and Achille Fokoue. Learning to Guide a Saturation-Based Theorem Prover. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(1):738–751, 2023.

[2] Shipra Agrawal and Navin Goyal. Thompson Sampling for Contextual Bandits with Linear Payoffs. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 127–135, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2Vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40:1–40:29, January 2019.

[4] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight Experience Replay. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[5] Eser Aygün, Ankit Anand, Laurent Orseau, Xavier Glorot, Stephen M Mcaleer, Vlad Firoiu, Lei M Zhang, Doina Precup, and Shibl Mourad. Proving Theorems using Incremental Learning and Hindsight Experience Replay. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 1198–1210. PMLR, 17–23 Jul 2022.

[6] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994.

[7] Ali Ballout, Célia da Costa Pereira, and Andrea G. B. Tettamanzi. Learning to Classify Logical Formulas Based on Their Semantic Similarity. In Reyhan Aydoğan, Natalia Criado, Jérôme Lang, Victor Sanchez-Anguix, and Marc Serramia, editors, *PRIMA*

2022: Principles and Practice of Multi-Agent Systems, pages 364–380, Cham, 2023. Springer International Publishing.

[8] Mislav Balunovic, Pavol Bielik, and Martin 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*, volume 31. Curran Associates, Inc., 2018.

[9] Edouard Balzin and Boris Shminke. A neural network for semigroups. *arXiv [cs.LG]*, abs/2103.07388, 2021.

[10] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 454–463. PMLR, 2019.

[11] P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing finite models by reduction to function-free clause logic. *Journal of Applied Logic*, 7(1):58–74, 2009.

[12] James Bennett, Charles Elkan, Bing Liu, Padhraic Smyth, and Domonkos Tikk. KDD Cup and Workshop 2007. *SIGKDD Explor. Newsl.*, 9(2):51–52, dec 2007.

[13] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv [cs.LG]*, abs/1912.06680, 2019.

[14] A. Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In T. Balyo, M. Heule, and M. Järvisalo, editors, *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.

[15] Ekaba Bisong. *Google Colaboratory*, pages 59–64. Apress, Berkeley, CA, 2019.

[16] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010.

[17] D. Brand. Proving Theorems with the Modification Method. *SIAM Journal on Computing*, 4(4):412–430, 1975.

[18] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv [cs.LG]*, abs/1606.01540, 2016.

[19] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models Are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[20] Julie Cailler, Johann Rosain, David Delahaye, Simon Robillard, and Hinde Lilia Bouziane. Goéland: A concurrent tableau-based theorem prover (system description). In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning*, pages 359–368, Cham, 2022. Springer International Publishing.

[21] Karel Chvalovský. On the Independence of Axioms in BL and MTL. *Fuzzy Sets and Systems*, 197(C):123–129, jun 2012.

[22] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation-Principles, Algorithms, Applications*, pages 11–27. Citeseer, 2003.

[23] conda-forge community. The conda-forge Project: Community-based Software Distribution Built on the conda Package Format and Ecosystem. *Zenodo*, https://doi.org/10.5281/zenodo.4774216, July 2015.

[24] Nikolaus Correll, Kostas E. Bekris, Dmitry Berenson, Oliver Brock, Albert Causo, Kris Hauser, Kei Okada, Alberto Rodriguez, Joseph M. Romano, and Peter R. Wurman. Analysis and Observations From the First Amazon Picking Challenge. *IEEE Transactions on Automation Science and Engineering*, 15(1):172–188, 2018.

[25] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[26] A. Distler and J. Mitchell. Smallsemi, a library of small semigroups, Version 0.6.13. https://gap-packages.github.io/smallsemi, Feb 2022. GAP package.

