Guiding Automated Theorem Proving with Machine Learning

Jelle Piepenbrock

Institute for Computing and Information Sciences

RADBOUD UNIVERSITY PRESS

Radboud Dissertation Series

Guiding Automated Theorem Proving with Machine Learning

Jelle Piepenbrock

Author: Jelle Hermanus Piepenbrock
Title: Guiding Automated Theorem Proving with Machine Learning

Radboud Dissertations Series
ISSN: 2950-2772 (Online); 2950-2780 (Print)

Published by RADBOUD UNIVERSITY PRESS
Postbus 9100, 6500 HA Nijmegen, The Netherlands
www.radbouduniversitypress.nl

Design: Jelle Hermanus Piepenbrock
Cover: Proefschrift AIO | Guntra Laivacuma
Printing: DPN Rikken/Pumbo

ISBN: 9789465150499
DOI: 10.54195/9789465150499
Free download at: https://doi.org/10.54195/9789465150499
© 2025 Jelle Hermanus Piepenbrock

RADBOUD
UNIVERSITY
PRESS

This is an Open Access book published under the terms of Creative Commons Attribution-Noncommercial-NoDerivatives International license (CC BY-NC-ND 4.0). This license allows reusers to copy and distribute the material in any medium or format in unadapted form only, for noncommercial purposes only, and only so long as attribution is given to the creator, see http://creativecommons.org/licenses/by-nc-nd/4.0/.

Guiding Automated Theorem Proving with Machine Learning

Proefschrift ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.M. Sanders,
volgens besluit van het college voor promoties
in het openbaar te verdedigen op

dinsdag 11 maart 2025
om 16.30 precies

door

Jelle Hermanus Piepenbrock

Promotor:

- Prof. dr. T.M. Heskes

Copromotoren:

- Dr. M. Janota (České vysoké učení technické v Praze, Tsjechië)
- Dr. J. Urban (České vysoké učení technické v Praze, Tsjechië)

Manuscriptcommissie:

- Prof. dr. J.H. Geuvers
- Prof. dr. S. Schulz (Duale Hochschule Baden-Württemberg Stuttgart, Duitsland)
- Dr. M. Johansson (Chalmers University, Zweden)

Guiding Automated Theorem Proving with Machine Learning

Dissertation to obtain the degree of doctor
from Radboud University Nijmegen
on the authority of the Rector Magnificus prof. dr. J.M. Sanders,
according to the decision of the Doctorate Board
to be defended in public on

Tuesday, March 11, 2025
at 4.30 pm

by

Jelle Hermanus Piepenbrock

Supervisor:

- Prof. dr. T.M. Heskes

Co-Supervisors:

- Dr. M. Janota (Czech Technical University in Prague, Czech Republic)
- Dr. J. Urban (Czech Technical University in Prague, Czech Republic)

Manuscript Committee:

- Prof. dr. J.H. Geuvers
- Prof. dr. S. Schulz (Baden-Württemberg Cooperative State University, Germany)
- Dr. M. Johansson (Chalmers University, Sweden)

Contents

Chapter 1

Introduction

The formalization and automation of reasoning are long-standing goals in the sciences [18]. The practical implementation of reasoning based on machine-based calculations was not in reach for most of that history and is a relatively recent achievement, with the possibilities perhaps appearing as a vision on the horizon in the late 19th century to people like Babbage and Lovelace, who were working on their analytical engine [3]. However, many people throughout history have seen the advantage of automation, even back in antiquity.

A famous example is the Antikythera mechanism, which is a type of hand-driven analog computer that functions as a model of the solar system [35]. The artifact was built to predict the positions of astronomical objects in the second or first century BC. This type of special-purpose computer is more limited than the general-purpose computing paradigm that emerged in the 20th century after the work of Turing [127] and others. However, it indicates that, for a long time, people have understood that encoding knowledge inside mechanical artifacts and using them to derive the consequences could be a useful activity.

In this thesis, the focus is on improving modern computerized reasoning mechanisms with machine learning. While for many reasoning tasks, the procedures themselves are guaranteed to lead to correct answers, they may not finish in a practical time span, due to the space of possible consequences to reckon with being very large. Therefore, we applied machine learning to speed up the processes and improve the reasoning procedures. In the most basic sense, the goal is to compile statistics on what reasoning steps led to successful proofs, and then refocus the reasoning procedure to make reasoning steps that are similar to these successful choices. The statistics may be compressed (and ideally also

generalized [77]) in the form of a statistical predictor, such as a neural network, in a process known as machine learning. In this parlance, it becomes possible to think of the neural network statistical predictor as controlling and directing the reasoning search procedure, to guide it towards fast and correct conclusions. We will now go into more detail on the logical theories that underlie the modern automated reasoning procedures considered in this thesis. Afterwards, we introduce the machine learning methods used to control and guide these reasoning engines.

1.1 Automated Theorem Proving

In this section, we explain aspects of the field of automated reasoning or automated theorem proving (ATP) in such a way that someone from the machine learning field can, after reading, understand the main mechanisms of the automated reasoning systems used in the later research chapters. This is not meant to be an exhaustive account of ATP, but as a guide to ease readers into the context where machine learning was applied.

1.1.1 Classical Propositional Logic

In order to do reasoning, one must first choose a logic. The most widely used logical system is that of *classical* propositional logic. We specify classical here because there are other propositional logics that are not the same as the theory which is often referred to as *propositional logic*.

Classical propositional logic concerns itself with whether statements or propositions are true or false. These propositions can be combined using logical connectives, such as *and*, *or*, *implies*, and *not*. Consider two propositions a and b. These are propositions that cannot be further decomposed: they are *atomic*. One can make a new proposition that is only true when both a and b are true. This is called an 'and' statement or a *conjunction* and is usually denoted by the symbol $\wedge$:

$$a \wedge b.$$

Similarly, combined statements can be constructed to be true when at least one of their constituents is true ('or', denoted by $\vee$). This is also known as a *disjunction*:

$$a \vee b.$$

One can also discuss the opposite of a proposition by using logical negation ('not', $\neg$). In classical logic, the following holds:

$$a = \neg\neg a,$$

which says that double negation recovers the original proposition, where $=$ is meant as logical equivalence.

In fact, these three connectives, $\wedge, \vee, \neg$ are a complete set of logical operators and we do not need any more of them [97] to represent propositions in classical propositional logic.[1] In logical reasoning, one often speaks of implications, i.e. that the truth of one statement implies the truth of another. This is how reasoning is often represented: a statement can be proven to be true by proving that it follows from the assumptions (or 'axioms') by a chain of implications. If a implies b, it can be written as follows, using the $\implies$ symbol:

$$a \implies b.$$

This new implies symbol can be written out in terms of two of the three connectives we introduced above:

$$(a \implies b) = (\neg a \vee b),$$

which means that for a statement a to imply a statement b, either a is not true, or b is true.

While logical statements are often described in terms of the three operations $\wedge, \vee, \neg$, it is possible to use only one operation: the NAND, or 'not and' operation. This is an operation with the following logical content in terms of the connectives already introduced:

$$\text{NAND}(a, b) = \neg(a \wedge b).$$

This *NAND operator* has a very practical property: it by itself is enough to, by composing it, construct any proposition. As modern computer circuits are built using logical operations, this also means that one can construct any computer circuit implementing a propositional statement with just a single component.[2] A circuit that computes the 'or' function can be constructed from

[1] Strictly, even ($\wedge$ and $\neg$) or ($\vee$ and $\neg$) together are enough, as $\vee$ and $\wedge$ can be expressed in terms of each other with the help of $\neg$, via DeMorgan's laws: $\neg(a \wedge b) = \neg a \vee \neg b$ and $\neg(a \vee b) = \neg a \wedge \neg b$.

[2] This also holds for its counterpart, NOR.

NAND operations: $or(a, b) = \text{NAND}(\text{NAND}(a, a), \text{NAND}(b, b))$. This example was intended to prompt the reader to think about how the 'basic unit' of a logical reasoning system influences how easily or compactly one can write down and reason about certain propositions. While it is possible to construct a modern computer from just NAND gates, it could generate large circuits, as several NANDs have to be composed to implement the other operations such as *or, and, implies, not*, which are closer to the usual way a programmer or a mathematician is trained to think.

Now, we might consider an example of how one would encode the real world into propositional logic statements. Imagine that it is a cold winter day, and you want to decide whether you can go outside without being too cold. You need to wear warm clothes, and (in this example) you need to cover your hands, torso, head, and legs. However, you have both gloves and mittens that you can use to cover your hands, but you cannot wear both at the same time. Logically, you might think of the proposition 'ready to go outside' in the following way:

$$\begin{aligned} &\text{ReadyToGoOutside} = \\ &\text{WoolHat} \wedge \text{WinterSweater} \\ &\wedge(\text{Mittens} \vee \text{Gloves}) \\ &\wedge\{\neg(\text{Mittens} \wedge \text{Gloves})\} \\ &\wedge\neg\text{ShortPants} \end{aligned}$$

where we have used more informative names than a and b that were used in the definitions above, but the way the propositions that make up the larger statement are treated did not change.

1.1.2 Practical Uses of Propositional Logic

While the above example is small and concise enough to figure out a way to configure the components without the help of a computer, the same principles can be scaled up to encode more complicated logical statements. From chess and sudoku puzzles to extremely large industrial designs, the language of propositional logic can be used to formally capture the problems. In our example, we had some intuition about the meaning of the propositions, i.e. the words mittens or gloves, etc. allow the human brain to import some contextual knowledge that makes it possible to intuitively grasp the problem. However, what should be the procedure when one is handed a (possibly extremely) large logical statement and asked the question "is this statement true?" or "is this statement false?". One way is to phrase the problem in terms of *satisfiability*. For a statement to

be satisfiable, there has to be a setting for the propositional variables (such as a, b or Gloves) that make the statement evaluate to *true*. Such a setting is an *assignment*. The relationship between a satisfiable statement and a true statement is that while a satisfiable statement only needs to have one assignment that makes the statement evaluate to *true*, a true statement evaluates to true under all assignments. When there is no assignment under which a statement or problem evaluates to true, it is called *unsatisfiable*.

Because we know each proposition can only either be true or false, we could adopt a strategy of patiently checking each possible setting of the truth values of each atomic proposition. We could for each setting then check whether the entire statement evaluates to true. However, there are exponentially many different truth value settings that have to be checked (2^N, where N is the number of variables), and in the worst case, the very last setting that has to be checked can be the right one. Many researchers during the last century have investigated more practical, faster strategies to check whether a problem is satisfiable: in modern days, this area of research is known as 'SAT solving'.

1.1.3 SAT solvers

A satisfiability solver, or SAT solver in short, aims to exploit structure in the problem to answer the question of whether a given problem is satisfiable or not [79]. The problems are usually standardized in a format known as *conjunctive normal form* (CNF), which can be written as follows:

$$\bigwedge_{i=1}^{N} \left(\bigvee_{j=1}^{K_i} l_{ij} \right),$$

where N is the number of statements connected by $\wedge$ symbols. In the CNF SAT solving context, each atomic proposition is usually called a *variable* (for example, v) and a particular positive or negative (v or $\neg v$) occurrence of the variable is called a literal (here denoted by various different l). The $\bigwedge$ and $\bigvee$ symbols are shorthand for a conjunction or disjunction with potentially many subexpressions. Each of these top-level parts in the $\wedge$ expression is called a *clause*. K_i is the number of literals l, or statements connected by $\vee$ in each particular clause i. Any statement in propositional logic can be rewritten to this form.

SAT solvers can differ in their implementations and strategies to exploit the structure of particular problems. As a guide to intuition, consider again the structure of the winter clothing example in Section 1.1.1. Notice that the outer

part of the problem, which consists of 'and' statements, has the property that if even one of its parts cannot be satisfied, then the entire statement cannot be satisfied. This means that when a problem is written in CNF form, where all the clauses are connected by conjunctions ($\wedge$), one can potentially save a lot of work. If one can find out that one of the clauses can never be satisfied, then the problem cannot be satisfied.

1.1.4 Resolution and the Search for the Empty Clause

The first procedures we want to discuss are (propositional) resolution and the Davis-Putnam procedure [27]. This allows us to introduce terms and concepts that will be valuable later. Let us assume that the following two propositions are true:

$$\begin{array}{r} a \implies b \\ a \end{array}$$

which is the well-known setup known as *modus ponens* that allows us to conclude that b must also be true. Said in natural language: if "a implies b" is true, and "a" is true, then "b" must be true. Consider now the alternative representation of $\implies$ from Section 1.1.1:

$$\begin{array}{r} \neg a \vee b \\ a \end{array}$$

which makes it clear what the underlying structure is: if one has two clauses that contain literals of opposite polarity, the clauses may be contracted together or 'resolved'. This resolution operation can be written in proof style formatting as follows:

$$\frac{\neg a \vee b \qquad a}{b}\,.$$

The comparison with implication makes clear the mechanics of the resolution operation, but it is not limited to such simple cases: as long as there are literals of the same variable with opposite polarity, two clauses can be resolved and a new clause can be derived. A more general specification of the resolution rule is thus:

$$\frac{\neg r \vee \bigvee l_i \qquad r \vee \bigvee l_j}{\bigvee l_i \vee \bigvee l_j}\,,$$

where r is the variable that the clauses are being resolved on, the literals in the first clause that are not $\neg r$ are labelled by i, and the literals in the second

clause that are not r are labelled by j. All the non-r literals are combined into a newly derived clause.

Now we consider how these chains of implications, or more generally, resolutions, can be used to construct proofs. The clause derived by resolution (*resolvent*) can be both longer and shorter than its parent clauses. The resolvent will have fewer literals than some of its parents when at least one of the parents has only one literal. This is important, because, in the end, we may end up with two single-literal clauses (*unit clauses*) of opposite polarity that both have to be true to satisfy the problem. Resolving two such clauses will clearly lead us to a *conflict*:

$$\frac{\neg a \qquad a}{\bot},$$

where we have used $\bot$ to denote the empty clause. This can also be thought of as 'deriving *false*'. For an intuition on this, consider that for an 'or' statement to be true, at least one of the subexpressions has to be true: if there are zero subexpressions, the statement is therefore false.

This situation means that there is a conflict that follows from our current assignment. If we can change the assignment of certain variables, we may backtrack and search for another path through the search space. If we cannot change the assignments of variables (for example because they appear in unit clauses), or we have already tried all the other options, we can conclude that the problem is unsatisfiable. In many provers, one adds the negated version of a conjecture to a set of axioms that are presumed to be non-contradictory: if the problem is found to be unsatisfiable in such a situation, the conclusion is that the conjecture must be true given the axioms.

While this resolution style solving was one of the first ways to automate propositional logic and resolution is also still the lens through which theory about proof complexity is done [51], modern SAT solvers are mainly powered by somewhat different principles. In Section 1.1.8, we will revisit resolution in a more general setting.

1.1.5 DPLL & CDCL: Solving By Decision and Backtracking

The basic conceptual framework within which most modern SAT solvers operate is that of *decide and backtrack* [79]. In these algorithms, the central operation that moves the solving forward is the decision to set a variable to a certain value, and then examining the consequences of that for the rest of the formula. When no more easily found consequences can be obtained, one makes a new decision on a variable and the process is repeated until either a satisfiable assignment is found or when there is a conflicting consequence of the current variable settings.

In case of a conflict, one may backtrack and set the variables in a different way and continue the search. In the most basic version of this type of algorithm, the unsatisfiability of a set of clauses can only be proven after exhaustively going through each combination of variable settings and finding that every one of them leads to a conflict. The *DPPL algorithm*, which is named after its authors Davis, Putnam, Logemann, and Loveland contains, in addition to the backtracking, *pure literal elimination* and *unit propagation* [26].

In pure literal elimination, we take advantage of variables that occur always with positive or always with negative polarity (which makes it a *pure literal* in this terminology). If this happens with positive polarity, we can assign this literal the value *true*, as there are no opposing polarity literals. Similarly, we can assign this literal the value *false* if this happens with negative polarity. Doing this will make the clauses that contain the pure literal automatically satisfied and they do not have to be considered further for the satisfiability search. In this way, a part of the problem can be eliminated and the search space reduced.

Unit propagation makes use of the observation that a clause with a single literal (a *unit clause*) in it leaves the algorithm no choice: this literal has to be true to make the clause true and thus to satisfy the problem. This information can then be propagated to other clauses that contain the corresponding variable, as they now have one more true or false literal. The clauses with a true literal can be removed from the search. Clauses that contain the opposite polarity literal of the unit clause literal may also be simplified as the algorithm has just concluded that this opposite polarity literal cannot be true and therefore it can be removed from the clause. Deciding on a value for a variable can be seen as (temporarily) adding a unit clause, containing only the corresponding literal, to the problem.

In the most basic *DPLL* setup, we simply backtrack and try another variable assignment after we find a conflict. However, we may try to learn from the conflict, to find out what the cause of the conflict is, so that we may discourage the search procedure from trying assignments that do not work for the same underlying reason as that particular conflict. That is the central idea of the *conflict-driven clause learning* (CDCL) algorithm [78].

The core idea of the CDCL algorithm is still to decide on the truth value of certain variables and then to propagate the consequences. However, in a CDCL solver, the solver may learn from the conflicts. We show a partial solver state below, where we show some of the variable decisions and the part of the current clause set that is relevant to this current explanation. Imagine that during the solving process, we are forced by the previous decisions and propagations to conclude that $\neg e$ and e are both necessarily true in our current part of the

search space. That is clearly impossible and the origins of this conflict can be traced back. In this case, we concluded that $\neg e$ is true because we set d to false, and we concluded that e is true because we set f to false.

Current variable decisions: $\ldots, d = \mathit{false}, f = \mathit{false}, \ldots$
Clauses: $\ldots, \neg d \vee \neg e, \neg f \vee e, \ldots$

If we want to find a satisfying assignment, the solver must be prevented from setting variables in such a way that this current conflict is created again in the future. In this example, the solver must be denied the possibility of setting both d and f to false, i.e. the following condition must be enforced:

$$\neg(\neg d \wedge \neg f),$$

which can be rewritten to the clause:

$$d \vee f.$$

To make sure that the part of the search space that leads to the current conflict is blocked off, the *learned clause* $d \vee f$ is added to the clause set. While the current example was very simple, the same concept also holds for conflicts that are found much deeper in the branching search tree. This principle of learning conflict clauses is very powerful and used in modern SAT solving techniques to great effect.

1.1.6 SMT solvers

Up to this point, only values of *true* and *false* for our variables have been considered. However, in many settings, we may want to reason about variables with a different meaning. For example, we may want to ask questions about quantities: "Is it possible to fill each fruit basket with more than 5 apples under certain conditions?" We will take these integer number variables as an example of how to naively encode quantitative statements into propositional logic. We will start with a simple question: is the number x bigger than the number y? The question is now how to encode integer numbers into two-valued (Boolean) variables that can only be set to *true* or *false*. For this, the binary encoding can be used, where $\mathit{false} = 0$ and $\mathit{true} = 1$ (where one position, or one boolean variable of information, is called a *bit*). Note that these symbols 0 and 1 do not have the full meaning of the integers we usually associate with them (for example, the number 2 does not yet exist). To make it concrete, we will use a

2-bit integer that will encode the non-negative integers in the following (usual) way:

$$00 \equiv 0$$
$$01 \equiv 1$$
$$10 \equiv 2$$
$$11 \equiv 3,$$

which means that it is now possible to speak about the numbers 0 up to and including 3 in terms of Boolean variables. The next step is then to implement a comparison operation that allows the computer to decide whether one number is larger than another number. This can be done by asking whether the number x in binary notation has a lexicographically greater encoding than the other number y. This results in the following propositions, when calling the left bit x_1 or y_1 and the right bit x_2 or y_2:

$$\begin{aligned} &(x_1 \wedge \neg y_1) \vee \\ &((x_1 \wedge y_1) \wedge (x_2 \wedge \neg y_2)) \vee \\ &((\neg x_1 \wedge \neg y_1) \wedge (x_2 \wedge \neg y_2)). \end{aligned}$$

Put in natural language, the cases in which x is greater than y are:

1. The left bit of x is 1 and the left bit of y is 0.
2. The left bit of both x and y is 1 and the right bit of x is 1 while the right bit of y is 0.
3. The left bit of both x and y is 0 and the right bit of x is 1 while the right bit of y is 0.

The above propositional constraints can be converted into conjunctive normal form, for example by using the Tseytin transformation [96]. Note that it is also possible to use DeMorgan's laws (Section 1.1.1) to convert to CNF, but the resulting formula might grow much larger than the formula one started with.

The number of resulting constraints of such a naive Boolean encoding of integers can grow large (and here we have only implemented the 'greater than' operation, not the properties of multiplication and 'less than', and so on). In addition, we can only encode properties up to a finite range of integers, according to how many bits we used for the encoding.

One might want to have a convenient *type* of variable that automatically entails all the usual properties of integers. All the rules and constraints that

come with a certain type of object are called a *theory*. The theory can then be used at any time when reasoning about the specific properties of the type of object is necessary. Because the theory can be specialized and take the types into account, it is also possible to implement more efficient reasoning procedures that are not constrained by the propositional logic of the SAT solver. That is the logic behind the field of satisfiability modulo theories solvers (SMT solvers) [86]: to dispatch reasoning about specific theories to specialized solvers and treat the results that these routines return as propositional statements, with which a satisfiability solver can deal. For example, if there is a formula that contains the expression $5 > 7$, then an integer arithmetic theory would quickly conclude that the expression must be false, allowing the rest of the SMT solver to continue without keeping the details of the integer arithmetic theory in focus.

Most SMT solvers support several different theories and also combinations of theories: one might have a procedure that uses both the *bitvector* and the *integer arithmetic* theories, for example [6]. The SMT solver contains as its core inference engine a CDCL-based SAT solver, such as MiniSAT [33] or Cadical [10].

This separation of concerns allows a more user-friendly interface, where the system already knows, for instance, what integers are and what properties they are expected to have, instead of having to define those in the input problem in terms of propositional constraints. It also allows smart optimization procedures that can use the particularities of the domain theory to speed up the reasoning.

1.1.7 Quantifiers & First-Order Logic

The SMT approach allows us to encode many different types of problems into logical formulas to which automated reasoning can be applied. However, there are still more aspects to explain before the full power of mathematical reasoning is available to us.

The first of these is the introduction of *predicates*. A predicate is a kind of function that maps an object to the {*true, false*} options available in the propositional world. They can be used to implement all kinds of questions: one can imagine a predicate Animal(X), with X a variable, which returns *true* when the object is an animal, and *false* otherwise. It may also be the case that one wants to express a relation between two sets of objects, i.e. Knows(X, Y), which could stand for whether persons X know persons Y. Along with these predicates or relational symbols, we also have function symbols, which denote mathematical functions that do not necessarily output either true or false. Function symbols are used to construct more complicated terms. A

function symbol with zero input arguments (arity 0) is called a *constant*. In this context, a term is an expression constructed from variables, function symbols, and constants. For example, as a term, we may have a binary function f applied to two constants a and b: $f(a, b)$.

The last addition is the concept of *quantification*. Imagine that we are trying to encode knowledge about a particular problem that contains constraints in the form of integer values, for example, the dimensions of some industrial design problem or a number theory problem. We might then want to encode that a certain proposition is true for *all* integers. The "for all" concept is indicated with the *universal quantification* symbol $\forall$. For example, the following statement:

$$\forall \text{ Integer } X.\ (X < 0) \vee (X = 0) \vee (X > 0),$$

which encodes that all integers must be negative, equal to zero, or positive. Note that conceptually, the $\forall$ symbol can be thought of as invoking a large number of propositions, in this case, a proposition with three sub-conditions for each possible integer. Also note that if we had stated the conditions as:

$$\forall \text{ Integer } X.\ (X < 0) \vee (X > 0), \tag{1.1}$$

then we could have supplied the integer 0 for which the expression evaluates to false. The act of picking out specific members of the set of possible terms is called *instantiation*.

In addition to the $\forall$-type statements, there are also statements that are of the form "There is at least one such object such that some conditions are true". This is called the *existential* quantifier, $\exists$. For example:

$$\neg\ \exists \text{ Integer } X. \quad \neg((X < 0) \vee (X > 0)), \tag{1.2}$$

which says, "there does not exist an integer which is not negative or positive". The existential quantifier can be thought of as invoking many different sub-statements, all of them connected by an *or*. The attentive reader may have already noticed that statements 1.1 and 1.2 seem to mean the same thing: asking whether something is true for all instances is the same as asking whether there does not exist an instance for which it is false. The universal and existential quantifiers are thus related in a similar way to how DeMorgan's laws connect $\wedge$ and $\vee$.

Note that in the above example, we have used the Integer type to make the example less cluttered, even though in first-order logic this is not automatically included. The above example can either be seen as an SMT statement

with quantifiers where the theory of integers is included as a theory and can be invoked by using the Integer type, or as a shorthand rendering of a more complicated first-order formula where there are unmentioned parts of the problem that defined the Integer type.

As with propositional logic, there is a notion of a conjunctive normal form (CNF) for first-order logic. While we will not go into detail here, there exists a procedure to convert arbitrary first-order logic formulas into CNF (or *clausal*) form. In the clausal form, all existential quantifiers have been eliminated using a procedure called Skolemization and only universal quantifiers on the top level of the formula remain, and the formula is, like before, converted into a conjunction of disjunctions. A clause is a formula such as $\forall \vec{X}\ (l_1 \vee \cdots \vee l_n)$. A *ground clause* is a clause without variables and a *ground instance* of a clause can be obtained from a non-ground clause by substituting all variables with terms.

Herbrand's theorem: Now that first-order logic has been introduced, we can revisit the Davis-Putnam algorithm and the concept of resolution. The general idea behind the Davis-Putnam algorithm hinges on *Herbrand's theorem* [50][3].In the context of clausal (CNF) first-order logic, Herbrand's theorem says that a set T of clauses is unsatisfiable if and only if there is a finite set T' of ground instances of clauses in T which is unsatisfiable. Therefore, if we can find a finite, concrete set of ground consequences of the set of clauses that is unsatisfiable, we can conclude that the set of clauses is unsatisfiable.

The problem is then: how do we choose which ground instances we will evaluate? In some sense, a large part of the research in this thesis is focused on this exact question. In the original Davis-Putnam procedure, one simply starts enumerating all possible ground instances, occasionally calling a (resolution-based) SAT solver to check whether a propositional contradiction can be found. Chapter 2 included in this PhD thesis can be seen as a modern variant of this approach, where machine learning is used to generate the ground instances, whereafter a modern CDCL SAT procedure is used to check whether the problem is unsatisfiable. Chapter 3 of this thesis intervenes in the clause selection mechanism of the prover iProver, which also works on similar principles. Chapter 4 in this thesis also takes a similar viewpoint, where the enumerative instantiation procedure in an SMT solver is guided by machine learning to pick better instantiations.

[3]For historical accuracy, we note that an earlier use of Herbrand's theorem was by Gilmore's procedure [43]

1.1.8 First-Order Resolution & Equality

Instead of immediately starting to reason about the problem at the propositional level by producing ground instances, it may be more efficient to reason on the first-order level for a longer time [103]. The first-order quantifiers allow us to reason about infinite domains without explicitly handling all the terms contained in them. The resolution operation that is so central to the reasoning at the propositional level has a first-order version, which works in a similar way: when there are complementary literals in two clauses that may be unified with each other, we may resolve the two clauses. Unification is the process of finding a substitution that makes two terms with variables the same. Here, we will confine ourselves to the first-order version of *conjunctive normal form*, where there are only universally quantified variables scoped to each clause left after a preprocessing transformation that includes Skolemization, which is a way to eliminate existential quantifiers. In this notation, the actual $\forall$ symbols are subsequently dropped as as each variable is universally quantified over at the top level of the formula.

A small example where the unification of the two sides is simple because there are only variables (X, Y) in the literals:

$$\frac{\neg P(X) \vee Q(X) \qquad P(Y) \vee R(Y)}{Q(Z) \vee R(Z)},$$

where Z is a new variable in the resolvent clause that represents the unification of X and Y. The general first-order resolution rule is also applicable when there are, in addition to the complementary literals, an arbitrary number of other literals in the two clauses. As an example of an application with substitution, there is the following: if the right clause contained $P(a)$ instead of $P(Y)$, the first-order resolution rule would still apply, as the two terms can be unified by the substitution $X \to a$. That gives:

$$\frac{\neg P(X) \vee Q(X) \qquad P(a) \vee R(Y)}{Q(a) \vee R(Y)}.$$

If there is more than one possible substitution that unifies the terms, one usually uses the *most general unifier*. This is the unifier θ which unifies two expressions and is most general in the sense that for all other unifiers k we could find a substitution s such that applying s to θ gives k. In the following example, the resolution rule can be applied using a more complication substitution:

$$\frac{\neg P(f(X, g(Y))) \vee Q(X, Y) \qquad P(f(g(Z), V)) \vee R(Z, V)}{Q(g(Z), Y) \vee R(Z, g(Y))},$$

where the most general unifiers are $\{X = g(Z), g(Y) = V\}$.

In many kinds of problems, it might be useful to be able to encode that certain objects are equal under certain circumstances. While some provers implement more specialized equality reasoning [6], one can use additional equational axioms to reason about the equality of terms. Such extra axioms encode equality reasoning using axioms of reflexivity, symmetry, transitivity and extensionality:

$$\begin{aligned} A = A & \\ (A = B) &\implies (B = A) \\ ((A = B) \wedge (B = C)) &\implies (A = C) \\ A = B &\implies f(A) = f(B). \end{aligned}$$

In the last axiom, we use a function symbol f. At least one of these extensionality axioms is required for each non-constant function or predicate symbol. In the research discussed in Chapter 2, we made use of the equality reasoning capabilities of a modern prover to lighten the load on the machine learning component for the task of picking instantiations.

Given that there is equality defined on certain terms in the problem formula, the prover may rewrite using those equality rules to solve the given problem. For example, in Chapter 5 we show how a machine learning-guided equality rewriter can be applied to certain equational reasoning tasks.

1.1.9 Modern ATPs: Superposition and more

Many modern automated theorem provers for first-order logic use the *superposition* calculus [74], which can be seen as a combination of first-order resolution steps with equational reasoning. In this setup, the equalities between certain terms are used to rewrite parts of the problem. For example, given an equality between t_1 and t_2,

$$\dfrac{\dfrac{P(t_3, t_2) \qquad t_1 \simeq t_2}{P(t_3, t_1)} \qquad \neg P(X, t_1) \vee R(X)}{R(t_3)},$$

where X was substituted with t_3 by unification. In the superposition procedure, an ordering is imposed on the terms, so that the equality-based rewriting rules can only be applied in one direction on each pair of terms. In the example, we replace t_2 with t_1.

The procedure on a high level works as follows: select a *given clause* from the available clauses and perform an inference step of this clause with all previously selected clauses. This generates new clauses, which are added to the problem (often with some kind of redundancy check performed first).

The first-order provers Vampire [102], E [111], Prover9 [80], and others have this given clause loop as one of their main inference engines. The crucial part is to choose which clauses will become the given clause and much work has focused on finding good heuristics for this. For example, there has been machine learning-related work on doing this for E [58], Vampire [61], Prover9 [2], and other provers.

In this thesis, Chapter 3 concerns a learned clause selection heuristic in the instantiation-based first-order prover iProver [72]. The configuration of iProver used there does not use the same calculus as the superposition provers, but a part of the mechanism is similar to the given clause loop. We investigate whether it is possible to improve the instantiation calculus module of iProver by selecting better clauses.

1.2 Machine Learning

In this section, we will give a short introduction to machine learning, especially the type of machine learning technology that is necessary to understand the research chapters in this thesis: the graph neural network.

1.2.1 Learning By Example

In many situations, it would be useful to have a way to create, from a set of data points consisting of an input-output pair, an automated system that converts a certain kind of input to a certain kind of output. The process of creating such an automated system is known as *(supervised) machine learning*. In this sense of the term, learning means the search for a function that converts the input data into the output data. The goal is therefore to find a prediction function P that maps from the input domain I to the output domain O,

$$P : I \to O,$$

in a way that matches the data as well as possible. The input domain I may be an image in the form of a collection of pixels, a graph that represents a formula, an audio recording of an interview, or many other things that can be characterized by numbers. In the most basic machine learning setups, the

output domain O is generally a set of a few different classes that the input data may be separated into (which is called a classification task) or a numerical score that represents some useful quantity (in the case of a regression task). The question then becomes: how does one obtain a function P that matches the example data as closely as possible? First one has to fix a class of candidate prediction functions that will be under consideration.

In this thesis, we are mostly concerned with the class of machine learning predictors called *neural networks*, which is a broad term that incorporates roughly the mathematical functions that compose matrix multiplication operators with differentiable nonlinear activation functions. Usually, the exact architecture of the neural network, meaning the number of matrix multiplications and nonlinear activation layers and how their respective results are routed, is decided upon and fixed at the start of the procedure. The focus after this decision is on how to decide upon the exact values of the matrices used to define the multiplication operations inside the network so that an error function or a cost function is minimized. The process of finding suitable numerical parameters for these matrices is called *training* the network. The set of data points used to determine these parameters is called the *training set*.

While many different search procedures could be used, one of the most common methods used is that of *gradient descent*.

1.2.2 Learning By Stepping Away From Errors: Gradient Descent

In order to explain how neural networks are trained by gradient descent, let us first go to the example of a parabolic function. In high school mathematics, most people learn how to find the minimum of a given parabola: one can solve the quadratic equation that characterizes the parabola. However, although this works for parabolas, it does not work for all functions. It is wholly unavailable if one does not have an explicit form of the mathematical function to examine. In many cases, one can perform local steps to find the minimum.

Finding a minimum via gradient descent: One characteristic of a continuous function is its steepness at a certain point. This concept of the value of the slope of a function is called the *derivative*. In a function with multiple input parameters, the concept of *gradient* is usually used and expressed with the ∇ symbol which incorporates the *partial derivatives*. The derivative or gradient can be used to find a local minimum of a function. The intuition behind it is the following: if one wants to find the lowest point in a landscape, it is a reasonable strategy (for many landscapes) to keep walking in the direction that points

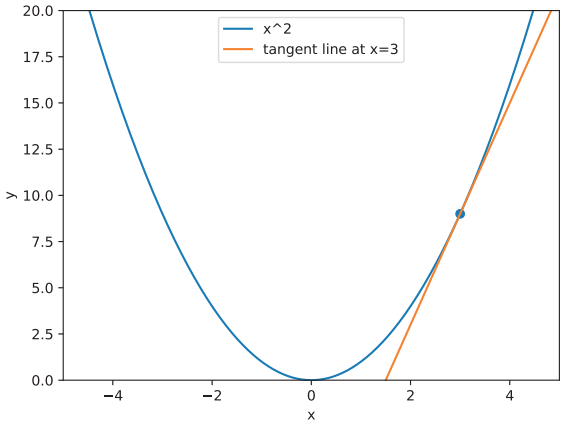

Figure 1.1: A parabola and the slope of the function at the value $x = 3$

downward. In our world, it is gravity that indicates what is up or down. In machine learning, there is a concept of a *loss function*, which is a differentiable function of the output of the neural network that quantifies the error of the predictions.

In Figure 1.1, we show a parabola f with the derivative at a certain point $x = 3$. As we are trying to find the minimum of the function, we will step in the direction that is opposite to the derivative (see the minus sign in Equation 1.3). The next choice is then how large the step in that direction should be. In the neural network literature, this step size is called the *learning rate* and is denoted by the symbol α. The update equation that governs how we change the parameter x according to the derivative therefore reads:

$$x_{t+1} = x_t - \alpha \frac{\mathrm{d}}{\mathrm{d}x} f(x_t). \tag{1.3}$$

By stepping in this direction, we can find a new x value that has a lower associated value for the parabola f. Note that if α is set to a value that is too large, we may end up with a value for x that is on the other side of the minimum. In general, to guarantee convergence of the training process, practitioners use decreasing learning rates, lowering them throughout the training process. In addition to overshooting the minimum (remember that in general, we do not know the overall shape of the function for which we are trying to find

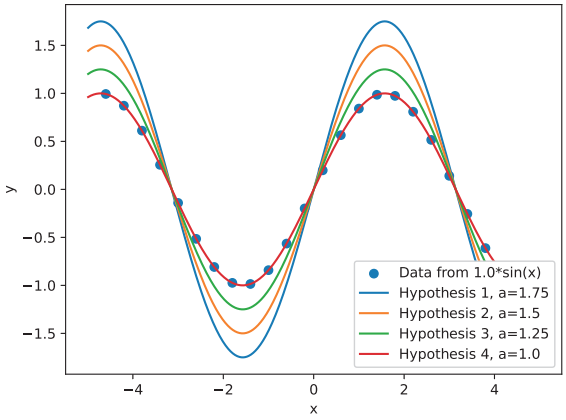

Figure 1.2: An example of different function values for our model $M = a{*}sin(x)$ when changing the parameter a. The red line is the function from which are data points are generated. This 'true' function can be unknown in machine learning problems. The dots are data points taken from the function that we want to approximate with our model.

a minimum), there is also a possibility that the minimum found is a sub-optimal local minimum.

Now, the perspective in the usual machine learning situation is slightly different: often the function that we are trying to minimize is an error function, a cost function, or a loss function that depends on the parameters of the network and the training data points. These functions describe the quality of a particular network output and are known at the start of the procedure. The training data points are fixed and the learning procedure changes the parameters of the network. We are thus trying to change the network's parameters so that the value of the loss function decreases and the quality of the network output increases.

In Figure 1.2, we show data points in the shape of a sinusoidal function. We also show a possible progression of functions with different a. In this example, we for simplicity assume that we chose the model $M = a * sin(x)$, which corresponds to us assuming that our sought-after function is a sinusoidal function with a single trainable parameter a. In the gradient descent procedure,

we would first pick a data point (x_k, y_k), calculate the difference between our model's output on input x_k and the example output y_k using a loss function such as the squared error. Then we calculate the derivative and update the value of a using a step as described in Equation 1.3.

Nonlinear functions: Neural networks are compositions of (matrix) multiplication operators and nonlinear functions. The crucial principle when choosing operations to incorporate into a neural network architecture is that one can easily calculate and chain the gradient functions for the operations: in that way, it will be possible to determine in which direction the parameters need to be changed to obtain a better set of parameters.

To give an intuition on how a neural network might encode non-trivial functions, we give the following example. Consider the following problem: we are trying to predict based on two binary variables. For simplicity, there are four data points, one for each of the possibilities. When one of the variables is true, but not the other one, the value is 1, and 0 otherwise. This is exactly the XOR function, or equivalently, the 'not equals' function.

Now, the task of the matrix parameters is to learn a function that uses a combination and transformation of the input parameters to predict the output as well as possible. The necessary function can be expressed in a standard *multilayer perceptron* architecture: first a matrix, followed by a nonlinear activation function, and then an output vector. Conceptually, each first-layer unit (of which there are two in this architecture, each with one weight for each input variable) can be thought of as a *feature extractor* that notices relevant patterns in the data. Possibly, there could be many more layers of these units that extract more and more high-level features and patterns in the data: this is the central concept in *deep learning* [76].

In Figure 1.3, we show a set of weights that computes the XOR function of two binary variables. To see how these were obtained, one can revisit Section 1.1.1 of this introduction and realize that the XOR function can be seen as $(a \vee b) \wedge (\neg(a \wedge b))$. This allows us to construct a weight setup that solves the problem: one first-layer unit must be equivalent to OR (Unit 1 in our case) and the other to AND (Unit 2 in our case), and the second-layer unit must somehow combine the positive result of the first neuron with the negative result of the second one. We have also introduced the concept of a *bias*, which is similar to the *intercept* of a regression function. These are parameters which are not dependent on the value of the input, which can for example be used to implement a threshold value.

In this example, we acted with previously acquired knowledge on how the function in the data could be decomposed into smaller relevant functions, but

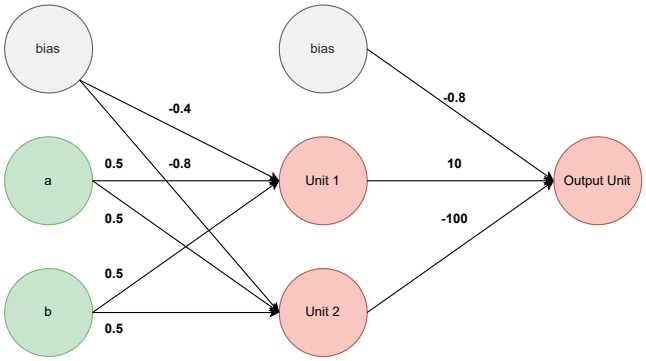

Figure 1.3: A small neural network that computes the XOR function. Each red unit multiplies its input with the corresponding weights, sums the results, and then applies a rectified linear unit function, $max(0, x)$. The output of the network is only positive when the input is (0, 1) or (1, 0), and 0 when the input is (0, 0) or (1, 1). Positive output of the network is interpreted as 1. For example, for the input (0, 0) Unit 1 computes $max(0, 0.5 * 0 + 0.5 * 0 - 0.4)$.

in general that luxury is not available: here, we have shown an example to help intuition on how nonlinear neural networks can represent patterns in the training data. In a typical machine learning task where neural networks are used, many different weights in many different layers are initialized to random values, which in many cases gives the optimization procedure enough options to find a reasonable setting for the weights that corresponds to an acceptable value for the loss function. Note that we rarely have the guarantee that the optimization procedure finds the 'real' pattern in the data (if we even have access to knowledge about such a pattern): the goal is to obtain an approximate function that mimics the relationship between the input and output variables in the data.

1.2.3 Structural Learning: Graph Data

The previous example concerned a type of input data shaped as a vector of (binary) variables. However, in some cases, the data used for machine learning is not easily capturable in such a form or has a different internal structure. One might make use of certain inherent structure in the data and have the representation of the neural network conform to that structure. One example is how in image processing, it can be useful to incorporate a spatial bias into

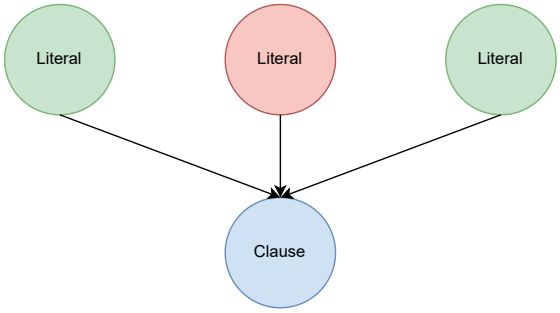

Figure 1.4: Schematic depicting the neighbourhood of one clause node. Representations of three literal nodes are used to update the representation of the clause node.

the network. The successful convolutional neural networks assume that pixels close to each other are more likely to form a useful pattern than pixels far away from each other.

The mathematical and formula data processed in the thesis can be interpreted as having a graph structure. For example, a function application to a constant can be represented as two graph nodes connected by an edge. The corresponding class of neural networks that deal with data structured as a graph is called *graph neural network*. In a graph neural network, there are two kinds of input data: the *features* associated with each node in the graph and the structure of the connection network between the nodes. The features associated with each node allow us to distinguish different types of nodes and input information about those nodes specifically.

The graph neural network uses the connectivity information to send messages from and to neighbouring nodes in a process called *message passing*. In this way, information can be shared across the nodes while taking the input graph structure into account.

For example, imagine that we have a SAT problem or a first-order problem in CNF form (for the example, it only matters that there are literals and clauses). For a given clause-type algorithm (see Section 1.1.9), the choice of given clause is important for the performance of the procedure. Therefore, learning a scoring mechanism for clauses can be useful.

In Figure 1.4, we show a schematic visualization of a small graph neigh-

bourhood to illustrate the message-passing principle. We show a clause, which in our graph is connected to the literals in the clause. Each node is initialized with a numerical feature vector according to its type. Each node is then updated with a function that takes as its arguments the current feature vector of its neighbours. The multiple incoming message vectors are combined using an aggregation function, which is permutation invariant (so without further modification, there is no ordering to the neighbours of the nodes). In the figure, we show a single message-passing step for a single node, but in practice, the vector representations of all nodes are updated according to their neighbours' node vectors. We have assigned colors to the literals, to imply some different characteristics that are relevant for the task of selecting clauses. Perhaps the representations of the literals in the clause indicate that they will be useful in a resolution operation (green color in the figure) or not (red color): the clause representation vector would have to reflect that so that a prediction of usefulness can be made from the clause vector. The example in Figure 1.4 is simple, but an example of a more complicated graph can be found in Chapter 2. In the literature, there are also examples of SAT problem graph encodings, including edges between the literal pairs with opposite polarity [117].

In Equation 1.4, we give an example of a possible node vector update rule with a sum aggregation function and a neural network layer Φ.

$$\mathbf{x}_i^{t+1} = \Phi\left(\sum_{j=1}^{N_i} \mathbf{x}_j^t\right), \tag{1.4}$$

where N_i is the number of neighbouring nodes for a node i. The neighbours are indexed by j and $\mathbf{x}$ is a vector of numbers that represents a node. In some architectures, the previous representation of node i is also used, which makes preserving values over time steps t simpler. The update rule as a whole assembles messages from neighbouring nodes, aggregates them into a single vector, and then uses a neural network layer Φ with a learnable matrix transformation and a nonlinearity to create the new node representation. At the end of a certain number of message passing steps, the representations are used to predict some quantity. For example, the representations of clauses may be used to predict whether they are a good candidate to be a given clause or a representation of a term or variable may be used to predict which instantiations are useful.

The expressivity of GNNs, in the sense of what functions a GNN can learn to express, is an active field of research [134]. Especially the choice of aggregation function can have an effect on what information is preserved.

1.3 Thesis Outline & Research Questions

We have introduced briefly most of the concepts needed to understand the main contributions of the research chapters that follow. On the automated theorem proving side, we have given an introduction to propositional logic, SAT solvers, SMT solvers, and first-order automated theorem provers. On the machine learning side, we gave a quick introduction to the main principles behind neural networks, one of today's most dominant machine learning methods. In particular, we have explained the graph neural network, which is suited to the type of data produced by automated theorem provers.

Automated theorem proving systems have to choose between many possible steps in their search for a proof. With machine learning, we can analyze the state of the prover and predict a course of action that will lead to a solution. In this thesis, we combine automated theorem proving with machine learning, in the form of graph neural networks. Of course, we are not the first to combine machine learning and automated theorem provers. There is a thriving subfield that concerns itself with the integration of machine learning into various kinds of provers [11, 55]. In this introduction, we will not go into further detail, as the chapters have their own sections that discuss related and previous works.

Many previous works focus on clause selection or premise selection as the target for a machine learning heuristic, often in a superposition-based proving system [45, 58, 123]. In our research, we have tried to focus on other aspects, such as instantiation-based proving methods. We also incorporate a modern machine learning method that is well suited to the type of graph-based data that represents prover states: the graph neural network.

The following research questions will be discussed in this thesis. The research questions are coupled to a Chapter in the thesis that corresponds to a research project. The corresponding articles are cited below. The contributions to the research projects made by me are specified in the separate 'Contributions & Publication List' Chapter on page 141.

- **How can proof instantiations be chosen by a neural network based on the input formula?**

  As we have seen, a crucial part of computer algorithms for proving first-order logic conjectures is choosing which instantiations to perform. The space of possible instantiations is difficult to navigate, and the question is whether it is possible to use machine learning to generate the instantiations based on previously successful proof instantiations for other problems. In Chapter 2, we explore this with a novel approach that combines

a name-invariant graph neural network used for abstract pattern recognition with a recurrent neural network used for term synthesis for concrete problems [94].

- **How can an instantiation-calculus prover be guided by machine learning?**

  In previous work, clause-selection guidance for automated theorem provers using a superposition calculus was constructed [58]. The question arises whether a similar approach also generalizes to other proof calculi. In Chapter 3, we discuss research that uses a graph neural network to perform clause-selection guidance in the instantiation-calculus prover *iProver* [23].

- **How can the instantiation process within an SMT solver be controlled by machine learning?**

  In SMT solvers, instantiation is an important step in the solving process. In this research project, described in Chapter 4, we tightly integrated a GNN into *cvc5*, where it controls both the selection of quantified expressions to instantiate and the term rankings that are used to decide which terms to use for the instantiations [93].

- **How can machine learning learn to control an equational rewriting prover?**

  In many applications, it may be beneficial to have a specialized rewriting engine or a preprocessor that handles a certain type of problem. In Chapter 5, we investigate whether it is possible to control an equational proving system and use it as a preprocessing step for a more general prover [92].

Chapter 2

Invariant Neural Architecture for Learning Term Synthesis in Instantiation Proving*

Abstract

The development of strong CDCL-based propositional (SAT) solvers has greatly advanced several areas of automated reasoning (AR). One of the directions in AR is therefore to make use of SAT solvers in expressive formalisms such as first-order logic, for which large corpora of general mathematical problems exist today. This is possible due to Herbrand's theorem, which allows reduction of first-order problems to propositional problems by instantiation. The core challenge is synthesizing the appropriate instances from the typically infinite Herbrand universe.

In this work, we develop a machine learning system targeting this task, addressing its combinatorial and invariance properties. In particular, we develop a GNN2RNN architecture based on a graph neural network (GNN) that learns from problems and their solutions independently of many symmetries and symbol names (addressing the abundance of Skolems), combined with a recurrent neural network (RNN) that proposes for each clause its instantiations. The

*This chapter is based on an article of the same name published in the *Journal of Symbolic Computation* [94].

architecture is then combined with an efficient ground solver and, starting with zero knowledge, iteratively trained on a large corpus of mathematical problems. We show that the system is capable of solving many problems by such educated guessing, finding proofs for 32.12% of the training set. The final trained system solves 19.74% of the unseen test data on its own. We also observe that the trained system finds solutions that the iProver and CVC5 systems did not find.

2.1 Introduction

Quantifiers lie at the heart of mathematical logic, modern mathematics and reasoning. They enable expressing statements about infinite domains. Practically all today's systems used for formalization of mathematics and software verification are based on expressive foundations such as first-order and higher-order logic, set theory and type theory, that make essential use of quantification.

Instantiation is a powerful tool for formal reasoning with quantifiers. The power of instantiation is formalized by *Herbrand's theorem* [50], which states that a set S of first-order clauses is unsatisfiable if and only if there is a finite set of ground instances of S that is (propositionally) unsatisfiable. Herbrand's theorem further states that it is sufficient to consider instantiations from the *Herbrand universe*, which consists of terms with no variables (*ground terms*) constructed from the symbols appearing in the problem. This fundamental result has been explored in automated reasoning (AR) systems since the 1950s [25,27,110]. In particular, once the right instantiations are discovered, the problem typically becomes easy to decide by state-of-the-art SAT solvers [73,79].

Coming up with the right instantiations is often difficult. As soon as there are non-nullary functions (e.g., the successor, $s(x)$) in a problem, its set of ground terms (the Herbrand universe) becomes infinite (e.g., $s(0), s(s(0)), \ldots$). There are typically many non-nullary functions in common mathematical problems. The general undecidability of theorem proving is obviously connected to the hardness of finding the right instantiations, which includes finding arbitrarily complex mathematical objects.

Investigating the interaction between symbolic computation, in the form of an automated theorem proving system, and machine learning, to learn heuristics to guide the choices of the provers, is a promising direction. In this way, both the

powerful exact reasoning of the prover and the approximate, heuristic reasoning of machine learning can be exploited.

Contributions: In this work we develop a completely naming-agnostic machine learning (ML) method that automatically proposes suitable instantiations. The system learns to instantiate from scratch, improving in an iterative fashion. This is motivated both by the growing ability of ML methods to prune the search space of automated theorem provers (ATPs) [67], and also by their growing ability to synthesize various logical data [38, 129]. In particular:

1. We decompose the problem of theorem proving into a neural instantiation step followed by a ground solver (Section 2.2.2). We devise an incremental procedure by which a predictor can propose instantiations (Section 2.2.4).

2. We develop a targeted GNN2RNN neural architecture based on a graph neural network (GNN) that learns to represent the problems and their clauses independently of many symmetries and symbol names (addressing the abundance of Skolems), combined with a recurrent neural network (RNN) that proposes for each clause its instantiations based on the GNN characterization (Section 2.2.5).

3. We construct an initial corpus of instantiations by repeatedly running a randomized grounding procedure followed by a ground solver (Section 2.2.2) on 113 332 clausal ATP problems extracted from the Mizar Mathematical Library. We analyze the solutions, showing that almost two-thirds of the instances contain newly introduced Skolem symbols created by the clausification (Section 2.2.8).

4. The GNN2RNN is trained and used to propose instances for the problems. Its training starts with the randomized solutions and continues by incrementally learning from its own successful predictions. We show that the system can predict the appropriate instances and that it can continually expand the set of proven problems, improving its performance (Section 2.3).

5. The trained neural network when combined with the ground solver is shown to be able to solve 19.74% of testing problems by making educated guesses (Section 2.3). The system finds solutions that were not found by existing systems such as iProver and CVC5. To our knowledge, this is the first naming-agnostic neural system for general synthesis of relevant elements from arbitrary Herbrand universes.