[27] Andreas Distler, Chris Jefferson, Tom Kelsey, and Lars Kotthoff. The Semigroups of Order 10. In Michela Milano, editor, *Principles and Practice of Constraint Programming*, pages 883–899, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[28] Iulia Dragomir, Viorel Preoteasa, and Stavros Tripakis. The Refinement Calculus of Reactive Systems Toolset. *Int. J. Softw. Tools Technol. Transf.*, 22(6):689–708, 2020.

[29] André Duarte and Konstantin Korovin. Implementing Superposition in iProver (System Description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 2020.

[30] Davide Fazio, Antonio Ledda, Francesco Paoli, and Gavin St. John. A substructural Gentzen calculus for orthomodular quantum logic. *The Review of Symbolic Logic*, page 1–22, 2022.

[31] Гладкий А. В. *Введение в современную логику*. МЦНМО, Москва, 2001. `https://www.mccme.ru/free-books/gladkii/glad.pdf`.

[32] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artif. Intell.*, 301:103572, 2021.

[33] Wesley Fussner and Peter Jipsen. Distributive laws in residuated binars. *Algebra Universalis*, 80.54, 2019.

[34] Wesley Fussner and Boris Shminke. Mining counterexamples for wide-signature algebras with an Isabelle server. *arXiv [cs.LO]*, 2021.

[35] N. Galatos and J.G. Raftery. A category equivalence for odd Sugihara monoids and its applications. *J. Pure Appl. Algebra*, 216:2177–2192, 2012.

[36] Nikolaos Galatos, Peter Jipsen, Tomasz Kowalski, and Hiroakira Ono. *Residuated Lattices: An Algebraic Glimpse at Substructural Logics*, volume 151 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2007.

[37] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exp.*, 30(11):1203–1233, 2000.

[38] Jason Gauci, Edoardo Conti, Yitao Liang, Kittipat Virochsiri, Yuchen He, Zachary Kaden, Vivek Narayanan, Xiaohui Ye, Zhengxing Chen, and Scott Fujimoto. Horizon: Facebook's Open Source Applied Reinforcement Learning Platform. *arXiv [cs.LG]*, 2018.

[39] Bernhard Gleiss, Laura Kovács, and Lena Schnedlitz. Interactive visualization of saturation attempts in vampire. In Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa, editors, *Integrated Formal Methods*, pages 504–513, Cham, 2019. Springer International Publishing.

[40] Zarathustra Goertzel, Jan Jakubův, and Josef Urban. ENIGMAWatch: ProofWatch Meets ENIGMA. In Serenella Cerrito and Andrei Popescu, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 374–388, Cham, 2019. Springer International Publishing.

[41] Zarathustra A. Goertzel, Jan Jakubův, Cezary Kaliszyk, Miroslav Olšák, Jelle Piepenbrock, and Josef Urban. The Isabelle ENIGMA. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:21, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[42] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[43] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets.

In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[44] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a Nutshell. *Journal of Formalized Reasoning*, 3(2):153–245, 2010.

[45] Danijar Hafner, James Davidson, and Vincent Vanhoucke. TensorFlow Agents: Efficient Batched Reinforcement Learning in TensorFlow. *arXiv [cs.LG]*, abs/1709.02878, 2017.

[46] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[47] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[48] John Harrison. HOL Light: An Overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 60–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[49] Maximilian PL Haslbeck and Simon Wimmer. Competitive Proving for Fun. *Kalpa Publications in Computing*, 10:9–14, 2019.

[50] Eric C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer, 1993.

[51] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep Reinforcement Learning That Matters. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[52] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed Prioritized Experience Replay. In *International Conference on Learning Representations*, 2018.

[53] Jan Jakubův and Josef Urban. ENIGMA: Efficient Learning-Based Inference Guiding Machine. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *Intelligent Computer Mathematics*, pages 292–302, Cham, 2017. Springer International Publishing.

[54] Di Jin, Zhijing Jin, Zhiting Hu, Olga Vechtomova, and Rada Mihalcea. Deep Learning for Text Style Transfer: A Survey. *Computational Linguistics*, 48(1):155–205, March 2022.