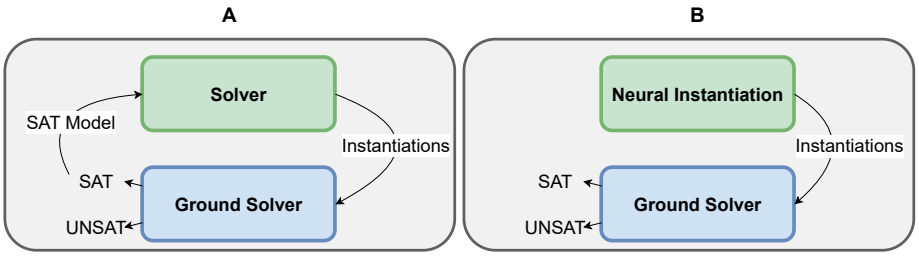

Figure 2.1: Solving problems by instantiation. On the left, in subfigure A, we show a general scheme, which fits certain modes for existing automated theorem provers such as iProver and SMT solvers such as CVC5. The first-order prover or SMT solver can generate instantiations and call the ground solver. The ground solver either finds a contradiction (and thus a proof) and returns UNSAT, or it finds a suitable interpretation of the problem and returns SAT with a propositional model, which the solver can use. On the right, in subfigure B, we show our approach, in which the instantiation is done solely by a neural component. Note that we currently do not use the SAT models generated, but are using a "single-shot" approach.

2.2 Methods

2.2.1 Solving by Instantiation

First, we give an overview of the problem and recall existing approaches, after which we explain our specific methods and solutions. For our examples and most of our treatment in this work, we will work in first-order logic, but instantiation can be used in a broader context, notably in higher-order and SMT solving [24]. In particular, we will work with clausal first-order problems: the problems are expressed as a conjunction of *clauses*. These clauses contain *literals* which are connected by disjunctions. The literals have as their head symbol a *relational symbol*, which denotes a predicate that assigns a truth value, or a negation, which negates that truth value. Under these symbols there are *terms* that can be constructed from a combination of *variables*, *function symbols* and *constants*. A term without variables is called a *ground* term.

The basic setting is that we have a set of (universally quantified) clauses that can be instantiated to a set of clauses without variables (*ground* clauses)

which is unsatisfiable. Consider the following set of clauses:

$$\forall x.P(f(x)) : \text{ Axiom 1} \tag{2.1}$$

$$\forall x.\neg P(x) \vee Q(x) : \text{ Axiom 2} \tag{2.2}$$

$$\neg Q(f(g(c))) : \text{ Negated Conjecture} \tag{2.3}$$

Axioms 1 and 2 can be instantiated to create the ground clauses $P(f(g(c)))$ and $\neg P(f(g(c))) \vee Q(f(g(c)))$ causing a propositional contradiction with the clause $\neg Q(f(g(c)))$. In this example, the substitutions $x \to g(c)$ and $x \to f(g(c))$ are not hard, but in general, the process of choosing the right instantiations of even a single quantified clause can be non-trivial. There can be many variables in one clause, and, in the general case, we have to choose the right instantiations for many clauses at once to arrive at a contradiction (and we might even need several different instantiations of the *same* clause). This is a problem of high combinatorial complexity, and there are several existing strategies to choose the right terms for the corresponding variables.

The automated theorem prover iProver is based on the *Inst-Gen calculus* [73], in which the first-order instantiation and grounding are interwoven with calls to a propositional SAT solver. The propositional model generated by the SAT solver is then used to decide how to instantiate. In SMT [24], there are several methods in use for instantiation, for example *enumerative instantiation*, which simply enumerates term substitutions, prioritizing the ground terms created earlier, *E-matching* [28], in which a pattern-matching strategy is used to decide instantiations, and *model-based instantiation* [42].

In Figure 2.1A, we show the general framework for these solvers. There is a solver, that can generate instantiations, which are passed to a ground solver (for example, a propositional SAT solver in combination with congruence closure procedures). This ground solver can produce SAT or UNSAT outcomes. In the case of SAT, the generated propositional model can be used to decide the next instantiations. However, it is difficult to decide what instantiations to use. In the next sections, we explain the components of our learning-based approach to this task.

2.2.2 Combining an Instantiator with a Ground Solver

Our general procedure is shown schematically in Figure 2.1B. We train a suitable neural architecture (Section 2.2.5) to do incremental instantiation (Section 2.2.4) which is combined with an efficient ground solver.

There are several ways to combine instantiation of clausal first-order problems with decidable and efficient (un)satisfiability checking of their proposed

ground instances. The most direct approach (used in instantiation-based ATPs such as iProver [72]) is to explicitly add axioms for equality, allowing their instantiation as for any other axioms, and directly use SAT solvers for the ground checking. An alternative approach is to avoid explicit addition of the equality axioms, and instead use combinations of SAT solvers with ground congruence closure [31, 85].

We have explored both approaches and ultimately decided to use the latter in this work. The main reason is that the combinations of SAT solvers with ground congruence closure are today very efficiently implemented [8], posing practically no issues even with thousands of instances. Using the most direct approach would, on the other hand, require a large number of additional instances of the equality axioms to successfully solve the ground problems. In our preliminary measurements, the average ratio of such necessary additional instances was over 40%, which would exponentially decrease the chance of predicting the right set of instances.

2.2.3 Ground Solver for CC + SAT

We use as our ground solver an efficient combination of a SAT solver with ground congruence closure (CC + SAT). In CC + SAT, the SAT solver abstracts the ground atoms as propositional variables and starts producing satisfying assignments (models) of this abstraction, which are then checked against the properties of equality (reflexivity, symmetry, transitivity, congruence). This process terminates when a model is found that satisfies the equality properties, or when the SAT solver runs out of models to try. In that case, the original ground problem is unsatisfiable.

2.2.4 Incremental Instantiation Procedure

We first give a high-level view of how the neural network predicts the instantiations. The neural prediction is decomposed into levels, where each level deepens the non-ground terms. The idea is depicted in Figure 2.2. At each level, the network predicts a single function symbol for each variable of a given clause C. Then, a new instance C_1 of C is created by replacing the variables with the proposed function symbols and fresh variables as their arguments. This whole process is iterated.

Besides the increasing depth, the network also needs to be able to deal with an arbitrary number of variables in each clause. This is handled in an RNN fashion—variables are being predicted in a fixed order and the information

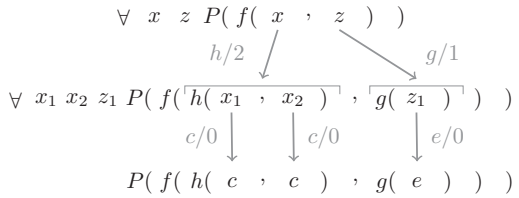

Figure 2.2: Term instantiation through incremental deepening. In the figure, there are two instantiation steps, one after the other.

about the previous predictions is stored in a hidden state. Note that the number of symbol predictions needed increases or decreases depending on the arity of the predicted symbols. In particular, if all predicted symbols are constants (symbols of arity 0), no new variables are generated and the process stops. We are most often required to predict instances for multiple clauses at the same time, as well as in some cases multiple instances for the same clause.

As an example, this iterative process would be able to generate the term $f(c, (f(c, c))$ starting from $f(x, y)$, but it would take 2 levels: one in which x is set to c and y is set to f, and another level in which two fresh variables generated under f are both instantiated with c.

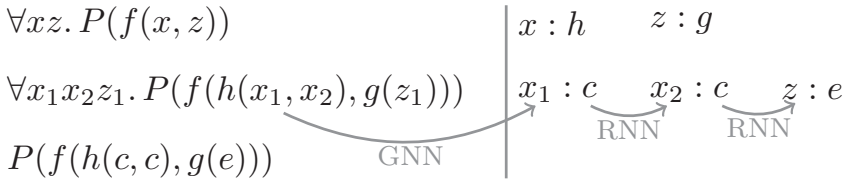

Figure 2.3: Schematic of RNN Predictions corresponding to Figure 2.2. A GNN communicates via variable representations to the RNN predictor and previous predictions within the same clause are communicated by an RNN hidden state. Figure 2.5 shows a more detailed procedure.

2.2.5 Neural Network Architecture

Mathematical problems often have many symmetries, making them challenging for the naive use of off-the-shelf sequence-based learning methods. Clausal problems are invariant under the reordering of clauses and literals, renaming of variables in each clause, and also under consistent renaming of symbols in a problem. To address this, we base our architecture on a graph neural network (GNN) with such properties proposed by [88] and used so far in several ATP tasks [21, 57] to *classify existing objects*. In this work, we re-implement that GNN in PyTorch [90], and add to it a novel anonymous, signature-bound recurrent neural network (RNN) that allows us to also *generate new objects* (clause instantiations). The relation between the graph neural network and the RNN is illustrated by Figure 2.3. To our knowledge, this is the first ML architecture combining such strong invariant and nameless problem encoding with the need for non-anonymous symbolic decoding (synthesis).

Next, we detail the neural network architecture. Code and data to run the system is available on Github at `https://github.com/JellePiepenbrock/neural-synthesis`.

Graph Neural Network

The GNN architecture we use is specifically constructed to be invariant to symbol names, as it treats the input clauses in a fully anonymous manner [88]. In addition, the network has a notion of negation. The particular structure of clausified first-order logic problems is taken into account, with clause nodes able to communicate with literals, terms able to communicate with their subterms, and all symbols able to directly communicate with all terms they are in. The graph neural network is invariant to clause order permutations and literal permutations.

In Figure 2.4, we show a schematic graph corresponding to the example in Section 2.2.1 that indicates how the nodes are connected, from the graph neural network's perspective. For reading convenience, we repeat here the three clauses from that example: Axiom 1: $\forall x.P(f(x))$, Axiom 2: $\forall x.\neg P(x) \vee Q(x)$ and the negated conjecture $\neg Q(f(g(c)))$.

There are *term nodes* $\mathbf{T}$ (three types of these are distinguished in the representation: 'standard' terms, variables and literal), the *symbol nodes* $\mathbf{S}$ (which can be function symbols or relational) and the *clause nodes* $\mathbf{C}$ (which can be axioms or conjecture clauses).

When a problem is processed by the graph neural network, each node in the graph is first given an initial numerical vector representation according to

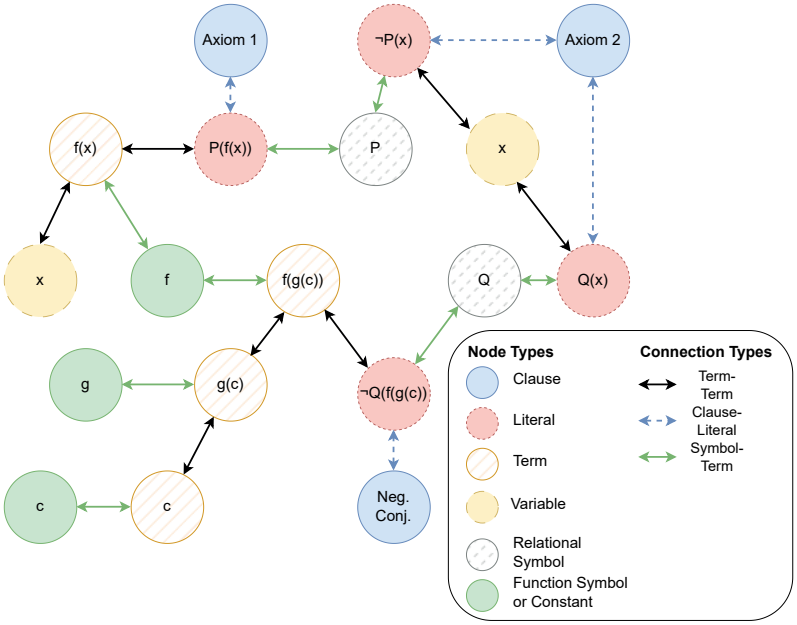

Figure 2.4: A schematic representation of the graph corresponding to the example in Section 2.2.1. Note that several aspects of the GNN's message passing structure have been suppressed for visual clarity, such as the treatment of polarity and the handling of argument ordering in terms with function arities higher than 1.

its type. The representations of these nodes are then updated according to the representations of each node's neighbours (a process called *message passing*). Each node aggregates the incoming vectors by aggregation operators, in our case the element-wise *mean* and *max* functions. After aggregation, a linear transformation in the form of a matrix with learnable parameters is applied to the aggregated representation. Then a non-linear *activation* function is applied which creates an updated vector representation. This update procedure can be iterated, so that nodes that are not direct neighbours can also update according to each other's representations.

Different matrices are used for the transformation in different connection types (for example, the transformation that updates clause representation has different parameters than the transformation that updates symbol represen-

tations.) During the training procedure, these parameters will be tuned to produce node vector representations that are useful to predict the instantiation of clauses.

For a full exposition of the update equations used, we refer to Section 2 of [88]. For our purposes here, it is enough that after several message passing rounds, the GNN outputs updated vector representations for the nodes in the graph. These vector representations of the nodes, that can incorporate structural information about the graph surrounding each node, are then used to predict how to instantiate the clauses.

RNN Function Symbol Prediction

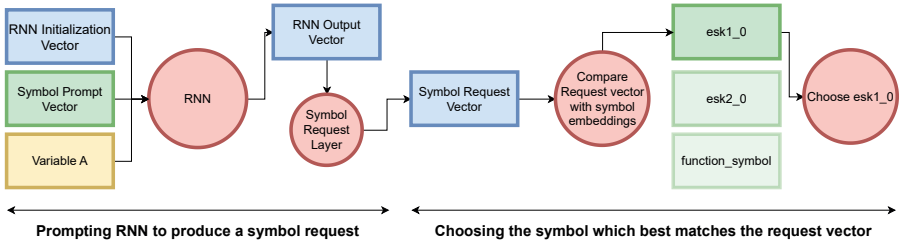

Figure 2.5: RNN architecture, shown predicting a symbol for Variable A, the first variable in a clause. A special, trainable *symbol prompt vector* is used to mark a step where the RNN must predict a symbol for the queried variable. The vectors representing Variable A and the symbols $esk1\_0$, $esk2\_0$ and $function\_symbol$ are created by the GNN. GNN and RNN are trained end-to-end as one. Variable vectors are colored yellow and symbol-related vectors are green. Blue indicates that the vector is an intermediate vector or an RNN state. Red indicates a transformation, calculation or a choice.

We modify the original GNN architecture to allow the network to produce instantiations for each clause by using an RNN after running the GNN (Figure 2.5). The setup is as follows for the first variable in each clause. We predict an *output vector* using the RNN by taking for every clause the representation of the first variable that occurs (in de Bruijn order) from the set $\mathbf{T}$, a special *symbol prompt vector* and an initialization vector.

This RNN output vector is processed by the *symbol request layer* to give a *symbol request* vector. The symbol request layer is a neural network linear layer that has as its task to transform the RNN output vector into a vector similar

to the vector representing the required symbol. We compute the dot product between this request vector and the representations of each function symbol in the signature. We then apply the softmax function to get a probability distribution over the function symbols for that variable. We then continue with the next step of the procedure (see Figure 2.6), where the RNN gets its own output from the previous step, the chosen symbols, as well as the representation of the second variable.

Conditional Prediction

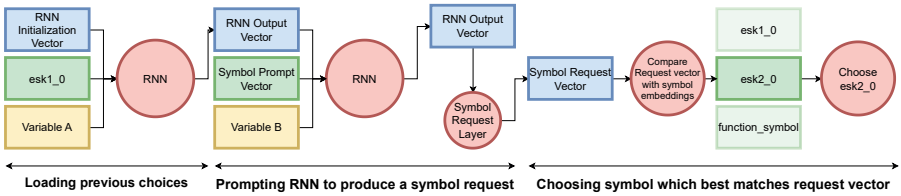

Figure 2.6: RNN architecture, shown in the process of predicting a symbol for the second variable B in a clause. $esk1\_0$ and $esk2\_0$ are two symbol node representations (for Skolem constants). Here $esk1\_0$ was chosen for Variable A. Before using the *symbol prompt vector*, the prefix of currently assigned symbols is shown to the RNN and the RNN can predict a symbol for variable B conditioned on the choice for A.

After having chosen a symbol for the previous variables, we want to choose the next symbols while taking into account our earlier choices. To create symbol predictions that are conditioned on the symbols that were already chosen for other variables in the current clause, we created the setup as shown in Figure 2.6. There, we schematically show the network in the process of predicting a symbol for the second variable (B) in a clause. First, the network gets as its input an initialization vector, a variable representation vector and the representation vector of the symbol that was already chosen (esk1_0) for that variable (A). The RNN then produces an output vector that should encode all the information about these prior decisions that is necessary to predict the next symbol. In the second step, the RNN processes this output vector, a vector representation of the second variable and a special *symbol prompt vector* that indicates that the loading of previous decisions has ended and that we are currently expecting a new symbol prediction. The RNN then produces a new output vector, which we process as a symbol request using the symbol request layer. A symbol is then

chosen according to the probability distribution induced by the dot products of the request and the symbol vectors, as before.

During training, we maximize the probability of choosing the symbol used in the known proof. During evaluation (Section 2.3) we can either (i) *decode greedily*, choosing the maximum probability symbol, (ii) *sample symbols* according to the distribution defined by the model (which introduces some randomness), or (iii) use a *beam search* procedure to find the most likely sequences. In this work, we always use the sampling method (ii), as it can produce varied output and is easier to implement than beam search.

This procedure preserves the anonymity of the entire setup: the system is invariant to naming. In the end, we obtain a mapping of variables to symbols for each clause. In addition, the network can choose a special *stop* symbol (represented by a special trainable vector) when shown the first variable, which indicates that the clause should not be instantiated. This means the RNN predictor is first performing a premise selection task and then an instantiation task when the clause is selected. After all variables are processed, we show the first variable again. The RNN can either produce stop, or another symbol to continue producing another instance for the clause. If the RNN predicts a stop symbol at a point where only some variables have been processed, we do not use the sample. For our purposes here, we only produce instantiations where all variables in a clause have been replaced by a symbol (which possibly has new variables associated with it).

2.2.6 Dataset of Mathematical Problems

We use a dataset of 113 332 first-order ATP problems made available to us by the AI4REASON project.[2] They originate from the Mizar Mathematical Library (MML) [66] and are exported to first-order logic by the MPTP system [128]. All these problems have an ATP proof (in general in a high time limit) found either by the E/ENIGMA [57,112] systems or by the Vampire/Deepire [102,123] systems. Additionally, the problems' premises have been *pseudo-minimized* [64] by iterated Vampire runs. In this procedure, input clauses are dismissed when they are not necessary for the proof. We use the pseudo-minimized versions because our focus here is on guiding instantiation rather than premise selection. The problems come from 38 108 *problem families*, where each problem family corresponds to one original Mizar theorem. Each theorem can have multiple minimized ATP proofs using different sets of premises. The problems range from easier to challenging ones, across mathematical fields such as topology,

[2] `https://github.com/ai4reason/ATP_Proofs`

set theory, logic, algebra and linear algebra, real, complex and multivariate analysis, trigonometry, number and graph theory.

While we have 113 332 problems in the full dataset, a smaller subset was selected to allow quicker iteration for some of our experiments. The problems selected correspond to 2003 Mizar theorems known as the M2k subset, which is a subset of related Mizar articles [67]. Since we typically have multiple premise selections proving the same Mizar theorem, 4817 problems constitute the dataset which we will refer to as our *M2k Dataset*.

2.2.7 CC + SAT Implementation

CC + SAT is a standard procedure used in state-of-the-art SMT solvers such as CVC5 [6]. After some experiments comparing several CC + SAT implementations, we have chosen the implementation provided in the Vampire system [74, 132]. The choice for using Vampire as CC + SAT backend was made after also testing CVC5. We have found that Vampire's parser was faster, while there is no difference in the solved problems. We always use a 30s time limit for the ground solver and a faster implementation allows a higher number of ground instances to be given to the CC + SAT system. For more information, see also Appendices A.1, A.2 and A.8. Note that all clauses with variables are removed before calling Vampire in its CC + SAT mode.[3]. This means that Vampire is really used only as a standard CC + SAT solver for ground problems and no other parts of Vampire are being used for instantiation. In Section 2.2.4, we explained how our procedure can produce non-ground clauses at each level, which could potentially be made ground at the next level by instantiation with constants.

2.2.8 Generating Training Data Via Random Grounding

To create our initial training dataset of instantiations, we use a randomized grounding procedure. We first clausify the problems using E [112], and then repeatedly run a randomized grounding procedure on all of them, followed by the ground solver.

Randomized grounding can be parameterized in various ways. To develop the initial dataset here we use a simple multi-pass randomized grounding with settings that roughly correspond to our ML-guided instantiation architecture (Section 2.2.5). These settings are as follows. We use at most two passes (levels) of instantiation for every input clause. In the first pass, for each variable

[3]We call Vampire with the *acc* argument.

we randomly select an arbitrary function symbol from the problem's signature, and provide it (if non-constant) with fresh variables as arguments. In the second pass, we ground all variables with randomly selected constants from the problem's signature. The first pass is repeated 25 times for each clause in its input, and the second pass 5 times. The input to the first pass is the original clausal problem. The input to the second pass is the (deduplicated) union of the clauses produced in the first pass and of the original clauses.

This means that each input clause can produce up to $(25 + 1) \times 5 = 130$ ground instances, potentially resulting in ground problems with thousands of ground clauses. Such input sizes typically pose no problems to the ground solver. The average ground problem sizes are however typically below 1000, because of the overlaps and limited number of options during the random grounding.

The first run solves 3897 of the problems, growing to 7790 for the union of the first 9 runs, and to 11 675 for the union of the first 100 runs. The 113 332 problems have on average 35.6 input clauses and the 3897 problems solved in the first run have on average 12.8 input clauses. There are 6.0 instances needed on average to solve a problem.[4] Note that each clause can in general be instantiated more than once. Also, 3.9 (almost two-thirds) of the instances contain on average at least one Skolem symbol. For these symbols, the name is not necessarily informative. These data statistics confirm that we would strongly benefit from a learning architecture invariant under symbol renaming, rather than off-the-shelf architectures (e.g., transformers) that depend on fixed consistent naming. This motivates our use of a naming-invariant architecture (Section 2.2.5). Appendix A.3 contains some more information on the random instantiation.

2.2.9 Neural Network Data Processing and Training Details

In the following subsections, we will discuss practical details of the neural network training process, such as the data split for evaluation purposes, the preparation of the data, the balancing of the loss contributions from different samples and hyperparameters. Appendices A.5, A.6 and A.7 contain some more details on the neural network choices and the training.

Data split

For the machine learning experiments, the Mizar theorems were split into 90% training and validation data and 10% testing data. Of the largest set, 5% of

[4] Over 40 instances (max. 63) are used in some problems

data was used as the validation set and 95% as training data. Note that our split keeps problems which are versions of the same theorem in the same section of the split, so that data leakage between minor variations of the same proof idea is prevented. Also, **the fact that a problem is assigned to the training set does not imply we already have a proof from the randomized grounding: the split is done independently of the availability of a solution.** This means that each of these sets is a mixture of problems that already have a proof from the randomized grounding and problems that do not have one. In the end, we have 96 532 training, 5293 validation and 11 507 test problems. For the M2k subset, we have 4163, 188 and 466 problems in the respective sets.

Hardware

All experiments were run either on a DGX machine with 8 NVIDIA Tesla V100 GPUs with 32GB memory, 512GB RAM and 80 cores of Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz type (looping and running provers) or on a machine with 4 NVIDIA GTX 1080 GPUs with 12GB memory, 692GB RAM and 72 cores of Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz type (only training).

Data Preparation

The initial training data were the cumulative proofs obtained by 9 runs of the random instantiator with 25 samples on level_0 and 5 on level_1. For the full dataset, there are 6592 solved training problems, while for the M2k set, there are 421. If multiple proofs were found for a problem, one of those proofs was chosen randomly. Note that the number of solved problems here can be lower than the number of available training problems. If a given proof needed two levels of instantiation, the proof is split into 2 training examples E_1, E_2. The input part of E_1 is the original CNF problem, while for E_2 it is the CNF problem with the right head symbols (and corresponding fresh variables) filled in for the proof-related clauses. For E_1, the label part corresponds to the head symbols in the input of E_2, while the labels for E_2 are the constants that ground the terms. The parameters of the neural network are trained by minimizing the cross-entropy between the predicted distribution and the labels.

When there are multiple instantiations for the same clause needed for the proof, we concatenate all the proof instantiations for a clause into a single sequence. The RNN component was trained to predict this concatenated sequence of instantiations so that the model can capture the conditional dependence between multiple instantiations of one clause. To make it possible for the model to

stop instantiating a clause, a special *stop* vector is added in addition to the actual symbol vectors when comparing with the symbol request vectors. The label corresponding to the *stop* vector was added to the end of each label sequence. For clauses where no instantiation is part of the proof, the label sequence is only the *stop* label. Therefore premise selection is included in the task.

While we choose to handle multiple instances for a clause sequentially with an RNN, there is no ordering on these instances for a given clause. Therefore, during training, we randomize the order in which the different label sequences corresponding to different instances are concatenated. Each time a sample is encountered, we use a randomized order (so over the training process, many different orderings are observed, as a form of data augmentation). In principle, the same holds for the ordering on the variables, but this ordering is not randomized in the current setup, to limit the computational resources needed. The variable order is as they appear in the clause.

Loss balancing

The number of choice points, and thus the number of contributions to the total loss when naively added, is not the same for each training example. Some training examples require as many as 100 symbols to be chosen, while others need less than 10. Therefore, we normalize the loss contribution coming from each training example in the batch by the number of choice points (i.e. the total length of all the concatenated label sequences for all clauses with variables in the training example). We then minimize the sum of these averages. This loss corresponds better to our use case: it is more important to get all 3 instances for a small problem, than it is to get 3 out of 80 instances for another bigger problem.

Hyperparameters

The GNN was used with node and layer dimensions, for all nodes, set to 64. The network uses 10 message passing steps, with different layer parameters at each step. Each different node type is initialized with a different trainable initial embedding of 64 numbers. The RNN consists of a linear neural network layer with 192 inputs and 64 output dimensions followed by a rectified linear unit activation function, followed by a linear layer of input size 64 and output size of 64. The symbol request layer is a linear layer with 64 input and 64 output dimensions. To optimize the parameters of the network, the ADAM algorithm [68] was used with *learning rate* 0.0001, minimizing the cross entropy between the symbols used in the known proofs and the predicted symbols. The

maximum number of RNN iterations per input clause (which limits the total number of symbols chosen per clause) was set to 12.

Parameter settings such as layer size and learning rate were determined by increasing them (decreasing in the case of learning rate) until the network reliably converged to a low loss on the training dataset, indicating a capacity to fit the data. In the end, we settled on 10 layers and 64-wide embedding dimensions, starting from smaller ones. While this seems enough to fit the data reasonably, we did not test networks that were deeper or wider than this. It is possible that more performance can be gained with a more extensive examination of the various hyperparameters.

2.3 Results

To characterize the performance of our setup, we carried out several experiments. First, we explain the training of the initial network from the dataset created by randomized grounding and how we selected specific predictors. We also measure how well the predictors can generalize to samples not in the training data.

Second, we show the results of an iterated loop on the M2k dataset, where we take this trained predictor and use it to predict solutions, add the new solutions to the training data, and repeat. We use the same CC + SAT setup as with the randomized grounding procedure. After showing that this iterated procedure works, we also iterate on the full dataset, where we show how the system can self-improve up to the point of proving some theorems existing automated theorem provers did not find in 60s.

2.3.1 Prediction Model Selection

For the looping and evaluation experiments, we use the models that have the earliest highest median validation accuracy (indicating that after this, the model could start overfitting). For the predictor trained on the M2k set, this happened after 56 passes over the training data. For the training on the full dataset, we use checkpoint 79, for the same reason.[5]

[5] In Appendix A.4, we show a plot of accuracy during the training process.

2.3.2 Validation and Instance Accuracy of the Neural Networks

The training setting does not fully reflect how the model is used for the real instantiation task. In the training procedure, accuracy is computed while the input and the previous choices are always fully correct (i.e. according to the labels). However, in the practical setting, the model can only run on its own previously generated, and possibly suboptimal, output. This is explored in Table 2.1.

Table 2.1: Coverage of the instantiations needed for the proof, on problems from the validation set that the random instantiator found a proof for. The columns indicate the maximum number of instantiations per clause (which are sampled from the probability distribution learned by the model). The rows are split into the results for level_0 and level_1. The accuracy corresponding to the quantiles 0.1, 0.5 and 0.9 are given. Note that even if the instantiations of the proof in our validation set for a given problem are not covered, the predicted instances might constitute a different proof.

No. samples / clause	1 inst.	5 inst.	10 inst.	25 inst.
level_0 - q=0.1	0.08	0.40	0.53	0.67
level_0 - q=0.5	0.50	0.83	1.00	1.00
level_0 - q=0.9	1.00	1.00	1.00	1.00
level_1 - q=0.1	0.00	0.21	0.50	0.82
level_1 - q=0.5	1.00	1.00	1.00	1.00
level_1 - q=0.9	1.00	1.00	1.00	1.00

The results are for the model trained on the data from the proofs generated by the random instantiator, but applied on unseen validation data and compared to unseen proofs from the random instantiator (which may have 2 levels of instantiation). We show the fraction of label instances covered as a function of the number of samples taken per clause (more simply, the accuracy). The accuracy values correspond to certain quantiles q. For example, a 0.67 score for $q = 0.1$ means that for 10% of all problems, at least 33% of the required instantiations are still missing. While the median ($q = 0.5$) and the 90th quantile indicate that with 25 samples, most instantiations are covered, the $q = 0.1$ data show that there is a subset of problems for which instances are missing. We also see that, given correct instantiations on level_0, filling in the right instances on level_1 (bottom 3 rows) is easier than starting from the base problem (top 3

rows). In general, there is an indication that the predictor has learned the task. Next, we show our iterative self-improvement experiments.

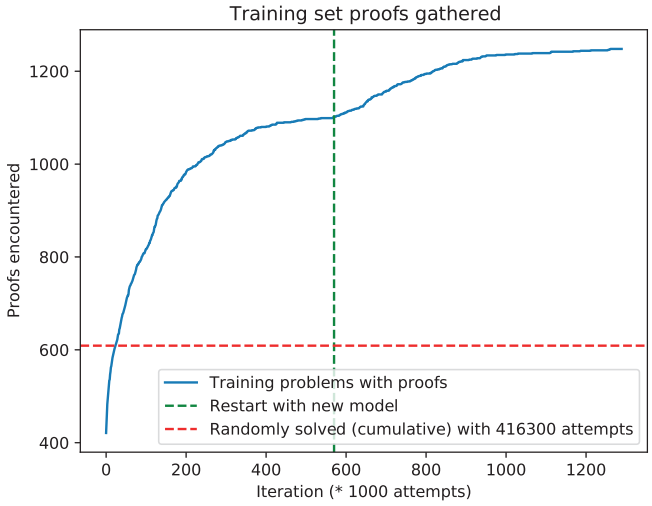

Figure 2.7: Number of problems with a solution (cumulative, M2k). Horizontal dashed line indicates the number of problems solved by random attempts and vertical dashed line indicates the time at which a newly trained model was introduced.

2.3.3 Self-Improving Loop (M2k Dataset)

The previous section indicates that the model has learned to recreate the right instances for many proofs on unseen data. Next, we test whether the system can generate new proofs, and then learn from these new proofs, in a self-improving loop. To test this capability, we do a looping experiment. We use two levels of instantiation, limiting the maximum number of instantiations per clause to 25 and 5 respectively. This way we can compare to the random instantiator baseline, which also uses 2 levels of instantiations with the same limits.

We run the system on problems from the training set, including those where the random instantiator could not find a proof. We keep all the proofs, and train again including the new proofs. This procedure is then repeated.

Because one full loop iteration can take time, we first test this setup on the M2k subset. There are 4163 problems derived from the M2k theorems assigned to the training set. Of these, the random instantiator solved 421 total in 9 runs (this is the initial training data). In 100 runs (416 300 attempts total), it found 609. In each iteration, 1000 problems are attempted and 1000 random previous proofs are trained on (but the model parameters are kept between iterations). Every 10 iterations, we also run the proof attempts on the test set.

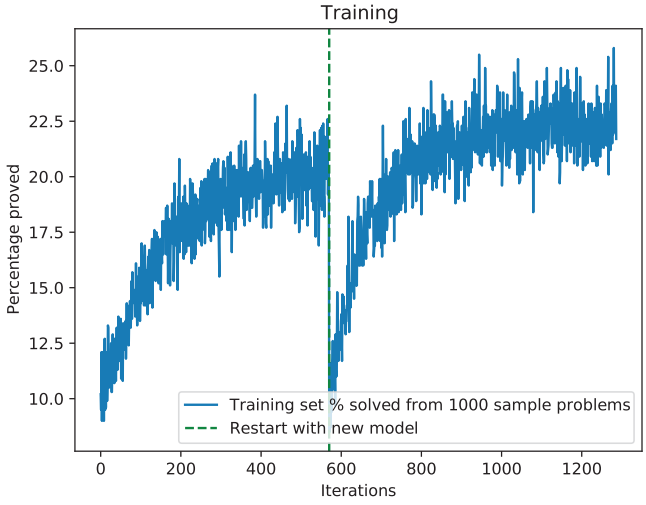

Figure 2.8: Percentage of problems solved (training set problems, M2k)

After 580 loop iterations, there are 1102 problems from the training set that we have a proof for (see Figure 2.7). However, the discovery of new proofs stagnates after around 400 iterations. We concluded that restarting the training with a fresh model might bring more new proofs, as by this point the training solutions had been seen many times, which could lead to an overfit model. The restart is therefore a chance for the model to find a new optimum that possibly better incorporates recently added solutions. As seen in Figure 2.7, this restarts the process and the system finds 146 more proofs, for a total of 1248. The system found almost double the amount of proofs as the random instantiator (dashed red line) in the same amount of iterations, indicating that the learned distribution of terms can lead to more proofs.

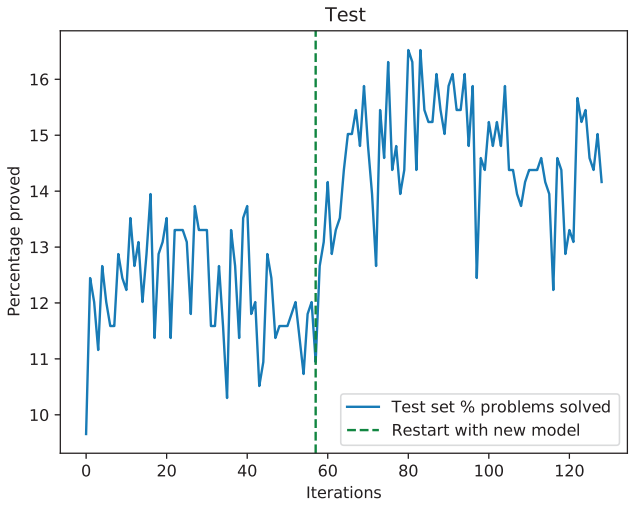

Figure 2.9: Percentage of problems solved (test set problems, M2k)

In Figures 2.8 and 2.9, the non-cumulative training and test set performance of the system are shown. The system learns how to prove around 22% of training problems, which corresponds roughly to the ratio between 1248 solved training problems and 4163 total training problems. Most training problems that were solved once can be solved reliably after training on their solutions. On the M2k test set, the behavior is similar, although the unseen data makes it harder: the system can prove 14.5%, which is more than double the performance of 1 run of the random instantiator with the same sampling settings (6.9%). The extra performance gain after the restart is noteworthy. This indicates that the optimization found a more generalizable optimum when trained from scratch with the solutions found before the restart of the self-improvement loop.

2.3.4 Self-Improving Loop (Full Dataset)

As the self-improvement loop was successful on the M2k dataset, the experiment was repeated for the larger, full dataset. For this larger experiment, the system uses 5 levels of instantiation, with $(25, 5, 5, 5, 5)$ maximum samples per clause for the respective levels. There are 10 000 proof attempts and 10 000 training proofs are randomly selected from all proofs at each iteration.

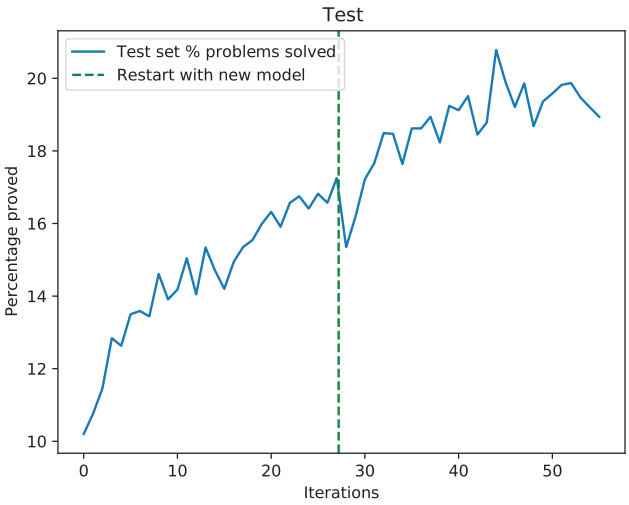

Figure 2.10: Percentage of problems solved (test set problems, full set, sample of 10 000 problems taken per iteration).

We start from the 6592 training problems solved by the random instantiator. After 550 iterations, with a restart of the model included, there are 31 003 problems solved, more than 4.5× the number of solved problems the procedure started with. This means that the system cumulatively found proofs of 32.12% of the training dataset. With the predictor from the last iteration, the system can prove 26.25% of training set problems. Most of the previously cumulatively solved training problems are solved by the final system. In Figure 2.10, we can see the progression of test performance. Of the full set of unseen test problems, the final system can solve 19.74%.

The depth of the iteration procedure, or the number of levels, has an effect of the number of proofs found. In general, allowing more levels improves the performance. However, the number of clauses that serve as input for the next level potentially grows as fast as the limit on the number of samples per clause. Therefore, the smaller the number of levels, the faster the system runs in general. We tried depths from 2-5, but there were diminishing returns in terms of number of proofs and rising computational costs, which is why 5 was chosen as the limit.

Table 2.2: Performance of various methods. iProver is used in pure instantiation mode. Random is 1 run of the 2-level random grounding. In parentheses, we indicate which dataset was used.

Time limit	1s	10s	60s	Inst. + 30s
Random (all)	-	-	-	3.44%
Neural (train)	-	-	-	26.25%
Neural (test)	-	-	-	19.74%
iProver (train)	43.28%	59.99%	67.6%	-
iProver (test)	43.16%	59.75%	68.69%	-
CVC5 (test)	83.44%	85.6%	86.28%	-

2.3.5 Comparison with existing provers

Our system takes about 1 second per level of instantiation, plus a maximum of 30s for the ground solver (though most ground problems are solved in under 1 second). We compare the performance of our system with existing automated provers. Table 2.2 shows the percentage of solved problems for various solvers and Table 2.3 shows how many problems our system solved that were not solved by the existing automated theorem provers. For each prover, there are some solutions that our system adds. For iProver in instantiation mode, the set of added solutions is the largest, but even for a highly optimized SMT solver like CVC5, we can add some solutions. With the final cumulative 31 003 solutions, we add 6821 new solutions compared to 1s iProver on the training problems, or a 16.33% gain. On the test set (non-cumulative, last iteration predictor) we add 317, or a 6.38% gain.

If we compare to the longest iProver run, there are still 97 test set problems not found by iProver in 60s. Some types of problems seem to be dominant: 15 of the new solutions are about asymptotes, while 9 are about quaternions and 7 are about complex numbers. While these types of problems are somewhat prevalent in the dataset, they are not the most prevalent. Problems about finite sequences, euclidean geometry and group theory are much more numerous.

2.4 Related Work

In addition to the recent general work on synthesis of logical data mentioned in Section 2.1, there is recent work on choosing instantiations in automated reasoning using ML, but with a different focus than our work. Several examples

Table 2.3: New solutions gained compared to existing provers. Solutions already contained in the random instantiator runs that serve as the starting training data are not counted.

Gain (# / %)	vs. 1s	vs. 10s	vs 60s
vs iProver (train cumul.)	6821 / 16.33	3459 / 5.97	2471 / 3.79
vs iProver (test)	317 / 6.38	159 / 2.31	97 / 1.23
vs CVC5 (test)	75 / 0.78	73 / 0.74	73 / 0.74

are found in the SMT community, where gradient boosted tree algorithms were used to filter possible terms [17, 34] and to rank them for the SMT solving procedure [63]. These however work within the solving loop of an existing SMT solver, whereas we are synthesizing instances, attempting to do most of the non-ground reasoning within a trained neural network.

There is also work on synthesizing loop invariants, which is similar in spirit to what is attempted in this work [118]. A difference is that we are synthesizing many objects (instances of each clause) simultaneously, whereas loop invariant synthesis is more concerned with a single object. Also, the specific grammar of loop invariants used is limited, but we jointly learn synthesis over arbitrary function signatures, within a single signature-invariant system.

2.5 Conclusion

We have developed a fully neural instantiation mechanism for many clauses at the same time. Starting from data generated by randomly instantiating variables in problems from a real-world mathematics dataset, the machine learning component can learn how to instantiate and improve based on its own newly found proofs. The final system, after self-improvement on the full dataset, can solve 26.25% of the training dataset, starting from a core of problems comprising 6.83 % obtained by a random instantiator. The system can also solve 19.74% of the unseen test problems, indicating generalization capabilities. The total pool of solved problems is more than 4.5× larger after the self-improvement loop.

The combination of a neural instantiator with a strong ground solver with a congruence closure mechanism combines two techniques according to their respective strengths: the graph neural network that can learn global heuristics for the instantiation of first-order variables is combined with the fast and optimized reasoning components of SAT-based solvers to propagate the consequences of

the instantiations.

While the current system does not outperform highly engineered existing systems, such as iProver or CVC5, in absolute terms, we do gain new proofs compared to them. This indicates that the system has learned how to synthesize instantiations that are hard to reach for these systems. Taking these results in combination with the demonstrated self-improvement capabilities of the architecture, we believe that this opens up a new research direction of theorem proving systems that incorporate learned neural synthesis, applicable not only in instantiation, but also in other settings such as tableaux and saturation-style proving.

A possible future direction of research could be to close the loop in Figure 2.1B: the neural network's prediction of instantiations could benefit from explicit information about the SAT model. Another direction could be the direct integration of a similar neural instantiation method into an SMT solver.

2.6 Acknowledgements

This work was partially supported by the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15 003/0000466 (JP, JU), the European Union under the project ROBOPROX (reg. no. CZ.02.01.01/00/22_008/0004590) (MJ), Amazon Research Awards (JP, JU), by the Czech MEYS under the ERC CZ project POSTMAN no. LL1902 (JP, MJ, JU), EPSRC grant EP/V000497/1 UK (KK), and the EU ICT-48 2020 project TAILOR no. 952215 (JU).

Chapter 3

Guiding an Instantiation Prover with Graph Neural Networks*

Abstract

In this work we extend an instantiation-based theorem prover iProver with machine learning (ML) guidance based on graph neural networks. For this we implement an interactive mode in iProver, which allows communication with an external agent via network sockets. The external (ML-based) agent guides the proof search by scoring generated clauses in the given clause loop. Our evaluation on a large set of Mizar problems shows that the ML guidance outperforms iProver's standard human-programmed priority queues, solving more than twice as many problems in the same time. To our knowledge, this is the first time the performance of a state-of-the-art instantiation-based system is doubled by ML guidance.

*This chapter is based on the article of the same name published in the *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR) 2023* [23].

3.1 Introduction

In the recent years, machine learning (ML) and related AI methods are increasingly combined with automated deduction. One of the most interesting tasks in this area is equipping fast state-of-the-art automated theorem provers (ATPs) with *efficient internal guidance* of their calculi based on learning from many previous proof-search decisions. This is challenging, because the fast ATPs typically generate and evaluate thousands to millions of inferences within seconds. While related AI/TP tasks such as learning-based *premise selection* [1], *tactical guidance* [41], and *neural conjecturing* [129] can use relatively slow and expensive ML methods that are called only rarely on a single problem, internal guidance requires efficient ML methods and their nontrivial integration with the fast ATPs.

In the last five years, several advances in internal guidance were made for connection-based [67, 99, 135] and resolution/superposition-based ATPs [45, 59, 123]. However, there has so far been only limited success in guiding instantiation-based systems, which are – thanks to the integration with powerful SAT solvers – today becoming very competitive [30].

In this work we therefore develop strong internal ML guidance for one of today's main instantiation-based theorem provers: iProver [72] (Section 3.2). In more detail, the ML guidance is based on naming-invariant graph neural networks (GNNs) (Section 3.3). To combine the two, we develop an interactive mode in iProver, which allows communication with external agents via network sockets (Section 3.4). The GNN guidance is then implemented as an instance of such an external agent – a GPU server (Section 3.5). Our evaluation on a large set of Mizar problems (Section 3.6) shows that the ML guidance doubles the performance of iProver's standard human-programmed priority queues (Section 3.7). To our knowledge, this is the first time a state-of-the-art instantiation-based system is considerably improved by neural ML guidance.

3.2 iProver

3.2.1 iProver and the Inst-Gen Calculus

iProver [72] is an automated theorem prover for quantified first-order logic. At the core of iProver is an instantiation calculus, Inst-Gen [36, 73], which can be combined with resolution and superposition calculi [32]. The Inst-Gen calculus is based on propositional reasoning to deal with propositional abstractions of first-order clauses and model-guided incremental instantiations using unification

to generate new first-order instances. At a high level, the procedure works as follows.

Given a set of first-order clauses S, its propositional abstraction $S\bot$ is obtained by mapping all variables to a designated ground term $\bot$. A propositional solver is applied to $S\bot$ and it either proves that $S\bot$ is unsatisfiable and in this case the set of first-order clauses S is also unsatisfiable or shows that $S\bot$ is satisfiable and in this case returns a propositional model of the abstraction $S\bot$. This propositional model is analysed if it can be extended to a full first-order model. If it can not be extended then it is possible to show that there must be complementary literals in the model that are unifiable. In this case the Inst-Gen calculus produces instances of relevant clauses with the most general unifier which resolves this conflict in the model and provide sufficient information to the propositional solver that this conflict will not occur in the future. This loop is repeated with more instances added until the unsatisfiability is witnessed by the propositional abstraction or a saturated set is obtained (possibly in the limit) in which case the original first-order formula is satisfiable. The Inst-Gen calculus is *refutationally complete*, which means that if the set of first-order clauses S is unsatisfiable then in a finite number of iterations, the propositional abstraction of derived instantiations $S'\bot$ will be unsatisfiable [36].

3.2.2 Guiding iProver

iProver leverages the power of propositional solvers for a) solving propositional abstractions, and b) guiding instantiations based on propositional models. Although this approach often works well in practice, one of the major bottlenecks is the number of generated first-order instances with only a few of them usually needed in the final proof.

In this work we therefore investigate how machine learning can be used to select iProver inferences that are most likely to be used in a proof. In particular, the propositional model typically leads to multiple Inst-Gen inferences that can be made to provide sufficient information to the propositional solver that will avoid particular conflicting assignment of unifiable literals in the propositional model. Figure 3.1 shows an overall scheme of the approach. The ML advice will be used to select the clauses for performing the Inst-Gen inferences (see Section 3.4 for details).

3.3 Name-Independent Graph Neural Network

To learn a machine learning heuristic for clause selection in the Inst-Gen calculus, we use a graph neural network [106]. Specifically, we use a PyTorch [90] implementation of the name-independent architecture developed originally for connection provers [88], but also used for the recent ENIGMA systems for the E automated theorem prover [46].

In this work, we will give a brief primer on the main ideas underlying this neural network architecture, so far as they are directly relevant to the current work. Given a first-order logic problem expressed as a set of clauses named *Clauses*, we parse the problem and create a (hyper)graph that represents the problem. In this representation, three different types of nodes are distinguished. These are the clause nodes $\mathcal{C}$, the symbol nodes $\mathcal{S}$ and the term nodes $\mathcal{T}$. The collection of all nodes is the set $\mathcal{N}$. Each clause node corresponds to a clause in the input file. Symbol nodes correspond to either function or predicate symbols.

The structure of the original first-order problem is reflected in the edges connecting these different types of nodes. This allows the network to see how the symbols are used without knowing their names, and therefore handle common ATP issues such as the typically quite unstable naming of Skolem symbols between problems. It is also important when used in ITP-based (*hammer*) scenarios, where new terminology is introduced frequently during the formalization (see Section 3.7.2). There are several different types of edges between the nodes. Clauses are connected to their literals, while the polarity of the literals is explicitly handled (so the network has a built-in notion of negation). Symbols are connected to the terms they are used in. For example, given a term $f(t_1, \ldots, t_k) \in \mathcal{T}$, with k subterms labelled as t_i, the nodes corresponding to the function symbol $f \in \mathcal{S}$ will be connected to t_i via the term node $f(t_1, \ldots, t_k)$, for $1 \leq i \leq k$.

This graph representation then contains the relationships between the various mathematical objects occurring in the problem. The basic idea of using this representation to learn a heuristic for clause selection is to let each node exchange messages with its neighbours, to update some state representation according to a learned transformation parameterized by weights that are learned by gradient descent on an error function. This message passing is iterated for a fixed amount of steps, after which the representations for clauses are used to predict whether the clause is useful or not. Initially, each node is represented by a vector of floating point numbers (embedding), which differs based on the type of node. We distinguish the following node types: conjecture clauses, non-conjecture clauses, function symbol nodes, predicate symbols, terms, literals

and variables.

After some message-passing steps, the predicted probability of a clause being useful for the proof is computed by another learned transformation that takes the final representation of each clause node and combines this with the representation of the conjecture clauses. This predicted score for the clauses is then used to influence iProver's clause priority queues. The machine learning system is trained by minimizing, through *gradient descent*, the *binary cross-entropy* error function that measures how well the network can predict which clauses are useful for the proofs and which ones are not. In Appendix B.3 we give more details on GNN settings.

3.4 Interactive Mode

We have newly developed an Interactive Mode for iProver, which is in detail described in our repository.[2] In this mode, iProver communicates with an *external agent (EA)* (e.g., ML-based, that can be written in any language, e.g., Python) via TCP/IP sockets. The external agent can be used to provide proof search guidance by either assigning scores to clauses which are used for prioritising them for the next inferences or explicitly selecting the given clause for the next inferences. The communication is bi-directional:

- iProver submits different messages to the agent, such as the given clauses, generated clauses, simplified clauses etc.
- The agent can guide the iProver search by different actions such as selection of the given clause, assigning scores for clauses in passive queues etc.

Figure 3.1 details the given clause loop extended with the external guidance. The Input clauses are first submitted to the Unprocessed set and simplified (simp I). Then, the EA evaluates the clauses and assigns scores that are used as priorities in *passive priority queues*. These are used to store Passive clauses, i.e., the clauses that are waiting to be involved in inferences. Each priority queue is based on a lexicographic combination of different clause features, such as: the number of literals, the number of variables, clause age, proof distance from the conjecture, etc. Priority queues are combined in a round-robin fashion with specified multipliers. The EA scores are treated as one of the clause features and can be used in a standalone priority queue based on just EA scores or combined in queues with other clause priority features.

[2] `https://gitlab.com/korovin/iprover/-/blob/master/README-interactive.md`

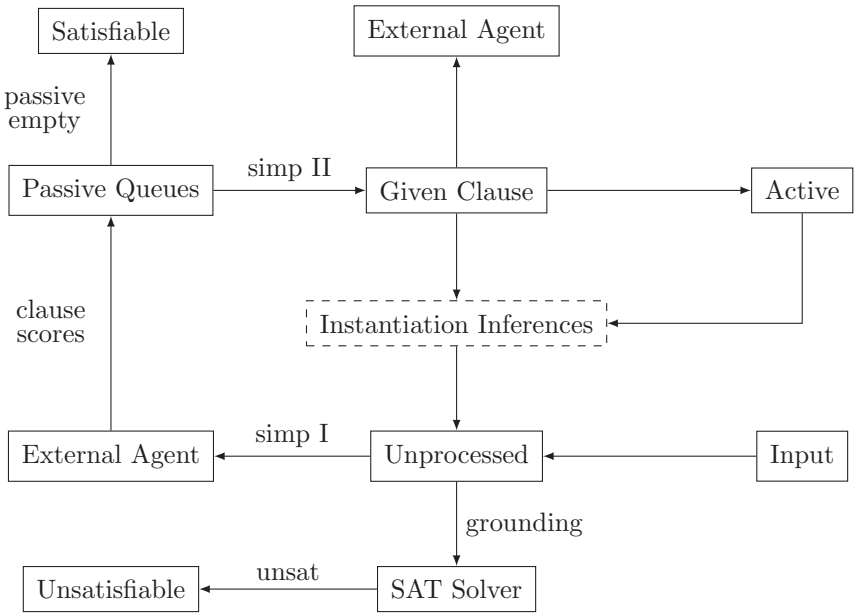

Figure 3.1: Interactive Given Clause Loop in iProver

The Given Clause is the clause selected from the priority queues for the next inferences with clauses stored in the Active set. The Given Clause is also submitted to the EA to be used as a part of the *context* for the next rounds of evaluations. After all inferences between the Given Clause and clauses in Active are performed, the Given Clause is moved to Active and all newly derived clauses are moved to the Unprocessed set. Groundings of clauses from the Unprocessed are also submitted to the SAT solver which is executed with some specified intervals. If the SAT solver returns that the accumulated groundings are unsatisfiable, then the Input set of first-order clauses is also unsatisfiable and the problem is solved. If the Passive Queues are empty then due to the completeness of the procedure [36, 73], the Input set is satisfiable.