[55] Peter Jipsen and Michael Kinyon. Nonassociative right hoops. *Algebra Universalis*, 80.47, 2019.

[56] Peter Jipsen, Olim Tuyt, and Diego Valota. The structure of finite commutative idempotent involutive residuated lattices. *Algebra universalis*, 82(4):57, Sep 2021.

[57] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement Learning of Theorem Proving. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[58] Cezary Kaliszyk, Josef Urban, and Jirí Vyskocil. Certified Connection Tableaux Proofs for HOL Light and TPTP. In Xavier Leroy and Alwen Tiu, editors, *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 59–66. ACM, 2015.

[59] Cezary Kaliszyk, Josef Urban, and Jiři Vyskočil. Certified Connection Tableaux Proofs for HOL Light and TPTP. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, page 59–66, New York, NY, USA, 2015. Association for Computing Machinery.

[60] Tero Karras, Miika Aittala, Samuli Laine, Erik Härkönen, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Alias-free generative adversarial networks. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 852–863. Curran Associates, Inc., 2021.

[61] Tero Karras, Samuli Laine, and Timo Aila. A Style-Based Generator Architecture for Generative Adversarial Networks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4396–4405, 2019.

[62] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[63] Michael Kinyon. Proof simplification and automated theorem proving. *Philos. Trans. Roy. Soc. A*, 377.20180034, 2019.

[64] Peter Koepke, Anton Lorenzen, and Boris Shminke. CICM'22 System Entries. In Kevin Buzzard and Temur Kutsia, editors, *Intelligent Computer Mathematics*, pages 344–348, Cham, 2022. Springer International Publishing.

[65] Sergey Kolesnikov and Oleksii Hrinchuk. Catalyst.RL: A Distributed Framework for Reproducible RL Research. *arXiv [cs.LG]*, abs/1903.00027, 2019.

[66] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[67] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. Ray RLlib: A Composable and Scalable Reinforcement Learning Library. *arXiv [cs.AI]*, abs/1712.09381, 2017.

[68] Martin Líska, Dávid Lupták, Vít Novotný, Michal Ruzicka, Boris Shminke, Petr Sojka, Michal Stefánik, and Makarius Wenzel. CICM'21 Systems Entries. In Fairouz Kamareddine and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics - 14th International Conference, CICM 2021, Timisoara, Romania, July 26-31, 2021, Proceedings*, volume 12833 of *Lecture Notes in Computer Science*, pages 245–248. Springer, 2021.

[69] Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep Network Guided Proof Search. In Thomas Eiter and David Sands, editors, *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 85–105. EasyChair, 2017.

[70] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The Expressive Power of Neural Networks: A View from the Width. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[71] Suda Martin. LAWA: Learning Age-Weight Alternation for Vampire with LSTM. https://github.com/quickbeam123/lawa, August 2022. GitHub.

[72] Martin Suda. Elements of Reinforcement Learning in Saturation-based Theorem Proving. http://aitp-conference.org/2022/abstract/AITP_2022_paper_11.pdf, 9 2022. 7th Conference on Artificial Intelligence and Theorem Proving.

[73] Nina Mazyavkina, Sergei Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement Learning for Combinatorial Optimization: A Survey. *arXiv [cs.LG]*, abs/2003.03600, 2020.

[74] W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[75] William McCune and Larry Wos. Otter - The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997.

[76] Nabor C. Mendonça, Craig Box, Costin Manolache, and Louis Ryan. The Monolith Strikes Back: Why Istio Migrated From Microservices to a Monolithic Architecture. *IEEE Software*, 38(5):17–22, 2021.

[77] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv [cs.LG]*, abs/1312.5602, 2013.

[78] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

[79] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. Curriculum Learning for Reinforcement Learning Domains: A Framework and Survey. *Journal of Machine Learning Research*, 21(181):1–50, 2020.