The problem of selecting the most suitable given clause is the "Holy Grail" of automated theorem proving: with perfect clause selection the proof can be directly reconstructed. In most cases, though the clause selection is far from perfect which results in an explosion in the search space. One of our main contributions is to show that an External Agent based on GNN models can be trained to select clauses considerably better than human-based fine tuning of priorities.

3.5 GPU Server

We have developed[3] an external agent that uses a trained GNN to score clauses generated in the given clause loop. The main function is similar to E/ENIGMA, where a Python GPU server [45] is used to reduce the overhead of repeated model loading. However, in ENIGMA, the score requests are created as full graphs already in E, where the *context* consisting of the given clauses and conjectures is added to the generated clauses that are to be scored, and hence the server is *stateless*. The sole purpose of the server there is to receive the queries, evaluate them using a GPU, and send the results back. Here, we instead build a richer *stateful* server that keeps track of the given clauses and conjectures, and is itself capable of generating in various ways the context that is used to score the requests.

The agent is a Python server that contains three types of processes. The *main process* checks a network socket and distributes the incoming connections to the *state processes*. Each state process handles one client (iProver) connection. Whenever the state process receives a score request, it adds the context

[3] `https://gitlab.ciirc.cvut.cz/chvalkar/iprover-gnn-server`, `https://github.com/JellePiepenbrock/iprover-gnn-server`

of the given clauses and conjectures and sends a request to a *GPU process*. To create contexts, the state process must keep all the generated clauses, given clauses, and conjectures in its memory. The GPU process then evaluates the clauses using a GNN preloaded on a GPU and sends the scores back to the state process that communicates them to the client iProver. While this architecture is more involved than the simple ENIGMA server, it is more flexible, allowing different parameterizations and experiments with the GNN guidance. We show in Section 3.7 that the overhead incurred by this more involved guiding architecture is reasonably low, and that its use results in very high *real time* improvements of iProver's performance. See also Appendix B.1 for more details on the server settings.

3.6 Experimental Setting

3.6.1 Evaluation Problems

The evaluation is performed on a large benchmark of 57 880 problems[4] originating from the Mizar Mathematical Library (MML) [66] exported to first-order logic by MPTP [128]. The Mizar problems are split[5] (in a 90-5-5% ratio) into 3 subsets: (1) 52k problems for *training*, (2) 2896 problems for *development*, and (3) 2896 problems for final evaluation (*holdout*). We use this split for the evaluation done here. Since we are interested in internal ATP guidance and not premise selection, we use problems with premises limited to those used in the human-written Mizar proofs (*bushy problems*). As an additional measure of the generalization, we also evaluate (Section 3.7.2) the trained system on 13 370 theorems in 242 articles that were added in a newer version of MML (1382) and thus never seen during the training. More than half of those problems contain new terminology.

3.6.2 Data: Classic vs Dynamic

The training data are collected from previous successful runs. The standard approach (*classic data*), which was introduced by the ENIGMA systems, is to take all given clauses and consider those that ended up in the proof as useful (*positive clauses*) and those not used as useless (*negative clauses*). Hence, for each proof, we have one training example (a graph of clauses) containing the useful/useless clauses and the conjecture we want to prove. However, the size

[4] `http://grid01.ciirc.cvut.cz/~mptp/1147/MPTP2/problems_small_consist.tar.gz`
[5] `http://grid01.ciirc.cvut.cz/~mptp/Mizar_eval_final_split`

of such a training example corresponds to the *final* size of the set of all given clauses proposed in the proof search, not to the *intermediate* sizes of the *actual* score requests generated multiple times during the proof search. This means that in the classic setting, the distribution of the training graph sizes may be shifted quite far from the actual distribution encountered during the proof search. Such subtle shifts between the training and evaluation distributions have quite often a negative effect on the performance of more complex machine learning architectures.

To remedy that, we experiment with using the actual score requests (*dynamic data*) from the proof searches as the training data. However, such score requests contain not only the given clauses but also the passive clauses. It is not always clear how the passive clauses should be labeled, since they may or may not lead to an alternative proof. For simplicity, we use a common pragmatic approach and consider them as negatives. We also want to prevent the learning from focusing on too many minor alternative proofs of the same problem and neglecting problems that have fewer proofs. When we learn from multiple collections of proofs (produced in multiple iterations of the proving and learning over the training data), we sample these proofs so that each problem contributes the same number of training examples.

3.6.3 iProver Settings

We run iProver using its instantiation mode.[6] The score requests are performed in batches of size approximately 1000 to improve the performance. For details, see Appendix B.2. For our evaluations of the baseline non-guided iProver, we run it with its standard human-programmed priority queues for clause selection in the Inst-Gen calculus in three different modes. These modes are the non-interactive mode, the interactive mode without ML evaluation, and the interactive mode with ML evaluation that is ignored.

In the *non-interactive mode*, iProver does not communicate with the server. In the interactive mode without ML evaluation (*no eval*), the ML server returns zeros without any evaluation. The purpose of this mode is to measure the overhead caused by the communication protocol and processing requests. In the interactive mode with ignored ML evaluation (*ignore eval*), the server evaluates clauses but the scores are ignored by iProver. By running in this mode, we can investigate the overhead incurred by the ML calculations.

[6] Recent iProver can also use superposition and resolution and combine them with instantiation. However, in this work our focus is to establish if instantiation-based methods can be effectively guided by learning.

In the setting where the scores provided by the server are used for clause selection, we use two modes: *solo* and *coop*. In the solo mode, there is just one priority queue ordered by the scores provided by the server. In the coop mode, we combine server-provided scores with the human-programmed priority queues in an equal ratio, see Appendix B.2.

3.7 Results

As a starting reference point, we run iProver with the human-programmed priority queues for clause selection to collect the initial training examples. In this mode, without being slowed down by the server (non-interactive mode), iProver solves 502 theorems out of the 2896 theorems in the development set (Section 3.6.1) in 15 seconds. However, to extract the training data, we need to run iProver in the interactive mode. It solves 451 and 482 theorems in the development set in ignore eval and no eval modes, respectively. Therefore, there is some overhead from the communication and GNN calculations, but its impact is manageable.

Using the no eval mode, we get 9087 training proofs and also 482 proofs in the development set that are used as a validation set for finding the best performing model. We train three models, one with classic data and two with dynamic data (taking randomly 4 or 10 queries from the successful runs, respectively). Each model is run in either solo, or coop mode. Two best performing models on the development set were trained on dynamic data using 4 and 10 samples and run in the coop mode, see Table 3.1. We use these two models to obtain further training data for the next iteration. It is worth mentioning that models trained on dynamic data perform significantly better than models trained on classic data.

In the next iteration, we use all proofs found so far by iProver in the no eval mode together with the proofs found using the two best performing models. This yields 14 994 and 834 proved theorems on the training and development set, respectively. We again trained three models, from which the best performing was the model trained on the dynamic data using 4 samples run in the solo mode. The model using 10 dynamic samples run in the coop mode was reasonably complementary, and we also used it for the next iteration.[7]

In the last iteration, we trained on 18 452 theorems solved (using possibly multiple proofs available for each theorem) on the training set and used 1026

[7]It is likely that the model with 10 dynamic samples performed worse than the model trained on 4 dynamic samples per problem, because models in this iteration were trained for a shorter period of time.

problems solved on the development set for evaluating the best performing models. In this iteration, we also trained dynamic models with an increased size of embeddings (from 16 to 32) and the number of layers (from 10 to 11). Increasing the size of embeddings leads to a better performance, see Table 3.1, but increasing solely the number of layers does not. However, the best performing models come from increasing both the size of embeddings and the number of layers.

3.7.1 Holdout Set Performance

Table 3.1: Proving-learning iterations and their performance on the devel and holdout sets. Model parameters are the size of embeddings (d) and the number of layers (l).

Iter.	Solver (15 s)	Data	Model	Devel	Holdout	Train
	Ignore eval			451	455	
	No eval			482	475	9087
	Non-interactive			502	500	
0	Solo	classic	$d = 16$, $l = 10$	663	656	
0	Coop	classic	$d = 16$, $l = 10$	699	704	
0	Solo	dynamic (4)	$d = 16$, $l = 10$	714	723	
0	Coop	dynamic (4)	$d = 16$, $l = 10$	744	729	13 403
0	Solo	dynamic (10)	$d = 16$, $l = 10$	739	739	
0	Coop	dynamic (10)	$d = 16$, $l = 10$	760	759	13 534
1	Solo	dynamic (4)	$d = 16$, $l = 10$	951	945	16 964
1	Coop	dynamic (10)	$d = 16$, $l = 10$	834	835	14 953
2	Solo	classic	$d = 16$, $l = 10$	674	689	
2	Coop	classic	$d = 16$, $l = 10$	739	741	
2	Solo	dynamic (4)	$d = 16$, $l = 10$	1004	1017	
2	Solo	dynamic (4)	$d = 16$, $l = 11$	1003	987	
2	Solo	dynamic (4)	$d = 32$, $l = 10$	1028	1032	
2	Solo	dynamic (4)	$d = 32$, $l = 11$	1033	1032	
2	Coop	dynamic (4)	$d = 16$, $l = 10$	955	945	
2	Coop	dynamic (4)	$d = 16$, $l = 11$	945	942	
2	Coop	dynamic (4)	$d = 32$, $l = 10$	984	990	
2	Coop	dynamic (4)	$d = 32$, $l = 11$	988	983	
2	Solo	dynamic (10)	$d = 16$, $l = 10$	1018	1022	
2	Solo	dynamic (10)	$d = 16$, $l = 11$	922	901	
2	Solo	dynamic (10)	$d = 32$, $l = 10$	1068	1063	
2	Solo	dynamic (10)	$d = 32$, $l = 11$	**1094**	1093	
2	Coop	dynamic (10)	$d = 16$, $l = 10$	955	956	
2	Coop	dynamic (10)	$d = 16$, $l = 11$	897	883	
2	Coop	dynamic (10)	$d = 32$, $l = 10$	1018	1024	
2	Coop	dynamic (10)	$d = 32$, $l = 11$	1037	1034	

iProver using guidance from our best performing model solves 1094 problems on the development set and 1093 on the holdout set. Moreover, it solves a very similar fraction of problems on the training set. Similar results hold also for other models, see Table 3.1.

The training procedure seems to be quite robust to overfitting on the training data and to generalize well. This could be due to several aspects: (i) different runs lead to different sets of given clauses, (ii) only a limited number of the dynamic samples is seen during the training per problem (4 or 10), and (iii) the contexts are randomly sampled from the available given clauses.

Interestingly, when we evaluate just the accuracy of the trained GNN model, its performance on the train, development and holdout sets slightly differs. The GNN has a balanced accuracy of 0.9503 on the training examples, 0.9397 on the examples from the development set that were used for selecting the best performing model, and 0.9406 on the examples from the holdout set. This difference is probably not large enough to cause significant differences in the ultimate ATP performance.

3.7.2 Transfer to Newly Added Mizar Articles

We have also tested our trained system on the problems from a newer version of Mizar (1382) that has 242 new articles and 13 370 theorems in them; more than half of which contain new terminology. Whereas iProver in the non-interactive mode solves 1662 theorems, iProver guided by our best model (dynamic trained on 10 samples with $d = 32$ and $l = 11$) solves 3657 theorems. Hence the improvement is similar to the results on the dataset that we used for our training, see Section 3.6.1.

3.8 Conclusion

We have developed efficient learning-based guidance for the Inst-Gen calculus and shown that the performance of the instantiation prover iProver is very substantially improved by doing the inferences recommended by a name-independent graph neural network. The number of theorems proved on our holdout set by the ML-guided iProver is more than doubled compared to the number of proofs the unguided solver can find. Additionally, we found that the model also generalizes very well to a large number of problems added to the Mizar Mathematical Library much later than our initial data set. This indicates that we can reasonably expect the trained predictors to generalize to new problems coming in over time.

The interactive interface of iProver that we have developed here can be used for other purposes. The interface exposes some of the internal data of iProver, which may be further relevant to the clause prediction task. We have already seen that the dynamic setting improves over the classic ENIGMA setting. In

the future, we could use the interface e.g. for giving the GNNs access to the SAT model which may serve as a semantic characterization of the search.

3.9 Acknowledgements

This work was partially supported by the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466 (KC, JP, JU), Amazon Research Awards (JP, JU), by the Czech MEYS under the ERC CZ project POSTMAN no. LL1902 (KC, JP), EPSRC grant EP/V000497/1 UK (KK), and the EU ICT-48 2020 project TAILOR no. 952215 (JU).

Chapter 4

Instantiation in SMT solvers with Graph Neural Networks*

Abstract

The cvc5 solver is today one of the strongest systems for solving first order problems with theories but also without them. In this work we equip its enumeration-based instantiation with a neural network that guides the choice of the quantified formulas and their instances. For that we develop a relatively fast graph neural network that repeatedly scores all available instantiation options with respect to the available formulas. The network runs directly on a CPU without the need for any special hardware. We train the neural guidance on a large set of proofs generated by the e-matching instantiation strategy and evaluate its performance on a set of previously unseen problems.

4.1 Introduction

In recent years, machine learning (ML) and neural methods have been increasingly used to guide the search procedures of automated theorem provers (ATPs).

*This chapter is based on the article "First experiments with Neural cvc5" published in the *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR) 2024* [93].

Such methods have been so far developed for choosing inferences in connection tableaux systems [65, 67, 99, 130, 135], resolution/superposition-based systems [45, 57, 58, 123], SAT solvers [116], tactical ITPs [5, 12, 13, 40, 41, 71, 105] and most recently also for the iProver [72] instantiation-based system [23]. In SMT (Satisfiability Modulo Theories), ML has so far been mainly used for tasks such as portfolio and strategy optimization [4,95,115]. Previous work [17,34,63] has explored fast non-neural ML guiding methods based on decision trees and manual features. Direct neural guidance of state-of-the-art SMT systems such as cvc5 [6] and Z3 [29] however has not been attempted yet.

One reason for that is the large number of choices typically available within the standard SMT procedures. Any ground term that already exists in the current context can be used to instantiate any free variable in the problem. While e.g. in the resolution/superposition systems, only the choice of the given clause can be guided and the rest of the work (its particular inferences with the set of processed clauses) is computed automatically by the ATPs,[2] in SMT, the trained ML system needs to make many more and finer decisions. This is both more fragile — due to the many interconnected low-level decisions instead of one high-level decision — and also slower. The ML (and especially neural) guidance is typically much more expensive than the standard guiding heuristics implemented in the systems, and the more low-level and exhaustive such guidance is, the larger the slowdown incurred by it becomes.

Despite that, there is a good motivation for experimenting with neural guidance for instantiation. Today's instantiation-based systems and SMT solvers such as cvc5, iProver and Z3 are becoming competitive on large sets of related problems coming e.g. from the *hammer* [16] links between ITPs and ATPs, and even for problems that do not contain explicit theories in the SMT sense [15, 30, 46]. The problems that they solve are often complementary to those solved by the state-of-the-art superposition-based systems such as E [112] and Vampire [102].

Contributions: In this work we develop efficient neural guidance for the enumerative instantiation in cvc5. We first give a brief overview of the instantiation procedures used by cvc5 (and generally SMTs) in Section 4.2. We then design a graph neural network (GNN) that is trained on cvc5's proofs and tightly integrated with cvc5's proof search. This yields a neurally guided version of cvc5 that runs reasonably fast without the need for specialized hardware, such as GPUs. Section 4.3 explains the GNN design, its training, collection of the training data from cvc5 and the neural instantiation procedure. Finally in Sec-

[2]This is similar also in the iProver instantiation-based system with its given-clause loop.

tion 4.4, we show that the GNN-guided enumeration mode outperforms cvc5's standard enumeration by 72% in real (CPU) time. This is measured on previously unseen problems extracted from the Mizar Mathematical library, after training the GNN on many proofs obtained on a large training set. We also investigate the behavior of cvc5's instantiation strategies, in terms of the number of instantiations performed in successful proof attempts. We show that e-matching can instantiate many more times than the enumeration strategies on our dataset. When we compensate for this, we arrive at an ML solver that improves on the enumeration mode by 109% in real time.

4.2 Proving By Instantiation

Quantifiers are essential in mathematics and reasoning. Practically, all today's systems used to formalize mathematics and for software verification are based on foundations such as first-order and higher-order logic, set theory and type theory, which make extensive use of quantification. Instantiation is a powerful method for formal reasoning with quantifiers. For example, the statement "All countries are completely in the northern hemisphere" is a quantified (false) statement, where "All countries" is a quantifier. This statement is readily shown false by instantiating with the country Brazil. The power of instantiation is formalized by *Herbrand's theorem* [50], which states that a set S of first-order clauses is unsatisfiable if and only if there is a finite set of ground instances of S that is unsatisfiable. In other words, quantifiers in false formulas can always be eliminated by a finite number of appropriate instantiations. Herbrand's theorem further states that it is sufficient to consider instantiations from the *Herbrand universe*, which consists of ground terms constructed from the symbols appearing in the problem. This fundamental result has been explored in automated reasoning (AR) systems since the 1950s [25, 27, 36, 110].

SMT solvers such as cvc5 and z3 handle quantifiers in a loop illustrated by Figure 4.1. The loop alternates between the *quantifier module*, which provides new instantiations (called lemmas), and the *SAT solver*, which decides whether the instantiations already lead to unsatisfiability. In the context of this work, we refer to one iteration of this high-level loop as a *round*.

There is extensive work on techniques that calculate new instantiations. Here we provide a concise explanation of three different quantifier instantiation methods implemented in cvc5. While e-matching is usually seen as the standard method (and is in fact part of the default procedure of cvc5), we start with the enumerative instantiation method in our explanation, as it is relatively simple

and allows us to introduce concepts more gradually.

Enumerative Instantiation: exhaustively instantiates with ground terms present in the current context [100]. In each round, it instantiates each *quantified expression*[3] with a tuple of ground terms—one term per variable. For a schematic overview, see Figure 4.2. The default strategy of the solver is to first

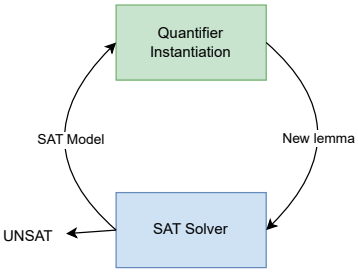

Figure 4.1: High-level architecture of cvc5. The techniques explained in Section 4.2, as well as our neural method (Section 4.3 and on), are particular cases of the top rectangle.

try the instantiations that use terms that were created earlier in the process (or were already in the input problem) — we refer to this as the *age heuristic*. For instance, for the ground part $\{p(c)\}$ and the quantifier $\forall x.\, q(f(x))$, the solver would instantiate by c in the first round and by $f(c)$ in the second round. There is also the option to restrict the set of terms using the *relevant domain*, but in our experiments we turn this off. Disabling this option simplifies the integration of the machine learning component. The relevant domain option is also turned off in our machine learning experiments. When we say enumeration mode, we mean the pure enumeration without *relevant domain*. The enumeration procedure only produces one instantiation per quantified expression per round.

E-matching: In e-matching instantiation [31, 84], the solver looks for instantiations (substitutions) that yield an existing ground term, modulo equality. Since there may be many such terms, it only considers terms fitting a certain pattern—called a *trigger*. Triggers may be provided by the user or generated by heuristics. As an example, consider the ground facts $\{a = f(17), p(a)\}$ and the quantifier formula $\forall x.\, \neg p(f(x)) \vee x < 0$. The trigger $p(f(x))$ would cause e-matching to instantiate x with 17 because the term $p(f(17))$ fits the trigger

[3]These are in general formulas, however in our experiments here we restrict ourselves to clausal problems.

and it is semantically equal to the existing term $p(a)$. This instantiation would yield the lemma $\neg p(f(17)) \vee 17 < 0$, which would give a contradiction with the ground facts. Preliminary investigations indicate that it is possible to use a similar machine learning setup as we used here to choose the triggers, but we leave it for future work here. Note that in contrast to the enumeration mode, e-matching may produce (potentially many) more than one instantiation per quantified expression per round. This difference in generative capacity becomes relevant in our experiments (Section 4.4).

Conflict-driven Instantiation: In conflict-driven quantifier instantiation [101], the solver looks for easily-detectable contradictions between the quantified part and the current model of the ground part; this reasoning is done modulo equality. As an example, consider a model M that contains the facts $\{f(a) \neq g(b), b = h(a)\}$ and there is a quantified formula $\forall x.\, f(x) \neq g(h(x))$. Then, the solver quickly identifies that instantiating x with a causes a contradiction with M and therefore, yielding the lemma $f(a) \neq g(h(a))$. Adding this lemma effectively excludes M from further search. The method only looks for instantiations that guarantee a conflict with the current model and aims to be fast and is therefore inherently incomplete. This technique is part of the default settings of cvc5 and we can turn this off using `--no-cbqi` to arrive at a solver that uses only e-matching to instantiate.

Term Creation & Proliferation: In any of these strategies, new ground terms are created by the instantiations that they propose. For example, when a subterm $f(f(X))$ is instantiated with a constant c, the new ground term $f(f(c))$ is created, as well as its subterm $f(c)$, if this subterm did not already exist as a ground term in the problem. Potentially, this can create a lot of new terms, which make the problem of choosing terms for instantiations harder.

4.3 Neural Instantiation for cvc5

Setting: We build our neural guidance on top of the enumerative instantiation. This is because (i) enumeration is the conceptually simplest of the instantiation strategies, (ii) it is general and complete, and (iii) it allows fine-grained control over the instantiations (which can however also have the downsides mentioned in Section 4.1). That said, our current neural guidance method is not necessarily complete — it may be very unfair. This is exactly the objective of training a machine learning heuristic: we want to learn from previous proofs how we can push the solver to be biased towards choices similar to the ones that were previously successful. While in principle, it is possible to create the training

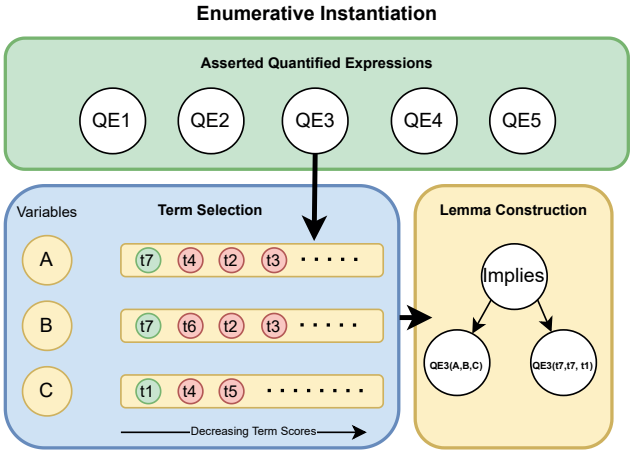

Figure 4.2: A schematic description of 1 instantiation within the enumerative instantiation procedure, which we heavily modify to create our neural solver. In the example a new ground lemma is created by instantiating the variables (A,B,C) in quantified expression 3 (QE3) with the ground terms (t7, t7, t1) respectively. In the default enumerative instantiation procedure, all quantified expressions are instantiated each round.

data for the neural network also from the enumerative instantiation mode, we chose to use the e-matching procedure for that. This is because e-matching is much stronger on our dataset of FOL problems (see Section 4.4), providing us with much more training data. This is similar to the experiments done with the E and ENIGMA systems in [44], where the training data collected from the "smart" E strategies are used to train guidance for a *tabula rasa* version of E.

GNN: While many different neural network methods can be used to guide automated theorem provers, a natural choice, based on the graph representation that cvc5 uses for the proof state, is the class of predictors called *graph neural networks* (GNNs) [106]. On a high level, GNNs represent each node in a graph with a vector of floating point numbers, and update these vectors using the vector representations of neighbouring nodes in the graph. By using optimization procedures, the GNN 'learns' to aggregate and update the node representations in such a way that at the end of several iterations of this neighbourhood-based updating procedure (usually called *message passing*), the node representations

contain useful information to predict some relevant quantity. In our setting, these relevant quantities are: (i) scores for each quantified expression that represent whether this expression should be instantiated and (ii) scores for each pair of variables and terms that represent whether this particular variable should be instantiated with a particular term.

We implement a custom GNN using the C++ frontend of PyTorch [90]. ML-guided ATPs often use a separate GPU server [23, 46], to which multiple prover processes send their requests for advice. Here, we are however interested in a tight integration within cvc5, allowing the ML component to use only one CPU thread. This also means that no time is lost communicating and encoding/decoding between different processes and different programming languages.

GNN Proof State Representation: Each cvc5 state is represented as a graph. Its nodes represent cvc5 expressions. They have a *kind*, such as *BOUND VARIABLE*, *APPLY*, etc. Each of the *kinds* that cvc5 recognizes internally is given a separately trainable embedding (vector in $\mathbb{R}^n$) that serves as the initial embedding of each node before the message passing phase (see below). The edges between nodes are collected by modifying the DAG that cvc5 uses to represent the state. The GNN uses a bidirectional (cyclic) version of the DAG. For example, if we have a term $f(c)$, we have not only an edge from f to c, but also an edge going the other way. We also use edge types to encode the argument ordering. For example, in $f(c, d)$ the edge from f to c has a different type than the edge from f to d. We recognize up to 5 different argument positions, with the fifth used to represent all remaining positions. Each edge type has a numerical vector associated with it that is used to make it possible to distinguish argument order during the message passing procedure.

GNN Message Passing: For the message passing part of the GNN, a concatenation of the *mean* and *max* aggregation of neighbourhood messages is used. We have also implemented and tested the simpler *mean* and the more complicated *attention-based* aggregation methods. However, the *mean-max* version had the best balance of complexity and performance.[4] Similarly, we tested the GNN with 2, 4, 10, 20 and 30 message passing layers, all with separate parameters. We found that 10 layers was a reasonable trade-off between the extra capability to fit the data and the execution speed within cvc5.

The different edge types are handled by adding a trainable vector e (which differs for each of the recognized edge types) to each source node in an edge,

[4]Here we consider both the complexity of the implementation and the computational complexity of the quadratic attention mechanism.

before doing the neighbourhood accumulations. This method avoids having separate weight matrices and thus message passing rounds for each edge type [107], which could complicate and possibly slow down the computations. In addition, these edge vectors can be seen as analogous to the *positional encodings* often used in transformer architectures. The embedding update rules (for embedding size K) are as follows for a single node j with N neighbours labeled by i:

$$s_t = x_t + e_{\text{type}_{ij}}$$
$$x_{t+1}^j = \text{RELU}(W \cdot \text{CONCAT}(\frac{1}{N}\sum_{i=0}^{N} s_t^i, \max(s_t))) + x_t^j,$$

where we compute a "source" vector s for each neighbour i depending on the type of the edge from i to j. The MAX function returns a vector that is the elementwise maximum over all the neighbourhood vectors. The matrix s_t has the shape $N \times K$ and x_{t+1}^j is a vector of size K. W is a linear transformation from dimensions $2K$ to K. CONCAT is a concatenation function that takes two vectors of size K and returns a vector of size $2K$. RELU is the Rectified Linear Unit function, max(0, x), computed elementwise. The above is performed for each node and its neighbourhood, in each message passing step. Each message passing step uses its own weight matrix W. At the end of each message passing step, we add to each node the associated x_t vector, which serves as a *residual connection*, a way to easily propagate information unchanged through the layers, if it is useful. In our experiments, the embedding size K was set to 64.

GNN Output: To decide what the solver should do, we use two different outputs of the GNN (see also Figure 4.2): (i) probability distributions for each of the top-level asserted quantified expressions, and (ii) for each variable a probability distribution over the terms of the right type, which we interpret as a probability that substituting this term in the variable leads to a useful instantiation. For output (i), note that we output a separate prediction for each quantified expression, which we can interpret as the probability that this quantified expression will be useful for the proof.

After the message passing steps, we have embeddings corresponding to each node in the graph. For the quantified expression selection task, we take all the nodes corresponding to the currently asserted top-level quantified expressions and apply a matrix transformation (a Linear layer) of size $K \times 1$ to each one, and use a sigmoid transformation to obtain a score between 0 and 1 for each one. Binary cross-entropy loss is used to train the network to optimize the scores according to the data.

For the term ranking task, we take the embedding representing variables, and then for each variable the embeddings representing terms of the correct type. We apply separate trainable linear transformations (of size K to K) to the term and variable embeddings and then compute dot products to obtain a distribution of term scores for each variable. We use a softmax transformation and cross-entropy loss to train the network to give high scores to variable-term substitutions occurring in the data.

Training Data Extraction: We have modified cvc5 to export the current proof state as a graph. For a particular problem, we extract for each solver round (Section 4.2 and Figure 4.1) the graph representation corresponding to the current proof state. To each such graph we also assign labels that indicate which quantified expressions and their instantiations were useful for the proof. In particular, our exporter labels instantiations as the correct answer as soon as the right ground terms become available. We also keep track of which instantiations were already done at a certain point in the run, so the GNN is not instructed to repeat instantiations. When there are multiple useful instantiations for the same quantified expression in a given proof state, we pick one at random. This is motivated by the *enumerative instantiation* mode's default behavior, where we only instantiate each expression once in every round.

Training Details: For a given set of training problems, we might have many transitions for a single problem and few for another one. To balance this out, in each *iteration* over the dataset, we randomly sample one of the transitions associated with the known proof for each training problem. The Adam optimizer implemented in PyTorch was used with default parameters, except for the learning rate, which was set to 0.0001.

4.4 Experiments

All our experiments were run on a machine with Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz CPU, 512GB RAM and NVIDIA Tesla V100 GPUs. The GPUs were used only for training the GNN. The trained neural models were always run on a single CPU when used for prediction inside cvc5. Our code and the trained GNN weights are available from our public repository.[5]

[5] `https://github.com/JellePiepenbrock/mlcvc5-LPAR`

4.4.1 Small Dataset

We first experimented with a small set of related problems extracted from the Mizar library. In particular, the MPTP2078 benchmark[6] [1] has been used for several earlier AI/TP experiments [65,67,75]. To make this smaller benchmark compatible with the larger benchmark we ultimately use (see below), we update the problems there to their version corresponding to the MML version 1147. This yields 2172 problems. These problems were split into three different sets, the *training* set (80%), the *devel*opment set (10%) and the *holdout* set (10%). The training set consists of problems where we are allowed to learn from solutions that we have, the development set consists of problems that we may tune the performance of our algorithm on and the holdout set is a set that is not used to tune parameters.

To collect training data for our approach, we ran our modified (Section 4.3) version of cvc5 in *only e-matching* mode.[7] The states, as well as the instantiations done were logged. These were converted into training data using the procedure described in Section 4.3. We end up with 814 solved problems, from which we extract 1934 training transitions. The model was then trained for 2000 iterations over the dataset.

In Table 4.1, we show the results of running various versions of cvc5 for 10s on the development and holdout sets.[8] The top 3 rows in the table correspond to a binary release of cvc5. The *enumeration* mode is observed to be a lot weaker on this dataset than *e-matching*. We also show a *dry run*, which is a run where we call the GNN to compute all the scores for quantified expressions (QEs) and term-variable combinations, but where we ignore those scores and simply use the default *enumeration* strategy's suggestions. This allows us to gauge the slowdown caused by calling the GNN and its message passing and scoring procedures. While the 10s timeout can be seen as relatively short compared to the multi-minute timeouts used in competitions like SMTCOMP, it is indicative of the performance in a hammer-type setting.

We can use the scores (which are between 0 and 1 for each QE) predicted by the GNN in different ways. Here, we experimented with two procedures for the quantified expression scores: (i) *QSampling*, where we take the scores associated with each QE and interpret this as the probability of using this QE in this round. A sampling procedure decides, by drawing a random number between 0 and 1 and comparing it with the score given, whether to instantiate

[6] `https://github.com/JUrban/MPTP2078`

[7] This means that we use the `--no-cbqi` and `--no-cegqi` parameters.

[8] In the evaluations, we always use 15 parallel processes, however each problem always uses a single CPU.

this QE in this round. If the random number is higher than the score, we do not use the QE in this round. (ii) Threshold, where we compare the score to a threshold number and only instantiate the QEs with scores above the threshold. In our tuning phase (done on the development set), we found that a very low threshold[9] value (0.00001) was the best setting for the Threshold variant. In general, false negatives (preventing a QE from being instantiated) seem to be a bigger problem for the solver than false positives. As in premise selection, having some extra QEs instantiated does not seem to be as problematic as omitting some necessary ones.

Table 4.1: Number of problems solved by 10s runs on the devel and holdout parts of the small dataset. Bcvc5 is an unchanged binary release of cvc5. MLcvc5 is our modified version. Some of the changes cause a slowdown.

	Development	Holdout
Bcvc5 — default strategies	148	134
Bcvc5 — only e-matching	129	115
Bcvc5 — only enumeration	48	49
MLcvc5 — dry run	33	22
MLcvc5 — model (QSampling)	49	29
MLcvc5 — model (Threshold — 1 × 10-5)	54	35

Comparing the 'Bcvc5 — only enumeration' and 'MLcvc5 — dry run' in Table 4.1 we see that the performance hit caused by calling the GNN is quite significant. However, we see that both on the development and holdout sets we do improve on the performance of the *dry run*. This means the predictions of the GNN are useful. Unfortunately, we are not able to improve on the binary version of *enumeration* on the holdout set. On the development set, we can improve on it by a few problems. The split between training, development and holdout was done randomly to prevent a bias in the different sets. However, several of the methods seem to have worse performance on the holdout set here. A larger selection of problems could help alleviate this. While these results indicate that the GNN can be useful for the guidance of cvc5 in the *enumeration* mode, this training dataset might be too small to learn sufficiently strong GNN guidance.

[9]We tested the following thresholds: 0.5, 0.4, 0.3, 0.2, 0.1, 0.01, 0.001, 0.0001 and 0.00001.

4.4.2 Large Dataset (MPTP1147)

In the final evaluation we use the full MPTP 1147 benchmark, used previously in the Mizar40 [66] and Mizar60 [60] experiments. This dataset is more than 25x as large as the MPTP2078-induced subset (57917 problems in total). We use the train/devel/holdout split as defined in the previous work [23, 60]. Running on the training problems with the data collection mode of cvc5 with only e-matching active gives us 10945 solved problems, which we use to generate the training data. The model was trained for 150 iterations on this training dataset. After 150 epochs, the model has 82.9% accuracy on predicting the right terms for each variable on the training problems. For the quantified expression task, which scores the QEs between 0 and 1, 70.7% of useful quantified expressions are given a score above 0.5, while 88.3% of the non-useful quantified expressions are given a score below 0.5. While the model did not perfectly fit the training data, it has some capacity to learn the data. In Table 4.2, we show the development

Table 4.2: MML1147: Number of problems solved by 10s runs. On both the development and the holdout sets the GNN-guided enumeration mode outperforms the unguided enumeration mode. Both the development and holdout sets contain 2896 problems.

	Development	Holdout
Bcvc5 — only e-matching	1096	1107
Bcvc5 — only enumeration	183	195
MLcvc5 — dry run	119	120
MLcvc5 — model (QSampling)	288	300
MLcvc5 — model (Threshold - 1x10-5, 1 inst)	324	336
MLcvc5 — model (Threshold - 1x10-5, 10 inst)	410	407

and holdout set performance of the ML-guided cvc5, along with various control experiments. We observe that the performance of the enumeration mode was improved by up to 72% ($336/195 = 1.723$) when we use the Threshold (1 inst) variant of our network. As a sanity check, we also ran a *randomized dry run* experiment on the development set, where we computed all the model scores, but used a randomly shuffled term ordering (instead of the age-based ordering for the usual dry run). This mode only solved 10 problems from the development set. These results taken together indicate that the network has learned a useful strategy from the e-matching generated training data, which it can apply in the ML enumeration mode.

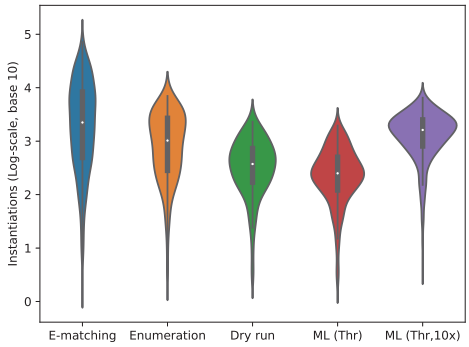

Figure 4.3: Violin plots of the number of instantiations performed in successful runs for Bcvc5 e-matching, Bcvc5 enumeration, dry run, the ML strategy with threshold 1e-5, and the ML strategy with 10x as many instantiations per quantifier per round. The white dots indicate the medians. The respective medians are 2235, 1026, 373.5, 250 and 1620.

Comparison of number of instantiations: While the runtime of all solvers was limited to 10s on 1 CPU, the various versions and settings of cvc5 can vary in terms of the absolute number of instantiations done within the same real time. The enumeration mode strives to perform at least 1 instantiation for each QE per *round* (Figures 4.1 and 4.2), and will not generally instantiate more than once for each QE in each round. E-matching, however, is not bound by this and will instantiate based on the number of pattern matches (which can be high). In Figure 4.3, we show the number of instantiations done in successful runs for five strategies. The median number for e-matching is an order of magnitude higher than in the ML strategy. E-matching is much more prolific than enumeration, and the ML strategy is less prolific in 10s than enumeration due to a combination of QEs that are not used and the GNN slowdown. The number of instantiations is of course also mediated by the time spent in computing the neural network's predictions. This time varies heavily per problem and potentially per run, depending on the size of the initial problem and how much the graph grows each round (for example due to a lot of new lemmas and terms). In the successful runs on the MPTP1147 development set, there are neural network predictions that take below 40ms and ones that take more than 2400ms.

GNN run with multiple instantiations: The above analysis indicates that some difference in performance is due to the difference in the raw number of instantiations done. As we are already incurring the cost of computing the GNN advice, it might be the case that instantiation with multiple high-scoring tuples per round, instead of only 1 per QE as the original enumeration does, is a better use of the GNN advice. To test this, we ran a version of the ML mode that performs up to 10 instantiations per QE per round (see Table 4.2). This led to 407 solved holdout problems (again in 10s real time). This is a 109% increase compared to the unmodified enumeration mode ($407/195 = 2.09$).

Overlaps of sets: In Table 4.3, we show the set differences between the sets of solved problems for e-matching, enumeration and the best-performing ML setting on the holdout set. We observe that we can solve many problems that were not solved by the unguided enumeration mode, but that the e-matching mode is stronger than our method on this dataset.

Table 4.3: Set differences in terms of number of solved problems on holdout set, row minus column. Example: the ML solves 246 problems that the enumeration mode does not. ML means the predictor with (Threshold 1x10-5, 10 inst).

	Bcvc5 e-mat	Bcvc5 - enum	ML
Bcvc5 — e-matching	0	922	717
Bcvc5 — enum	10	0	34
ML	17	246	0

4.5 Conclusion

In this work, we have shown that it is possible to improve cvc5's enumerative instantiation by using an efficient graph neural network trained on many related problems. Our best result is 109% improvement in (realistically low) real time and with exactly the same hardware resources (i.e., a single CPU). This is done here so far in a first-order clausal setting without theories, however extensions to non-clausal SMT with theories should be mostly straightforward.

While e-matching largely dominates on first-order logic problems extracted from the Mizar Mathematical Library, on problems from the SMTLIB database, the enumeration procedure is much more competitive [62], and can even outperform e-matching on certain types of benchmarks. In principle, we can extend the current method to SMT problems, aside from the fact that the logging pro-

cedure that extracts training data from cvc5 runs needs to be modified. This could lead to some difficulties with tracing of instantiations, as it is not always clear how a term came to be (as the mechanism may be 'hidden' inside a theory component).

Future work will also investigate whether a better balance between the computation of the advice itself and the number of instantiations done based on this advice can be found. We may be under-utilizing the expensive advice of the GNN. Another direction of investigation will be the optimization of the speed of the neural network: it may be possible to use a much smaller neural network and still get reasonable predictions, but much faster. We will also investigate the generalization performance of the method. For example, testing the performance on problems extracted from other systems, such as Isabelle or Coq could be insightful. In general, our work shows for the first time that efficient real-time neural guidance for SMT solvers is possible.

4.6 Acknowledgements

This work was partially supported by the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15 003/0000466 (JP, JU), the European Union under the project ROBOPROX (reg. no. CZ.02.01.01/00/22_008/0004590) (MJ), Amazon Research Awards (JP, JU), by the Czech MEYS under the ERC CZ project POSTMAN no. LL1902 (JP, MJ, JU, JJ), ERC PoC grant no. 101156734 *FormalWeb3* (JJ), and the EU ICT-48 2020 project TAILOR no. 952215 (JU).

Chapter 5

Guiding an Automated Theorem Prover with Neural Rewriting*

Abstract

Automated theorem provers (ATPs) are today used to attack open problems in several areas of mathematics. An ongoing project by Kinyon and Veroff uses Prover9 to search for the proof of the Abelian Inner Mapping (AIM) Conjecture, one of the top open conjectures in quasigroup theory. In this work, we improve Prover9 on a benchmark of AIM problems by neural synthesis of useful alternative formulations of the goal. In particular, we design the 3SIL (stratified shortest solution imitation learning) method. 3SIL trains a neural predictor through a reinforcement learning (RL) loop to propose correct rewrites of the conjecture that guide the search.

3SIL is first developed on a simpler, Robinson arithmetic rewriting task for which the reward structure is similar to theorem proving. There we show that 3SIL outperforms other RL methods. Next we train 3SIL on the AIM benchmark and show that the final trained network, deciding what actions to take within the equational rewrit-

*This chapter is based on the article of the same name published in the *International Joint Conference on Automated Reasoning (IJCAR), 2022* [92].

ing environment, proves 70.2% of problems, outperforming Waldmeister (65.5%). When we combine the rewrites suggested by the network with Prover9, we prove 8.3% more theorems than Prover9 in the same time, bringing the performance of the combined system to 90%.

5.1 Introduction

Machine learning (ML) has recently proven its worth in a number of fields, ranging from computer vision [49], to speech recognition [47], to playing games [83, 119] with *reinforcement learning* (RL) [125]. It is also increasingly applied in automated and interactive theorem proving. Learned predictors have been used for premise selection [1] in hammers [16], to improve clause selection in saturation-based theorem provers [22], to synthesize functions in higher-order logic [38], and to guide connection-tableau provers [67] and interactive theorem provers [5, 14, 41].

Future growth of the knowledge base of mathematics and the complexity of mathematical proofs will increase the need for proof checking and better computer support and automation. Simultaneously, the growing complexity of software will increase the need for formal verification to prevent failure modes [29]. Automated theorem proving and mathematics will benefit from more advanced ML integration. One of the mathematical subfields that makes substantial use of automated theorem provers is the field of quasigroup and loop theory [91].

5.1.1 Contributions

In this work, we propose to use a neural network to suggest lemmas to the Prover9 [80] ATP system by rewriting parts of the conjecture (Section 5.2). We test our method on a dataset of theorems collected in the work on the Abelian Inner Mapping (AIM) Conjecture [70] in loop theory. For this, we use the AIMLEAP proof system [19] as a reinforcement learning environment. This setup is described in Section 5.3. For development we used a simpler Robinson arithmetic rewriting task (Section 5.4). With the insights derived from this and a comparison with other methods, we describe our own 3SIL method in Section 5.5. We use a neural network to process the state of the proving attempt, for which the architecture is described in Section 5.6. The results on the Robinson arithmetic task are described in Section 5.7.1. We show our results on the

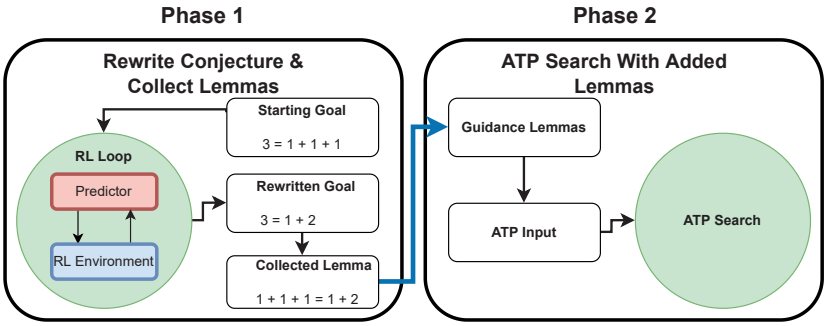

Figure 5.1: Schematic representation of the proposed guidance method. In the first phase, we run a reinforcement learning loop to propose actions that rewrite a conjecture. This predictor is trained using the AIMLEAP proof environment. We collect the rewrites of the LHS and RHS of the conjecture. In the second phase, we add the rewrites to the ATP search input, to act as guidance. In this specific example, we only rewrote the conjecture for 1 step, but the added guidance lemmas are in reality the product of many steps in the RL loop.

AIMLEAP proving task, both using our predictor as a stand-alone prover and by suggesting lemmas to Prover9 in Section 5.7.2. Our contributions are:

1. We propose a training method for reinforcement learning in theorem proving settings: *stratified shortest solution imitation learning* (3SIL). This method is suited to the structure of theorem proving tasks. This method and the reasoning behind it are explained in Section 5.5.

2. We show that 3SIL outperforms other baseline RL methods on a simpler, Robinson arithmetic rewriting task for which the reward structure is similar to theorem proving (Section 5.7.1).

3. We show that a standalone neurally guided prover trained by the 3SIL method outperforms the hand-engineered Waldmeister prover on the AIMLEAP benchmark (Section 5.7.2).

4. We show that using a neural rewriting step that suggests rephrased versions of the conjecture to be added as lemmas improves the ATP performance on equational problems (Sections 5.2 and 5.7.2).

5.2 ATP and Suggestion of Lemmas by Neural Rewriting

Saturation-based ATPs make use of the *given clause* [89] algorithm, which we briefly explain as background. A problem is expressed as a conjunction of many initial clauses (i.e., the clausified axioms and the negated goal which is always an equation in the AIM dataset). The algorithm starts with all the initial clauses in the *unprocessed set*. We then pick a clause from this set to be the given clause and move it to the *processed set* and do all inferences with the clauses in the processed set. The newly inferred clauses are added to the unprocessed set. This concludes one iteration of the algorithm, after which we pick a new given clause and repeat [69]. Typically, this approach is designed to be *refutationally complete*, i.e., the algorithm is guaranteed to eventually find a contradiction if the original goal follows from the axioms.

This process can produce a lot of new clauses and the search space can become quite large. In this work, we modify the standard loop by adding useful lemmas to the initial clause set. These lemmas are proposed by a neural network that was trained *from zero knowledge* to rewrite the left- and right-hand sides of the initial goal to make them equal by using the axioms as the available rewrite actions. Even though the neural rewriting might not fully succeed, the rewrites produced by this process are likely to be useful as additional lemmas when added to the problem. This idea is schematically represented in Figure 5.1.

5.3 AIM Conjecture and the AIMLEAP RL Environment

Automated theorem proving has been applied in the theory surrounding the Abelian Inner Mapping Conjecture, known as the AIM Conjecture. This is one of the top open conjectures in quasigroup theory. Work on the conjecture has been going on for more than a decade. Automated theorem provers use hundreds of thousands of inference steps when run on problems from this theory.

As a testbed for our machine learning and prover guidance methods we use a previously published dataset of problems generated by the AIM conjecture [19]. The dataset comes with a simple prover called AIMLEAP that can take machine learning advice.[2] We use this system as an RL environment. AIMLEAP

[2] `https://github.com/ai4reason/aimleap`

keeps the state and carries out the cursor movements (the cursor determines the location of the rewrite) and rewrites that a neural predictor chooses.

The AIM conjecture concerns specific structures in *loop theory* [70]. A loop is a quasigroup with an identity element. A quasigroup is a generalization of a group that does not preserve associativity. This manifests in the presence of two different 'division' operators, one left-division (\) and one right-division (/). We briefly explain the conjecture to show the nature of the data.

For loops, three *inner mapping functions* (left-translation L, right-translation R, and the mapping T) are:

$$L(u,x,y) := (y * x)\backslash(y * (x * u)) \qquad T(u,x) := x\backslash(u * x)$$
$$R(u,x,y) := ((u * x) * y)/(x * y)$$

These mappings can be seen as measures of the deviation from commutativity and associativity. The conjecture concerns the consequences of these three inner mapping functions forming an Abelian (commutative) group. There are two more notions, the *associator* function a and the *commutator* function K:

$$a(x,y,z) := (x * (y * z))\backslash((x * y) * z) \qquad K(x,y) := (y * x)/(x * y)$$

From these definitions, the conjecture can be stated. There are two parts to the conjecture. For both parts, the following equalities need to hold for all u, v, x, y, and z:

$$a(a(x,y,z),u,v) = 1 \qquad a(x,a(y,z,u),v) = 1 \qquad a(x,y,a(z,u,v)) = 1$$

where 1 is the identity element. These are necessary, but not sufficient for the two main parts of the conjecture. The first part of the conjecture asks whether a loop modulo its center is a group. In this context, the *center* is the set of all elements that commute with all other elements. This is the case if

$$K(a(x,y,z),u) = 1.$$

The second part of the conjecture asks whether a loop modulo its nucleus is an Abelian group. The *nucleus* is the set of elements that associate with all other elements. This is the case if:

$$a(K(x,y),z,u) = 1 \qquad a(x,K(y,z),u) = 1 \qquad a(x,y,K(z,u)) = 1.$$

5.3.1 The AIMLEAP RL Environment

Currently, work in this area is done using automated theorem provers such as *Prover9* [70, 80]. This has led to some promising results, but the search space is enormous. The main strategy for proving the AIM conjecture thus far has been to prove weaker versions of the conjecture (using additional assumptions) and then import crucial proof steps into the stronger version of the proof. The *Prover9* theorem prover is especially suited to this approach because of its well-established *hints* mechanism [131]. The AIMLEAP dataset is derived from this *Prover9* approach and contains around 3468 theorems that can be proven with the supplied definitions and lemmas [19].

There are 177 possible actions in the AIMLEAP environment [19]. We handle the proof state as a tree, with the root node being an equality node. Three actions are cursor movements, where the cursor can be moved to an argument of the current position. The other actions all rewrite the current term at the cursor position with various axioms, definitions and lemmas that hold in the AIM context. As an example, this is one of the theorems in the dataset ($\backslash$ and $=$ are part of the language):

$$T(T(T(x, T(x, y)\backslash 1), T(x, y)\backslash 1), y) = T((T(x, y)\backslash 1)\backslash 1, T(x, y)\backslash 1) \,.$$

The task of the machine learning predictor is to process the proof state and recognize which actions are most likely to lead to a proof, meaning that the two sides of the starting equation are equal according to the AIMLEAP system. The only feedback that the environment gives is whether a proof has been found or not: there is no intermediate reward (i.e. rewards are *sparse*). The ramifications of this are further discussed in Section 5.5.1.

5.4 Rewriting in Robinson Arithmetic as an RL Task

To develop a machine learning method that can help solve equational theorem proving problems, we considered a simpler arithmetic task, which also has a tree-structured input and *a sparse reward structure*: the normalization of Robinson arithmetic expressions. The task is to normalize a mathematical expression to one specific form. This task has been implemented as a Python RL environment, which we make available.[3] The learning environment incorporates an existing dataset, constructed by Gauthier for RL experiments in the interactive theorem prover HOL4 [37]. Our RL setup for the task is also modeled after [37].

[3] `https://github.com/JellePiepenbrock/neural_rewriting`

In more detail, the formalism that we use as an RL environment is Robinson arithmetic (RA). RA is a simple arithmetic theory. Its language contains the successor function S, addition $+$ and multiplication $*$ and one constant, the 0. The theory considers only non-negative numbers and we only use four axioms of RA. Numbers are represented by the constant 0 with the appropriate number of successor functions applied to it. The task for the agent is to rewrite an expression until there are only nodes of the successor or 0 types. Effectively, we are asking the agent to calculate the value of the expression. As an example, $S(S(0)) + S(0)$, representing $2 + 1$, needs to be rewritten to $S(S(S(0)))$.

The expressions are represented as a tree data structure. Within the environment, there are seven different rewrite actions available to the agent. The four axioms (equations) defining these actions are $x{+}0 = x$, $x{+}S(y) = S(x{+}y)$, $x * 0 = 0$ and $x * S(y) = (x * y) + x$, where the agent can apply the equations in either direction. There is one exception: the multiplication by 0 cannot be applied from right to left, as this would require the agent to introduce a fresh term which is out of scope for the current work. The place where the rewrite is applied is denoted by the location of the *cursor* in the expression tree.

In addition to the seven rewrite actions, the agent can move the cursor to one of the children of the current cursor node. This gives a total number of nine actions. Moving to a child of a node with only one child counts as moving to the left child. After a rewriting action, the cursor is reset to the root of the expression. More details on the actions are in the RewriteRL repository.

5.5 Reinforcement Learning Methods

This section describes the reinforcement learning methods, while Section 5.6 then further explains the particular neural architectures that are trained in the RL loops. We first briefly explain here the approaches that we used as reinforcement learning (RL) baselines, and then we go into detail about the proposed 3SIL method.