[80] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[81] Miroslav Olsák, Cezary Kaliszyk, and Josef Urban. Property Invariant Embedding for Automated Reasoning. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September*

2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020), 2020.

[82] OpenAI. Introducing ChatGPT. https://openai.com/blog/chatgpt, nov 2022.

[83] Jens Otten and Wolfgang Bibel. leanCoP: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36(1):139–161, 2003. First Order Theorem Proving.

[84] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. *arXiv [cs.CL]*, 2022.

[85] Benjamin Paaßen, Irena Koprinska, and Kalina Yacef. Recursive tree grammar autoencoders. *Machine Learning*, Aug 2022.

[86] Benjamin Paassen, Jessica McBroom, Bryn Jeffries, Irena Koprinska, and Kalina Yacef. Mapping Python Programs to Vectors using Recursive Neural Encodings. *Journal of Educational Data Mining*, 13(3):1–35, Oct. 2021.

[87] Kyubyong Park. Can convolutional neural networks crack sudoku puzzles? https://github.com/Kyubyong/sudoku, 2018. GitHub.

[88] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[89] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven Exploration by Self-supervised Prediction. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2778–2787. PMLR, 2017.

[90] J.D. Phillips and D. Stanovský. Automated theorem proving in quasigroup and loop theory. *AI Communications*, 23:267–283, 2010.

[91] Aye Phyu Phyu Aung, Xinrun Wang, Runsheng Yu, Bo An, Senthilnath Jayavelu, and Xiaoli Li. DO-GAN: A Double Oracle Framework for Generative Adversarial Networks. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11265–11274, 2022.

[92] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal Mathematics Statement Curriculum Learning. In *International Conference on Learning Representations*, 2023.

[93] Project Jupyter, Matthias Bussonnier, Jessica Forde, Jeremy Freeman, Brian Granger, Tim Head, Chris Holdgraf, Kyle Kelley, Gladys Nalvarte, Andrew Osheroff, M Pacer, Yuvi Panda, Fernando Perez, Benjamin Ragan Kelley, and Carol Willing. Binder 2.0 - Reproducible, interactive, sharable environments for science at scale. In Fatih Akici, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 17th Python in Science Conference*, pages 113 – 120, 2018.

[94] StanisŁaw PurgaŁ, Julian Parsert, and Cezary Kaliszyk. A study of continuous vector representations for theorem proving. *Journal of Logic and Computation*, 31(8):2057–2083, 02 2021.

[95] StanisŁaw PurgaŁ, Julian Parsert, and Cezary Kaliszyk. A study of continuous vector representations for theorem proving. *Journal of Logic and Computation*, 31(8):2057–2083, 02 2021.

[96] Michael Rawson and Giles Reger. A Neurally-Guided, Parallel Theorem Prover. In Andreas Herzig and Andrei Popescu, editors, *Frontiers of Combining Systems*, pages 40–56, Cham, 2019. Springer International Publishing.

[97] Michael Rawson and Giles Reger. Old or Heavy? Decaying Gracefully with Age/Weight Shapes. In Pascal Fontaine, editor, *Automated Deduction – CADE 27*, pages 462–476, Cham, 2019. Springer International Publishing.

[98] Michael Rawson and Giles Reger. lazyCoP: Lazy Paramodulation Meets Neurally Guided Search. In Anupam Das and Sara Negri, editors, *Automated Reasoning with Analytic Tableaux and Related Methods - 30th International Conference, TABLEAUX 2021, Birmingham, UK, September 6-9, 2021, Proceedings*, volume 12842 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2021.

[99] Emilio Jesus Gallego Arias and Thierry Martinez. PyCoq: Access Coq from Python! https://github.com/ejgallego/pycoq, January 2022. GitHub.