5.5.1 Reinforcement Learning Baselines

General RL setup

For comparison, we used implementations of four established reinforcement learning baseline methods. In reinforcement learning, we consider an *agent* that is acting within an *environment*. The agent can take actions a from the action-space $\mathcal{A}$ to change the state $s \in \mathcal{S}$ of the environment. The agent can be

rewarded for certain actions taken in a certain state, with the reward given by the *reward function* $\mathcal{R} : (\mathcal{S} \times \mathcal{A}) \to \mathbb{R}$. The behavior of the environment is given by the *state transition function* $\mathcal{P} : (\mathcal{S} \times \mathcal{A}) \to \mathcal{S}$. The history of the agent's actions and the environment states and rewards at each timestep t are collected in tuples (s_t, a_t, r_t). For a given history of a certain agent within an environment, we call the list of tuples (s_t, a_t, r_t) describing this history an *episode*. The *policy function* $\pi : \mathcal{S} \to \mathcal{A}$ allows the agent to decide which action to take. The agent's goal is to maximize the return R: the sum of discounted rewards $\sum_{t \geq 0} \gamma^t r_t$, where γ is a *discount factor* that allows control over how heavily rewards further in the future should be weighted. We will use R_t when we mean R, but calculated only from rewards from timestep t on. In the end, we are thus looking for a policy function π that maximizes the sum R of (discounted) expected rewards [125].

In our setting, every proof attempt (in the AIM setting) or normalization attempt (in the Robinson arithmetic setting) corresponds to an episode. The reward structure of theorem proving is such that there is only a reward of 1 at the end of a successful episode (i.e. a proof was found in AIM). Unsuccessful episodes get a reward of 0 at every timestep t.

A2C

The first method, *Advantage Actor-Critic*, or *A2C* [82] contains ideas on which the other three RL baseline methods build, so we will go into more detail about this method, while keeping the explanation for the other methods brief. For details we refer to the corresponding papers.

A2C attempts to find suitable parameters for an agent by minimizing a *loss function* consisting of two parts:

$$\mathcal{L} = \mathcal{L}^{\text{A2C}}_{\text{policy}} + \mathcal{L}^{\text{A2C}}_{\text{value}} .$$

In addition to the policy function π, the agent has access to a *value function* $\mathcal{V} : \mathcal{S} \to \mathbb{R}$, that predicts the sum of future rewards obtained when given a state. In practice, both the policy and the value function are computed by a neural network *predictor*. The parameters of the predictor are set by *stochastic gradient descent* to minimize $\mathcal{L}$. The set of parameters of the predictor that defines the policy function π is named θ, while the parameters that define the value function are named μ. The first part of the loss is the *policy loss*, which for one time step has the form

$$\mathcal{L}^{\text{A2C}}_{\text{policy}} = -\log \pi_\theta(a_t|s_t) A(s_t, a_t) ,$$

where $A(s, a)$ is the *advantage function*. The advantage function can be formulated in multiple ways, but the simplest is as $R_t - \mathcal{V}_\mu(s_t)$. That is to say: the advantage of an action in a certain state is the difference between the discounted rewards R_t after taking that action and the value estimate of the current state.

Minimizing $\mathcal{L}^{\text{A2C}}_{\text{policy}}$ amounts to maximizing the log probability of predicting actions that are judged by the advantage function to lead to high reward.

The value estimates $V_\mu(s)$ for computing the advantage function are supplied by the *value predictor* V_μ with parameters μ, which is trained using the loss:

$$\mathcal{L}^{\text{A2C}}_{\text{value}} = \frac{1}{2}\left(R_t - \mathcal{V}_\mu(s_t)\right)^2 ,$$

which minimizes the advantage function. The logic of this is that the value estimate at timestep t, $\mathcal{V}_\mu(s_t)$, will learn to incorporate the later rewards R_t, ensuring that when later seeing the same state, the possible future reward will be considered. Note that the sets of parameters θ and μ are not necessarily disjoint (see Section 5.6).

Note how the above equations are affected if there is no non-zero reward r_t obtained at any timestep. In that case, the value function $\mathcal{V}_\mu(s_t)$ will estimate (correctly) that any state will get 0 reward, which means that the advantage function $A(s, a)$ will also be 0 everywhere. This means that $\mathcal{L}^{\text{A2C}}_{\text{policy}}$ will be 0 in most cases, which will lead to no or little change in the parameters of the predictor: learning will be very slow. This is the difficult aspect of the structure of theorem proving: there is only reward at the end of a successful proof, and nowhere else. This implies a possible strategy is to imitate successful episodes, without a value function. In this case, we would only need to train a *policy function*, and no approximate *value function*. This is an aspect we explore in the design of our own method 3SIL, which we will explain shortly.

Compared to two-player games, such as chess and go, for which many approaches have been tailored and successfully used [120], theorem-proving has the property that it is hard to collect useful examples to learn from, as only successful proofs are likely to contain useful knowledge. In chess or go, however, one player almost always wins and the other loses, which means that we can at least learn from the difference between the two strategies used by those players. As an example, we executed 2 million random proof attempts on the AIMLEAP environment, which led to 300 proofs to learn from, whereas in a two-player setting like chess, we would get 2 million games in which one player would likely win.

ACER The second RL baseline method we tested in our experiments is ACER, *Actor-Critic with Experience Replay* [133]. This approach can make

use of data from older episodes to train the current predictor. ACER applies corrections to the value estimates so that data from old episodes may be used to train the current policy. It also uses trust region policy optimization [108] to limit the size of the policy updates. This method is included as a baseline to check if using a larger replay buffer to update the parameters would be advantageous.

PPO Our third RL baseline is the widely used *proximal policy optimization* (PPO) algorithm [109]. It restricts the size of the parameter update to avoid causing a large difference between the original predictor's behavior and the updated version's behavior. The method is related to the above trust region policy optimization method. In this way, PPO addresses the training instability of many reinforcement learning approaches. It has been used in various settings, for example complex video games [9]. With its versatility, the PPO algorithm is well-positioned. We use the PPO algorithm with clipped objective, as in [109].

SIL-PAAC Our final RL baseline uses only the transitions with positive advantage to train on for a portion of the training procedure, to learn more from good episodes. This was proposed as *self-imitation learning* (SIL) [87]. To avoid confusion with the method that we are proposing, we extend the acronym to SIL-PAAC, for positive advantage actor-critic. This algorithm outperformed A2C on the sparse-reward task Montezuma's Revenge (a puzzle game). As theorem proving has a sparse reward structure, we included SIL-PAAC as a baseline. More information about the implementations for the baselines can be found in the Implementation Details section at the end of this work.

5.5.2 Stratified Shortest Solution Imitation Learning

We introduce stratified shortest solution imitation learning (**3SIL**) to tackle the equational theorem proving domain. It learns to explicitly imitate the actions taken during the shortest solutions found for each problem in the dataset. We do this by minimizing the cross-entropy $-\log p(a_{solution}|s_t)$ between the predictor output and the actions taken in the shortest solution. This is in contrast to the baseline methods, where value functions are used to judge the utility of decisions.

In our procedure this is not the case. Instead, we build upon the assumption for data selection that shorter proofs are better in the context of theorem proving and expression normalization. In a sense, we value decisions from shorter proofs more and explicitly imitate those transitions. We keep a history H for each problem, where we store the current shortest solution (states seen and actions taken) found for that problem in the training dataset. We can also store

Algorithm 1 CollectEpisode

Input: problem p, policy π_θ, problem history H
Generate episode by following noisy version of π_θ on p
If solution, add list of tuples (s, a) to H[p]
Keep k shortest solutions in H[p]

Algorithm 2 3SIL

Input: set of problems P, randomly initialized policy π_θ, batch size B, number of batches NB, problem history H, number of warmup episodes m, number of episodes f, max epochs ME
Output: trained policy π_θ, problem history H
for $e = 0$ **to** ME $- 1$ **do**
  if $e = 0$ **then** num $= m$ **else** num $= f$

  for $i = 0$ **to** num $- 1$ **do**
    CollectEpisode(sample(P), π_θ, H) (Algorithm 1)
  end for
  for $i = 0$ **to** NB $- 1$ **do**
    Sample B tuples (s, a) with uniform probability for each problem in H
    Update θ to lower $-\sum_{b=0}^{B} \log \pi_\theta(a_b|s_b)$ by gradient descent
  end for
end for

multiple shortest solutions for each problem if there are multiple strategies for a proof (the number of solutions kept is governed by the parameter k).

During training, in the case $k = 1$, we sample state-action pairs from each problem's current shortest solution at an equal probability (if a solution was found). To be precise, we first randomly pick a theorem for which we have a solution, and then randomly sample one transition from the shortest encountered solution. This directly counters one of the phenomena that we had observed: the training examples for the baseline methods tend to be dominated by very long episodes (as they contribute more states and actions). This *stratified* sampling method ensures that problems with short proofs get represented equally in the training process.

The 3SIL algorithm is described in more detail in Algorithm 2. Sampling from a noisy version of policy π_θ means that actions are sampled from the predictor-defined distribution and in 5% of cases a random valid action is se-

lected. This is also known as the ϵ-greedy policy (with ϵ at 0.05).

Related Methods Our approach is similar to the imitation learning algorithm DAGGER (Dataset Aggregation), which was used for several games [104] and modified for branch-and-bound algorithms in [48]. The behavioral cloning (BC) technique used in robotics [126] also shares some elements. 3SIL significantly differs from DAGGER and BC because it does not use an outside expert to obtain useful data, because of the stratified sampling procedure, and because of the selection of the shortest solutions for each problem in the training dataset. We include as an additional baseline an implementation of behavioral cloning (BC), where we regard proofs already encountered as coming from an expert. We minimize cross-entropy between the actions in proofs we have found and the predictions to train the predictor. For BC, there is no stratified sampling or shortest solution selection, only the minimization of cross-entropy between actions taken from recent successful solutions and the predictor's output.

Extensions For the AIM tasks, we introduce two other techniques, *biased sampling* and *episode pruning*. In biased sampling, problems without a solution in the history are sampled 5 times more during episode collection than solved problems to accelerate progress. This was determined by testing 1, 2, 5 and 10 as sampling proportions. For episode pruning, when the agent encountered the same state twice, we prune the episode to exclude the looping before storing the episode. This helps the predictor learn to avoid these loops.

5.6 Neural Architectures

The tree-structured states representing expressions occurring during the tasks will be processed by a neural network. The neural network takes the tree-structured state and predicts an action to take that will bring the expression closer to being normalized or the theorem closer to being proven.

There are two main components to the neural network we use: an *embedding* tree neural network that outputs a numerical vector representing the tree-structured proof state and a second *processor* network that takes this vector representation of the state and outputs a distribution of the actions possible in the environment.[4]

Tree neural networks have been used in various settings, such as natural language processing [56] and also in Robinson arithmetic expression embedding [39]. These networks consist of smaller neural networks, each representing

[4]In the reinforcement learning baselines that we use, this second *processor* network has the additional task of predicting the value of a state.

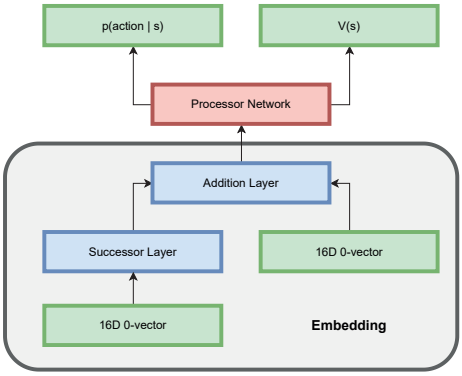

Figure 5.2: Schematic representation of the creation of a representation of an expression (*an embedding*) using different neural network layers to represent different operations. The figure depicts the creation of a numerical representation for the Robinson arithmetic expression $(S(0)+0)$. Note that the successor layer and the addition layer consist of trainable parameters, for which the values are set through gradient descent.

one of the possible functions that occur in the expressions. For example, there will be separate networks representing addition and multiplication. The cursor is a special unary operation node with its own network that we insert into the tree at the current location. For each unique constant, such as the constant 0 in RA or the identity element 1 for the AIM task, we generate a random vector (from a standard normal distribution) that will represent this leaf. In the case of the AIM task, these vectors are parameters that can be optimized during training.

At prediction time, the numerical representation of a tree is constructed by starting at the leaves of the tree, for which we can look up the generated vectors. These vectors act as input to the neural networks that represent the parent node's operation, yielding a new vector, which now represents the subtree of the parent node. The process repeats until there is a single vector for the entire tree after the root node is processed (see also Figure 5.2).

The neural networks representing each operation consist of a linear transformation, a non-linearity in the form of a rectified linear unit (ReLU) and another linear transformation. In the case of binary operations, the first linear transformation will have an input dimension of $2n$ and an output dimension of n, where n is the dimension of the vectors representing leaves of the tree (the

internal representation size). The weights representing these transformations are randomly initialized at the beginning of training.

When we have obtained a single vector embedding representing the entire tree data structure, this vector serves as the input to the *predictor* neural network, which consists of three linear layers, with non-linearities (Sigmoid/ReLU) in between these layers. The last layer has an output dimension equal to the number of possible actions in the environment. We obtain a probability distribution over the actions, e.g. by applying the softmax function to the output of this last layer. In the cases where we also need a value prediction, there is a parallel last layer that predicts the state's value (usually referred to as a *two-headed* network [120]). The internal representation size n for the Robinson arithmetic experiments is set to 16, for the AIM task this is 32. The number of neurons in each layer (except for the last one) of the predictor networks is 64.

In the AIM dataset task, an arbitrary number of variables can be introduced during the proof. These are represented by untrainable random vectors. We add a special neural network (with the same architecture as the networks representing unary operations, so from size n to n) that processes these vectors before they are processed by the rest of the tree neural network embedding. The idea is that this neural network learns to project these new variable vectors into a subspace and that an arbitrary number of variables can be handled. The vectors are resampled at the start of each episode, so the agent cannot learn to recognize specific variables. This approach was partly inspired by the *prime* mechanism in [39], but we use separate vectors for all variables instead of building vectors sequentially. All our neural networks are implemented using the PyTorch library [90].

5.7 Experiments

We first describe our experiments on the Robinson arithmetic task, with which we designed the properties of our 3SIL approach with the help of comparisons with other algorithms. We then train a predictor using 3SIL on the AIMLEAP loop theory dataset, which we evaluate both as a standalone prover within the RL environment and as a neural guidance mechanism for the ATP Prover9.

5.7.1 Robinson Arithmetic Dataset

Dataset details The Robinson arithmetic dataset [37] is split into three distinct sets, based on the number of steps that it takes a fixed rewriting strategy to normalize the expression. This fixed strategy, LOPL, which stands for *left*

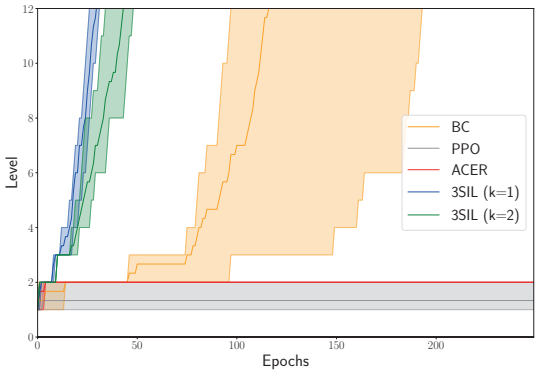

Figure 5.3: The level in the curriculum reached by each method. Each method was run three times. The bold line shows the mean performance and the shaded region shows the minimum and maximum performance. K is the number of proofs stored per problem.

outermost proof length, always rewrites the leftmost possible element. If it takes this strategy less than 90 steps to solve the problem, it is in the *low* difficulty category. Problems with a difficulty between 90 and 130 are in the *medium* category and a greater difficulty than 130 leads to the *high* category. The *high* dataset also contains problems the LOPL strategy could not solve within the time limit. The *low* dataset is split into a training and testing set. We train on the *low* difficulty problems, but after training we also test on problems with a higher difficulty. Because we have a difficulty measure for this dataset, we use a curriculum setup. We start by learning to normalize the expressions that a fixed strategy can normalize in a small number of steps. This setup is similar to [37].

Training setup The 400 problems with the lowest difficulty are the starting point. Every time an agent reaches a 95 percent success rate when evaluated on a sample of size 400 from these problems, we add 400 more difficult problems to the set of training problems P. One iteration of the *collection* and *training* phase is called an *epoch*. Agents are evaluated after every epoch. The blocks of size 400 are called *levels*. The number of episodes m and f are set to 1000. For 3SIL and BC, the batch size BS is 32 and the number of batches NB is 250. The baselines are configured so that the number of episodes and training transitions is at least as many as the 3SIL/BC approaches. Episodes that take over 100 steps are stopped. ADAM [68] is used as an optimizer.

Results on RA curriculum In Figure 5.3, we show the progression through the training curriculum for behavioral cloning (BC), the RL methods (PPO,

ACER) and two configurations of 3SIL. Behavioral cloning simply imitates actions from successful episodes. Of the RL baselines, PPO reaches the second level in one run, while ACER steadily solves the first level and in the best run solves around 80% of the second level. Both methods do not learn enough solutions for the second level to advance to the third. A2C and SIL-PAAC do not reach the second level, so these are left out of the plot. However, they do learn to solve about 70-80% of the first 400 problems. From these results we can conclude that the RL baselines do not perform well on this task in our experiment. We attribute this to the difficulty of learning a good value function due to the sparse rewards (Section 5.5.1). Our hypothesis is that because this value estimate influences the policy updates, the RL methods do not learn well on this task. Note that the two methods with a trust region update mechanism, ACER and PPO, perform better than the methods without this mechanism. From these results, it is clear that 3SIL with 1 shortest proof stored, $k = 1$, is the best-performing configuration. It reaches the end of the training curriculum of about 5000 problems in 40 epochs. We experimented with $k = 3$ and $k = 4$, but these were both worse than $k = 2$.

Generalization While our approach works well on the training set, we must check if the predictors generalize to unseen examples. Only the methods that reached the end of the curriculum are tested. In Table 5.1, we show the results of evaluating the performance of our predictors on the three different test sets: the unseen examples from the *low* dataset and the unseen examples from the *medium* and *high* datasets. Because we expect longer solutions, the episode limits are expanded from 100 steps to 200 and 250 for the *medium* and *high* datasets respectively. For the *low* and *medium* datasets, the second of which contains problems with more difficult solutions than the training data, the predictors solve almost all test problems. For the *high* difficulty dataset, the performance drops by at least 20 percentage points. Our method outperforms the Monte Carlo Tree Search approach used in [37] on the same datasets, which got to 0.954 on the *low* dataset with 1600 iterations and 0.786 on the *medium* dataset (no results on the *high* dataset were reported). These results indicate that this training method might be strong enough to perform well on the AIM rewriting RL task.

5.7.2 AIM Conjecture Dataset

Training setup

Finally, we train and evaluate 3SIL on the AIM Conjecture dataset. We apply 3SIL ($k = 1$) to train predictors in the AIMLEAP environment. Ten percent of

Table 5.1: Generalization with greedy evaluation on the test set for the Robinson arithmetic normalization tasks, shown as average success rate and standard deviation from 3 training runs. Generalization is high on the low and medium difficulty (training data is similar to the low difficulty dataset). With high difficulty data, performance drops.

	Low	Medium	High
3SIL (k=1)	1.00 ± 0.01	0.98 ± 0.03	0.77 ± 0.10
3SIL (k=2)	0.99 ± 0.00	0.96 ± 0.01	0.66 ± 0.08
BC	0.98 ± 0.01	0.98 ± 0.01	0.56 ± 0.05

the AIM dataset is used as a hold-out test set, not seen during training. As there is no estimate for the difficulty of the problems in terms of the actions available to the predictor, we do not use a curriculum ordering for these experiments. The number m of episodes collected before training is set to 2,000,000. These random proof attempts result in about 300 proofs. The predictor learns from these proofs and afterwards the search for new proofs is also guided by its predictions. For the AIM experiments, episodes are stopped after 30 steps in the AIMLEAP environment. The predictors are trained for 100 epochs. The number of collected episodes per epoch f is 10,000. The successful proofs are stored, and the shortest proof for each theorem is kept. NB is 500 and BS is set to 32. The number of problems with a solution in the history after each epoch of the training run is shown in Figure 5.4.

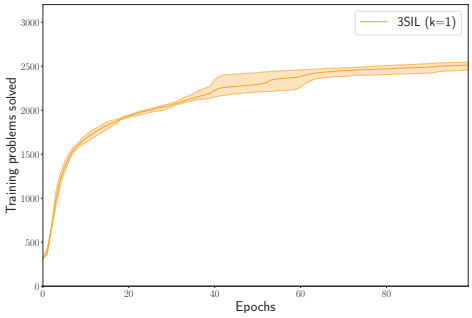

Figure 5.4: The number of training problems for which a solution was encountered and stored (cumulative). At the start of the training, the models rapidly collect more solutions, but after 100 epochs, the process slows down and settles at about 2500 problems with known solutions. The minimum, maximum and mean of three runs are shown.

Results as a standalone prover

After 100 epochs, about 2500 of 3114 problems in the training dataset have a solution in their history. To test the generalization capability of the predictors, we inspect their performance on the holdout test set problems. In Table 5.2 we compare the success rate of the trained predictors on the holdout test set with three different automated theorem provers: E [111, 113], Waldmeister [52] and Prover9. E is currently one of the best overall automated theorem provers [124], Waldmeister is a prover specialized in memory-efficient equational theorem proving [53] and Prover9 is the theorem prover that is used for AIM conjecture research and the prover that the dataset was generated by. Waldmeister and E are the best performing solvers in competitions for the relevant unit equality (UEQ) category [124].

Table 5.2: Theorem proving performance on the hold-out test set in fraction of problems solved. Means and standard deviations are the results of evaluations of 3 different predictors from 3 different training runs on the 354 unseen test set problems.

Method	Success Rate
Prover9 (60s)	0.833
E (60s)	0.802
Predictor + AIMLEAP(60s)	0.702 ± 0.015
Waldmeister (60s)	0.655
Predictor + AIMLEAP (1x)	0.586 ± 0.029

The results show that a single greedy evaluation of the predictor trying to solve the problem in the AIMLEAP environment is not as strong as the theorem proving software. However, the theorem provers got 60 seconds of execution time, and the execution of the predictor, including interaction with AIMLEAP, takes on average less than 1 second. We allowed the predictor setup to use 60 seconds, by running attempts in AIMLEAP until the time was up, sampling actions from the predictor's distribution with 5% noise, instead of using greedy execution. With this approach, the predictor setup outperforms Waldmeister.[5] Figure 5.5 shows the overlap between the problems solved by each prover. The diagram shows that each theorem prover found a few solutions that no other

[5] After the initial experiments, we also evaluated Twee [121], which won the most recent UEQ track: it can prove most of the test problems in 60s, only failing for 1 problem.

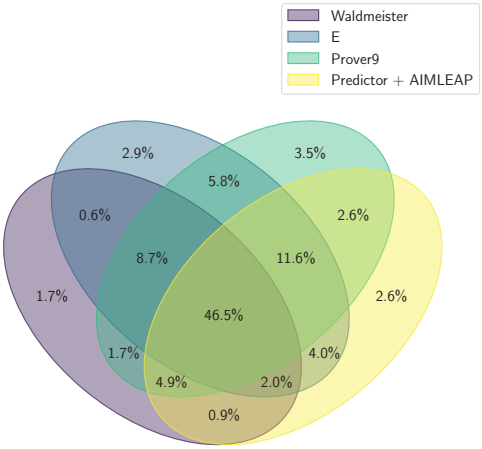

Figure 5.5: Venn diagram of the test set problems solved by each solver with 60s time limit.

prover could find within the time limit. Almost half of all problems from the test set that are solved are solved by all four systems.

Results of neural rewriting combined with Prover9

We also combine the predictor with *Prover9*. In this setup, the predictor modifies the starting form of the goal, for a maximum of 1 second in the AIMLEAP environment. This produces new expressions on one or both sides of the equality. We then add, as lemmas, equalities between the left-hand side of the goal before the predictor's rewriting and after each rewriting (see Figure 5.1). The same is done for the right-hand side. For each problem, this procedure yields new lemmas that are added to the problem specification file that is given to *Prover9*.

In Table 5.3, it is shown that adding lemmas suggested by the rewriting actions of the trained predictor improves the performance of *Prover9*. Running *Prover9* for 2 seconds results in better performance than running it for 1 second, as expected. The combined (1s + 1s) system improved on *Prover9's* 2-second performance by 12.7% (= 0.841/0.746), indicating that the predictor suggests useful lemmas. Additionally, 1 second of neural rewriting combined with 59 seconds of Prover9 search proves almost 8.3% (= 0.902/0.833) more theorems than Prover9 with a 60 second time limit (Table 5.2).

Table 5.3: Prover9 theorem proving performance on the hold-out test set when injecting lemmas suggested by the learned predictor. *Prover9*'s performance increases when using the suggested lemmas.

Method	Success Rate
Prover9 (1s)	0.715
Prover9 (2s)	0.746
Prover9 (60s)	0.833
Rewriting (1s) + Prover9 (1s)	0.841 ± 0.019
Rewriting (1s) + Prover9 (59s)	**0.902** ± 0.016

5.7.3 Implementation Details

All experiments for the Robinson task were run on a 16 core Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz. The AIM experiments were run on a 72 core Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz. All calculations were done on CPU. The PPO implementation was adapted from an existing implementation [7]. The model was updated every 2000 timesteps, the PPO clip coefficient was set to 0.2. The learning rate was 0.002 and the discount factor γ was set to 0.99. The ACER implementation was adapted from an available implementation [20]. The replay buffer size was 20,000. The truncation parameter was 10 and the model was updated every 100 steps. The replay ratio was set to 4. Trust region decay was set to 0.99 and the constraint was set to 1. The discount factor was set to 0.99 and the learning rate to 0.001. Off-policy minibatch size was set to 1. The A2C and SIL implementations were based on Pytorch actor-critic example code available at the PyTorch repository [98]. For the A2C algorithm, we experimented with two formulations of the advantage function: the 1-step lookahead estimate $(r_t + \gamma \mathcal{V}_\mu(s_{t+1})) - \mathcal{V}_\mu(s_t)$ and the $R_t - \mathcal{V}_\mu(s_t)$ formulation. However, we did not observe different performance, so we opted in the end for the 1-step estimate favored in the original A2C publication. For SIL-PAAC, we implemented the SIL loss on top of the A2C implementation. There is also a prioritized replay buffer with an exponent of 0.6, as in the original paper. Each epoch, 8000 (250 batches of size 32) transitions were taken from the prioritized replay buffer in the SIL step of the algorithm. The size of the prioritized replay buffer was 40,000. The critic loss weight was set to 0.01 as in the original paper. For the 3SIL and behavioral cloning implementations, we sample 8000 transitions (250 batches of size 32) from the replay buffer or history. For behavioral

cloning, we used a buffer of size 40,000. An example implementation of 3SIL can be found in the RewriteRL repository. On the Robinson arithmetic task, for 3SIL and BC, the evaluation is done greedily (always take the highest probability actions). For the other methods, we performed experiments with both greedy and non-greedy (sample from the predictor distribution and add 5% noise) evaluation and show the results of the best-performing setting (which in most cases was the non-greedy evaluation, except for PPO). On the AIM task, we evaluate greedily with 3SIL.