[100] Desik Rengarajan, Gargi Vaidya, Akshay Sarvesh, Dileep M. Kalathil, and Srinivas Shakkottai. Reinforcement Learning with Sparse Rewards using Guidance from Offline Demonstration. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[101] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[102] Constantin Ruhdorfer and Stephan Schulz. Efficient Implementation of Large-Scale Watchlists. In Pascal Fontaine, Konstantin Korovin, Ilias S. Kotsireas, Philipp Rümmer, and Sophie Tourret, editors, *Joint Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning (PAAR) and the 5th Satisfiability Checking and Symbolic Computation Workshop (SC-Square) Workshop, 2020 co-located with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France, June-July, 2020 (Virtual)*, volume 2752 of *CEUR Workshop Proceedings*, pages 120–133. CEUR-WS.org, 2020.

[103] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[104] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[105] Jason Rute, Patrick Massot, Julian Berman, and Frederic Le Roux. Lean client for Python. https://github.com/leanprover-community/lean-client-python, August 2021. GitHub.

[106] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[107] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv [cs.LG]*, abs/1707.06347, 2017.

[108] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, Higher, Stronger: E 2.3. In Pascal Fontaine, editor, *Automated Deduction – CADE 27*, pages 495–507, Cham, 2019. Springer International Publishing.

[109] Stephan Schulz and Adam Pease. Teaching Automated Theorem Proving by Example: PyRes 1.2. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 158–166, Cham, 2020. Springer International Publishing.

[110] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[111] Boris Shminke. gym-saturation: an OpenAI Gym environment for saturation provers. *Journal of Open Source Software*, 7(71):3849, 2022.

[112] Boris Shminke. Project proposal: A modular reinforcement learning based automated theorem prover. *arXiv [cs.AI]*, 2022.

[113] Boris Shminke. Python client for isabelle server. *arXiv [cs.LO]*, abs/2212.11173, 2022.

[114] Boris Shminke. Python client for Isabelle server (0.3.5). *Zenodo*, https://doi.org/10.5281/zenodo.6490275, April 2022.

[115] Boris Shminke. Python TPTP Parser, v.0.0.9. https://doi.org/10.5281/zenodo.7040540, September 2022. Zenodo.

[116] Boris Shminke. Scripts for finding finite models of residuated binars. *Zenodo*, https://doi.org/10.5281/zenodo.7723244, July 2022.

[117] Shminke Boris. Routh's, Menelaus' and Generalized Ceva's Theorems. *Formalized Mathematics*, 20(2):157–159, 2013.

[118] Carlos Simpson. Learning proofs for the classification of nilpotent semigroups. *arXiv [cs.LG]*, abs/2106.03015, 2021.

[119] Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[120] Nicholas Smallbone. Twee: An Equational Theorem Prover. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 602–613, Cham, 2021. Springer International Publishing.

[121] Shagun Sodhani, Amy Zhang, and Joelle Pineau. Multi-Task Reinforcement Learning with Context-based Representations. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 9767–9779. PMLR, 18–24 Jul 2021.

[122] M. Spinks and R. Veroff. Constructive Logic with Strong Negation is a Substructural Logic I. *Studia Logica*, 88:325–348, 2008.

[123] Alexander Steen. *Extensional paramodulation for higher-order logic and its effective implementation Leo-III*. PhD thesis, Free University of Berlin, Dahlem, Germany, 2018.

[124] Alexander Steen. Leo-III 1.7. https://doi.org/10.5281/zenodo.7650205, July 2022. Zenodo.

[125] Alexander Steen. scala-tptp-parser v1.7.0. https://doi.org/10.5281/zenodo.7739821, March 2023. Zenodo.

[126] Alexander Steen and Christoph Benzmüller. The Higher-Order Prover Leo-III. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning*, pages 108–116, Cham, 2018. Springer International Publishing.

[127] Adam Stooke, Kimin Lee, Pieter Abbeel, and Michael Laskin. Decoupling Representation Learning from Reinforcement Learning. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 9870–9879. PMLR, 18–24 Jul 2021.

[128] Martin Suda. Improving ENIGMA-style Clause Selection while Learning From History. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 543–561, Cham, 2021. Springer International Publishing.

[129] Martin Suda. Vampire getting noisy: Will random bits help conquer chaos? (system description). In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning*, pages 659–667, Cham, 2022. Springer International Publishing.

[130] Martin Suda. Vampire getting noisy: Will random bits help conquer chaos? (system description). EasyChair Preprint no. 7719, EasyChair, 2022.

[131] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure - From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[132] Geoff Sutcliffe. The 10th IJCAR automated theorem proving system competition - CASC-J10. *AI Communications*, 34(2):163–177, 2021.

[133] R.S. Sutton and A.G. Barto. *Reinforcement Learning, second edition: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018.

[134] The Coq Development Team. The Coq Proof Assistant. *Zenodo*, https://doi.org/10.5281/zenodo.5846982, January 2022.

[135] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.12.2*, 2022. https://www.gap-system.org.

[136] TIOBE Software BV. The TIOBE Programming Community index. https://www.tiobe.com/tiobe-index/, April 2022.

[137] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *In Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 632–647. Wiley, 2007.

[138] Sam van Gool, Adrien Guatto, George Metcalfe, and Simon Santschi. Time Warps, from Algebra to Algorithms. In Uli Fahrenberg, Mai Gehrke, Luigi Santocanale, and Michael Winter, editors, *Relational and Algebraic Methods in Computer Science*, pages 309–324, Cham, 2021. Springer International Publishing.

[139] Robert Veroff. Using Hints to Increase the Effectiveness of an Automated Reasoning Program: Case Studies. *Journal of Automated Reasoning*, 16(3):223–239, 1996.

[140] Robert Veroff. Solving Open Questions and Other Challenge Problems Using Proof Sketches. *Journal of Automated Reasoning*, 27(2):157–174, 2001.

[141] Petar Vukmirovic, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. Making Higher-Order Superposition Work. *Journal of Automated Reasoning*, 66(4):541–564, 2022.

[142] Uwe Waldmann, Sophie Tourret, Simon Robillard, and Jasmin Blanchette. A Comprehensive Framework for Saturation Theorem Proving. *Journal of Automated Reasoning*, 66(4):499–539, 2022.

[143] Mingzhe Wang and Jia Deng. Learning to Prove Theorems by Learning to Generate Theorems. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[144] Makarius Wenzel. Isabelle/PIDE after 10 years of development. In *13th International Workshop on User Interfaces for Theorem Provers (UITP 2018)*, Federated Logic Conference 2018, 2018.

[145] Makarius Wenzel. The Isabelle System Manual. https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/system.pdf, December 2021.

[146] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. TacticZero: Learning to Prove Theorems from Scratch with Deep Reinforcement Learning. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 9330–9342. Curran Associates, Inc., 2021.

[147] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

[148] Kaiyu Yang and Jia Deng. Learning to Prove Theorems via Interacting with Proof Assistants. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6984–6994. PMLR, 2019.

[149] Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-World: A Benchmark and Evaluation for Multi-Task and Meta Reinforcement Learning. In Leslie Pack Kaelbling, Danica Kragic, and Komei Sugiura, editors, *Proceedings of the Conference on Robot Learning*, volume 100 of *Proceedings of Machine Learning Research*, pages 1094–1100. PMLR, 30 Oct–01 Nov 2020.

[150] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep Sets. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[151] Zsolt Zombori, Adrián Csiszárik, Henryk Michalewski, Cezary Kaliszyk, and Josef Urban. Towards finding longer proofs, 2020. OpenReview.net.

[152] Zsolt Zombori, Adrián Csiszárik, Henryk Michalewski, Cezary Kaliszyk, and Josef Urban. Towards Finding Longer Proofs. In Anupam Das and Sara Negri, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 167–186, Cham, 2021. Springer International Publishing.