AIMLEAP expects a distance estimate for each applicable action. This represents the estimated distance to a proof. This behavior was converted to a reinforcement learning setup by always setting the chosen action of the model to the minimum distance and all other actions to a distance larger than the maximum proof length. Only the chosen action is then carried out.

Versions of the automated theorem provers used: Version 2.5 of E [114], the Nov 2017 version of Prover9 [81] and the Feb 2018 version of Waldmeister [54] and version 2.4.1 of Twee [122].

5.8 Conclusion and Future Work

Our experiments show that a neural rewriter, trained with the 3SIL method that we designed, can learn to suggest useful lemmas that assist an ATP and improve its proving performance. With the same limit of 1 minute, Prover9 managed to prove close to 8.3% more theorems. Furthermore, our 3SIL training method is powerful enough to train an equational prover from zero knowledge that can compete with hand-engineered provers, such as Waldmeister. Our system on its own proves 70.2% of the unseen test problems in 60s, while Waldmeister proved 65.5%.

In future work, we will apply our method to other equational reasoning tasks. An especially interesting research direction concerns selecting which proofs to learn from: some sub-proofs might be more general than other sub-proofs. The incorporation of graph neural networks instead of tree neural networks may improve the performance of the predictor, since in graph neural networks information not only propagates from the leaves to the root, but also through all other connections.

Acknowledgements

We would like to thank Chad Brown for his work with the AIMLEAP software. In addition, we thank Thibault Gauthier and Bartosz Piotrowski for their help with the Robinson arithmetic rewriting task and the AIM rewriting task respectively. We also thank the referees of the IJCAR conference for their useful comments.

This work was partially supported by the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466 (JP, JU), Amazon Research Awards (JP, JU) and by the Czech MEYS under the ERC CZ project *POSTMAN* no. LL1902 (JP, MJ).

Chapter 6

Conclusion

Here, a summary and discussion of the general conclusions of the research work in this thesis are included. There were overarching trends that may be useful to emphasize. With the benefit of hindsight, patterns in the conception, development, and execution of the research work can be observed. The main focus of this thesis was to guide automated theorem proving systems with machine learning heuristics. To be more specific, the projects focused on using graph neural network heuristics to analyze the proof states and predict quantities that influence the instantiation and rewriting choices of the various systems. First, what was accomplished in the research projects associated with each Chapter is briefly stated, and then a more general perspective using the experiences from the projects is provided.

In Chapter 2, it was investigated how a neural network could be used to predict the proof instantiations based on a first-order formula. Our chosen approach, with a graph neural network for embedding the formula and a recurrent neural network for the prediction of the right symbols to construct terms, was able to learn how to prove using instantiations.

Chapter 3 contained a description of how a graph neural network can be used to guide an instantiation calculus prover. In particular, the guidance for the clause selection mechanism inside the prover was targeted.

In Chapter 4, it was explained how the instantiation process within an SMT solver could be controlled by a graph neural network. In this instance, the network was equipped with the ability to predict which part of the formula to instantiate and which terms to use.

Chapter 5 described the use of a neural network to control an equational

rewriting system. After training, the system could in many cases successfully use rewriting rules to complete the proof goal.

The three different instantiation projects that form Chapters 2 to 4 together illustrated the main choice points to be considered when integrating machine learning methods, especially computationally costly ones, into fast-paced automated theorem provers. A balance has to be maintained between fine-grained control, computational requirements, difficulty of implementation, and predictor performance.

In the project described in Chapter 2, all the first-order instantiation reasoning was supposed to be done by the neural network component, while the ground reasoning was done by a non-ML component. This turned out to be a challenging task for the neural network, as the predictions for multiple different clauses had to be consistent to arrive at a proof. In this project, the framing of the task was perhaps too difficult for the machine learning methods. From this, a lesson was learned to involve more capacities of the already existing prover systems in the process.

The project described in Chapter 3 scaled back the task of the machine learning component: the task here was to predict which clauses would be useful for the instantiation calculus, and the rest of the first-order reasoning then was handled by iProver as normal. This separation of concerns led to improved performance.

The third project, described in Chapter 4, occupied a space in between the other two instantiation projects: here, the graph neural network predicted the clause scores and term scores which were used in the enumerative instantiation procedure. Although we predicted multiple quantities that, in principle, would allow quite fine-grained control over the solver, we could fall back on the enumeration procedure when the predictions did not deliver an immediately useful result. Compared especially to the work described in Chapter 2, where the ML component selected all instantiations without a fail-safe mechanism, this kind of hybrid architecture was a more elegant option.

In all of the projects, the extraction of the training data from the solvers posed technical and conceptual challenges. Even though integrating deeply into a solver gave potentially more control, it was not always clear how to cleanly obtain all relevant prover state information. Even after the training data corresponding to a proof was obtained, conceptual difficulties remained around the existence of multiple proofs. While we often chose to use the proofs obtained in the smallest number of steps, it remains to be seen whether preferring certain types of proofs as training data could lead to more generalizable machine learning heuristics for theorem proving.

In general, it was shown that integrating graph neural networks with an automated theorem proving system was possible in instantiation-based proving systems and rewriting systems and that it could improve the decision-making of the provers in certain circumstances. However, there are many more possible future paths that can be explored.

In particular, decreasing the difference in speed between the fast ATP systems and the comparatively slow graph neural network systems would be a useful avenue of future research. In the last few years, many techniques, such as quantization of network parameters and network distillation, have shown promise, and these could be applied to the situations described in this thesis. Another way to handle this problem would be a smart method to interleave the ATP and ML systems: perhaps, the expensive ML predictions can be limited to decisions that are particularly difficult for the existing systems.

Bibliography

[1] Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., and Urban, J. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning 52*, 2 (2014), 191–213.

[2] Aleksandrova, K., Jakubův, J., and Kaliszyk, C. Prover9 unleashed: Automated configuration for enhanced proof discovery. In *LPAR 2024: Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning, Port Louis, Mauritius, May 26-31, 2024* (2024), N. S. Bjørner, M. Heule, and A. Voronkov, Eds., vol. 100 of *EPiC Series in Computing*, EasyChair, pp. 360–369.

[3] Babbage, C., and Babbage, H. P. *Babbage's calculating engines: being a collection of papers relating to them; their history, and construction.* 1889.

[4] Balunovic, M., Bielik, P., and Vechev, M. T. Learning to solve SMT formulas. In *NeurIPS* (2018), pp. 10338–10349.

[5] Bansal, K., Loos, S. M., Rabe, M. N., Szegedy, C., and Wilcox, S. HOList: An environment for machine learning of higher order logic theorem proving. In *ICML* (2019), vol. 97 of *Proceedings of Machine Learning Research*, PMLR, pp. 454–463.

[6] Barbosa, H., Barrett, C. W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., and Zohar, Y. cvc5: A versatile and industrial-strength SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software,*

ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (2022), D. Fisman and G. Rosu, Eds., vol. 13243 of *Lecture Notes in Computer Science*, Springer, pp. 415–442.

[7] Barhate, N. Implementation of PPO algorithm. `https://github.com/nikhilbarhate99`.

[8] Barrett, C. W., Sebastiani, R., Seshia, S. A., and Tinelli, C. Satisfiability modulo theories. In *Handbook of Satisfiability - Second Edition*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., vol. 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021, pp. 1267–1329.

[9] Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al. DOTA 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680* (2019).

[10] Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., and Pollitt, F. CaDiCaL 2.0. In *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC Canada, July 24-27, 2024, Proceedings, Part I* (2024), A. Gurfinkel and V. Ganesh, Eds., vol. 14681 of *LNCS*, Springer, pp. 133–152.

[11] Blaauwbroek, L., Cerna, D. M., Gauthier, T., Jakubův, J., Kaliszyk, C., Suda, M., and Urban, J. Learning guided automated reasoning: A brief survey. *Logics and Type Systems in Theory and Practice: Essays Dedicated to Herman Geuvers on The Occasion of His 60th Birthday* (2024), 54–83.

[12] Blaauwbroek, L., Olšák, M., Rute, J., Massolo, F. I. S., Piepenbrock, J., and Pestun, V. Graph2tac: Online representation learning of formal math concepts. In *Forty-first International Conference on Machine Learning*.

[13] Blaauwbroek, L., Urban, J., and Geuvers, H. Tactic learning and proving for the Coq proof assistant. In *LPAR* (2020), vol. 73 of *EPiC Series in Computing*, EasyChair, pp. 138–150.

[14] Blaauwbroek, L., Urban, J., and Geuvers, H. The Tactician - A seamless, interactive tactic learner and prover for Coq. In *CICM* (2020), vol. 12236 of *Lecture Notes in Computer Science*, Springer, pp. 271–277.

[15] Blanchette, J. C., Böhme, S., and Paulson, L. C. Extending sledgehammer with SMT solvers. *J. Autom. Reason. 51*, 1 (2013), 109–128.

[16] Blanchette, J. C., Kaliszyk, C., Paulson, L. C., and Urban, J. Hammering towards QED. *J. Formalized Reasoning 9*, 1 (2016), 101–148.

[17] Blanchette, J. C., Ouraoui, D. E., Fontaine, P., and Kaliszyk, C. Machine Learning for Instance Selection in SMT Solving. In *AITP 2019 - 4th Conference on Artificial Intelligence and Theorem Proving* (Obergurgl, Austria, Apr. 2019).

[18] Boole, G. *An investigation of the laws of thought: on which are founded the mathematical theories of logic and probabilities*, vol. 2. Walton and Maberly, 1854.

[19] Brown, C. E., Piotrowski, B., and Urban, J. Learning to advise an equational prover. *Artificial Intelligence and Theorem Proving* (2020).

[20] Chételat, D. Implementation of ACER algorithm. `https://github.com/dchetelat/acer`.

[21] Chvalovský, K., Jakubův, J., Olšák, M., and Urban, J. Learning theorem proving components. In *Automated Reasoning with Analytic Tableaux and Related Methods - 30th International Conference, TABLEAUX 2021, Birmingham, UK, September 6-9, 2021, Proceedings* (2021), A. Das and S. Negri, Eds., vol. 12842 of *Lecture Notes in Computer Science*, Springer, pp. 266–278.

[22] Chvalovský, K., Jakubův, J., Suda, M., and Urban, J. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In *International Conference on Automated Deduction* (2019), Springer, pp. 197–215.

[23] Chvalovský, K., Korovin, K., Piepenbrock, J., and Urban, J. Guiding an instantiation prover with graph neural networks. In *LPAR 2023: Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Manizales, Colombia, 4-9th June 2023* (2023), R. Piskac and A. Voronkov, Eds., vol. 94 of *EPiC Series in Computing*, EasyChair, pp. 112–123.

[24] Clarke, E. M., Henzinger, T. A., Veith, H., Bloem, R., et al. *Handbook of model checking*, vol. 10. Springer, 2018.

[25] Davis, M. The early history of automated deduction. In *Handbook of Automated Reasoning*. Elsevier and MIT Press, 2001, pp. 3–15.

[26] Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. *Communications of the ACM 5*, 7 (1962), 394–397.

[27] Davis, M., and Putnam, H. A Computing Procedure for Quantification Theory. *Journal of the ACM 7*, 1 (1960), 215–215.

[28] De Moura, L., and Bjørner, N. Efficient e-matching for smt solvers. In *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21* (2007), Springer, pp. 183–198.

[29] de Moura, L. M., and Bjørner, N. Z3: An Efficient SMT Solver. In *TACAS* (2008), C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *LNCS*, Springer, pp. 337–340.

[30] Desharnais, M., Vukmirovic, P., Blanchette, J., and Wenzel, M. Seventeen provers under the hammer. In *ITP* (2022), vol. 237 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 8:1–8:18.

[31] Detlefs, D., Nelson, G., and Saxe, J. B. Simplify: A theorem prover for program checking. *J. ACM 52*, 3 (2005), 365–473.

[32] Duarte, A., and Korovin, K. Implementing superposition in iProver (system description). In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II* (2020), N. Peltier and V. Sofronie-Stokkermans, Eds., vol. 12167 of *Lecture Notes in Computer Science*, Springer, pp. 388–397.

[33] Eén, N., and Sörensson, N. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers* (2003), E. Giunchiglia and A. Tacchella, Eds., vol. 2919 of *Lecture Notes in Computer Science*, Springer, pp. 502–518.

[34] El Ouraoui, D. *Méthodes pour le raisonnement d'ordre supérieur dans SMT, Chapter 5.* PhD thesis, Université de Lorraine, 2021.

[35] Freeth, T., Higgon, D., Dacanalis, A., MacDonald, L., Georgakopoulou, M., and Wojcik, A. A model of the cosmos in the ancient greek antikythera mechanism. *Scientific reports 11*, 1 (2021), 5821.

[36] Ganzinger, H., and Korovin, K. New directions in instantiation-based theorem proving. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings* (2003), IEEE Computer Society, pp. 55–64.

[37] Gauthier, T. Deep reinforcement learning in HOL4. *arXiv preprint arXiv:1910.11797v1* (2019).

[38] Gauthier, T. Deep reinforcement learning for synthesizing functions in higher-order logic. In *LPAR* (2020), vol. 73 of *EPiC Series in Computing*, EasyChair, pp. 230–248.

[39] Gauthier, T. Tree neural networks in HOL4. In *International Conference on Intelligent Computer Mathematics* (2020), Springer, pp. 278–283.

[40] Gauthier, T., Kaliszyk, C., and Urban, J. TacticToe: Learning to reason with HOL4 tactics. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017* (2017), pp. 125–143.

[41] Gauthier, T., Kaliszyk, C., Urban, J., Kumar, R., and Norrish, M. Tactictoe: Learning to prove with tactics. *J. Autom. Reason. 65*, 2 (2021), 257–286.

[42] Ge, Y., and de Moura, L. M. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification, 21st International Conference, CAV* (2009), pp. 306–320.

[43] Gilmore, P. C. A proof method for quantification theory: Its justification and realization. *IBM Journal of Research and Development 4*, 1 (1960), 28–35.

[44] Goertzel, Z. A. Make E smart again (short paper). In *IJCAR (2)* (2020), vol. 12167 of *Lecture Notes in Computer Science*, Springer, pp. 408–415.

[45] Goertzel, Z. A., Chvalovský, K., Jakubův, J., Olšák, M., and Urban, J. Fast and slow enigmas and parental guidance. In *Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021, Birmingham, UK, September 8-10, 2021, Proceedings* (2021), B. Konev and G. Reger, Eds., vol. 12941 of *Lecture Notes in Computer Science*, Springer, pp. 173–191.

[46] Goertzel, Z. A., Jakubův, J., Kaliszyk, C., Olsák, M., Piepenbrock, J., and Urban, J. The Isabelle ENIGMA. In *ITP* (2022), vol. 237 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 16:1–16:21.

[47] Graves, A., Fernández, S., Gomez, F., and Schmidhuber, J. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning* (2006), pp. 369–376.

[48] He, H., Daume III, H., and Eisner, J. M. Learning to search in branch and bound algorithms. *Advances in Neural Information Processing Systems 27* (2014), 3293–3301.

[49] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 770–778.

[50] Herbrand, J. *Recherches sur la théorie de la démonstration*. Doctorat d'état, La Faculté des Sciences de Paris, 1930.

[51] Heule, M. J., and Biere, A. Proofs for satisfiability problems. *All about Proofs, Proofs for all* (2015).

[52] Hillenbrand, T. Citius altius fortius: Lessons learned from the theorem prover WALDMEISTER. *ENTCS 86*, 1 (2003), 9–21.

[53] Hillenbrand, T., Buch, A., Vogt, R., and Löchner, B. WALDMEISTER - High-Performance Equational Deduction. *Journal of Automated Reasoning 18* (2004), 265–270.

[54] Hillenbrand, T., Buch, A., Vogt, R. and Löchner, B. Waldmeister. `https://www.mpi-inf.mpg.de/departments/automation-of-logic/software/waldmeister/download`.

[55] Holden, S. B., et al. Machine learning for automated theorem proving: Learning to solve sat and qsat. *Foundations and Trends® in Machine Learning 14*, 6 (2021), 807–989.

[56] Irsoy, O., and Cardie, C. Deep recursive neural networks for compositionality in language. *Advances in Neural Information Processing Systems 27* (2014), 2096–2104.

[57] Jakubův, J., Chvalovský, K., Olšák, M., Piotrowski, B., Suda, M., and Urban, J. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II* (2020), N. Peltier and V. Sofronie-Stokkermans, Eds., vol. 12167 of *Lecture Notes in Computer Science*, Springer, pp. 448–463.

[58] Jakubův, J., and Urban, J. ENIGMA: efficient learning-based inference guiding machine. In *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings* (2017), H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke, Eds., vol. 10383 of *Lecture Notes in Computer Science*, Springer, pp. 292–302.

[59] Jakubův, J., and Urban, J. Hammering Mizar by learning clause guidance. In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA* (2019), J. Harrison, J. O'Leary, and A. Tolmach, Eds., vol. 141 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 34:1–34:8.

[60] Jakubův, J., Chvalovský, K., Goertzel, Z. A., Kaliszyk, C., Olsák, M., Piotrowski, B., Schulz, S., Suda, M., and Urban, J. MizAR 60 for Mizar 50. In *ITP* (2023), vol. 268 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 19:1–19:22.

[61] Jakubův, J., Suda, M., and Urban, J. Automated invention of strategies and term orderings for vampire. In *GCAI 2017, 3rd Global Conference on Artificial Intelligence, Miami, FL, USA, 18-22 October 2017* (2017), C. Benzmüller, C. L. Lisetti, and M. Theobald, Eds., vol. 50 of *EPiC Series in Computing*, EasyChair, pp. 121–133.

[62] Janota, M., Barbosa, H., Fontaine, P., and Reynolds, A. Fair and adventurous enumeration of quantifier instantiations. In *2021 Formal Methods in Computer Aided Design (FMCAD)* (2021), IEEE, pp. 256–260.

[63] Janota, M., Piepenbrock, J., and Piotrowski, B. Towards learning quantifier instantiation in SMT. In *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel* (2022), K. S. Meel and O. Strichman, Eds., vol. 236 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 7:1–7:18.

[64] Kaliszyk, C., and Urban, J. Learning-assisted automated reasoning with Flyspeck. *J. Autom. Reasoning 53*, 2 (2014), 173–213.

[65] Kaliszyk, C., and Urban, J. FEMaLeCoP: Fairly efficient machine learning connection prover. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings* (2015), M. Davis, A. Fehnker, A. McIver, and A. Voronkov, Eds., vol. 9450 of *Lecture Notes in Computer Science*, Springer, pp. 88–96.

[66] Kaliszyk, C., and Urban, J. MizAR 40 for Mizar 40. *J. Autom. Reasoning 55*, 3 (2015), 245–256.

[67] Kaliszyk, C., Urban, J., Michalewski, H., and Olšák, M. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.* (2018), pp. 8836–8847.

[68] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015), Y. Bengio and Y. LeCun, Eds.

[69] Kinyon, M. Proof simplification and automated theorem proving. *CoRR abs/1808.04251* (2018).

[70] Kinyon, M., Veroff, R., and Vojtěchovský, P. Loops with abelian inner mapping groups: An application of automated deduction. In *Automated Reasoning and Mathematics*. Springer, 2013, pp. 151–164.

[71] Komendantskaya, E., Heras, J., and Grov, G. Machine learning in Proof General: Interfacing interfaces. In *UITP* (2012), vol. 118 of *EPTCS*, pp. 15–41.

[72] Korovin, K. iProver - an instantiation-based theorem prover for first-order logic (system description). In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings* (2008), A. Armando, P. Baumgartner, and G. Dowek, Eds., vol. 5195 of *Lecture Notes in Computer Science*, Springer, pp. 292–298.

[73] Korovin, K. Inst-Gen - A modular approach to instantiation-based automated reasoning. In *Programming Logics - Essays in Memory of Harald Ganzinger* (2013), A. Voronkov and C. Weidenbach, Eds., vol. 7797 of *Lecture Notes in Computer Science*, Springer, pp. 239–270.

[74] Kovács, L., and Voronkov, A. First-order theorem proving and Vampire. In *CAV* (2013), N. Sharygina and H. Veith, Eds., vol. 8044 of *LNCS*, Springer, pp. 1–35.

[75] Kühlwein, D., van Laarhoven, T., Tsivtsivadze, E., Urban, J., and Heskes, T. Overview and evaluation of premise selection techniques for large theory mathematics. In *IJCAR* (2012), B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364 of *LNCS*, Springer, pp. 378–392.

[76] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *Nature 521*, 7553 (2015), 436–444.

[77] Li, M., and Vitányi, P. *An introduction to Kolmogorov complexity and its applications*. Springer-Verlag, 2008.

[78] Marques-Silva, J., and Sakallah, K. A. Grasp-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design* (1996), IEEE, pp. 220–227.

[79] Marques-Silva, J. P., Lynce, I., and Malik, S. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., vol. 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009, pp. 131–153.

[80] McCune, W. Prover9 and Mace. `http://www.cs.unm.edu/~mccune/prover9/`, 2010.

[81] McCune, W. Prover9. `https://github.com/ai4reason/Prover9`.

[82] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (2016), pp. 1928–1937.

[83] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529–533.

[84] Moskal, M., Łopuszański, J., and Kiniry, J. R. E-matching for fun and profit. *Electronic Notes in Theoretical Computer Science 198*, 2 (2008), 19–35.

[85] Nieuwenhuis, R., and Oliveras, A. Fast congruence closure and extensions. *Inf. Comput. 205*, 4 (2007), 557–580.

[86] Nieuwenhuis, R., Oliveras, A., and Tinelli, C. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM) 53*, 6 (2006), 937–977.

[87] Oh, J., Guo, Y., Singh, S., and Lee, H. Self-imitation learning. In *International Conference on Machine Learning* (2018), pp. 3878–3887.

[88] Olšák, M., Kaliszyk, C., and Urban, J. Property invariant embedding for automated reasoning. In *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), G. D. Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, Eds., vol. 325 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, pp. 1395–1402.

[89] Overbeek, R. A. A new class of automated theorem-proving algorithms. *J. ACM 21*, 2 (Apr. 1974), 191–200.

[90] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[91] Phillips, J., and Stanovský, D. Automated theorem proving in quasigroup and loop theory. *AI Communications 23*, 2-3 (2010), 267–283.

[92] Piepenbrock, J., Heskes, T., Janota, M., and Urban, J. Guiding an automated theorem prover with neural rewriting. In *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings* (2022), J. Blanchette, L. Kovács,

and D. Pattinson, Eds., vol. 13385 of *Lecture Notes in Computer Science*, Springer, pp. 597–617.

[93] Piepenbrock, J., Janota, M., Urban, J., and Jakubův, J. First experiments with neural cvc5. In *LPAR 2024: Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning, Port Louis, Mauritius, May 26-31, 2024* (2024), N. S. Bjørner, M. Heule, and A. Voronkov, Eds., vol. 100 of *EPiC Series in Computing*, EasyChair, pp. 264–277.

[94] Piepenbrock, J., Urban, J., Korovin, K., Olšák, M., Heskes, T., and Janota, M. Invariant neural architecture for learning term synthesis in instantiation proving. *Journal of Symbolic Computation 128* (2025), 102375.

[95] Pimpalkhare, N., Mora, F., Polgreen, E., and Seshia, S. A. MedleySolver: Online SMT algorithm selection. In *Theory and Applications of Satisfiability Testing – SAT 2021* (Cham, 2021), C.-M. Li and F. Manyà, Eds., Springer International Publishing, pp. 453–470.

[96] Plaisted, D. A., and Greenbaum, S. A structure-preserving clause form translation. *J. Symb. Comput. 2*, 3 (1986), 293–304.

[97] Post, E. The two-valued iterative systems of mathematical logic. *Annals of Mathematics Studies* (1941).

[98] PyTorch. RL Examples. `https://github.com/pytorch/examples/tree/main/reinforcement_learning`.

[99] Rawson, M., and Reger, G. lazycop: Lazy paramodulation meets neurally guided search. In *TABLEAUX* (2021), vol. 12842 of *Lecture Notes in Computer Science*, Springer, pp. 187–199.

[100] Reynolds, A., Barbosa, H., and Fontaine, P. Revisiting enumerative instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II 24* (2018), Springer, pp. 112–131.

[101] Reynolds, A., Tinelli, C., and de Moura, L. M. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014* (2014), IEEE, pp. 195–202.

[102] Riazanov, A., and Voronkov, A. The design and implementation of VAMPIRE. *AI Commun. 15*, 2-3 (2002), 91–110.

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

[104] Ross, S., Gordon, G., and Bagnell, D. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics* (2011), pp. 627–635.

[105] Sanchez-Stern, A., Alhessi, Y., Saul, L., and Lerner, S. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (New York, NY, USA, 2020), MAPL 2020, Association for Computing Machinery, p. 1–10.

[106] Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *IEEE transactions on neural networks 20*, 1 (2008), 61–80.

[107] Schlichtkrull, M., Kipf, T. N., Bloem, P., Van Den Berg, R., Titov, I., and Welling, M. Modeling relational data with graph convolutional networks. In *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15* (2018), Springer, pp. 593–607.

[108] Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning* (Lille, France, 07–09 Jul 2015), F. Bach and D. Blei, Eds., vol. 37 of *Proceedings of Machine Learning Research*, PMLR, pp. 1889–1897.

[109] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[110] Schulz, S. A Comparison of Different Techniques for Grounding Near-Propositional CNF Formulae. In *Proc. of the 15th FLAIRS, Pensacola* (2002), S. Haller and G. Simmons, Eds., AAAI Press, pp. 72–76.

[111] Schulz, S. E - A Brainiac Theorem Prover. *AI Commun. 15*, 2-3 (2002), 111–126.

[112] Schulz, S. System description: E 1.8. In *LPAR* (2013), K. L. McMillan, A. Middeldorp, and A. Voronkov, Eds., vol. 8312 of *LNCS*, Springer, pp. 735–743.

[113] Schulz, S., Cruanes, S., and Vukmirovic, P. Faster, higher, stronger: E 2.3. In *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings* (2019), P. Fontaine, Ed., vol. 11716 of *Lecture Notes in Computer Science*, Springer, pp. 495–507.

[114] Schulz, S. Eprover. `https://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html`.

[115] Scott, J., Niemetz, A., Preiner, M., Nejati, S., and Ganesh, V. MachSMT: A machine learning-based algorithm selector for SMT solvers. In *Tools and Algorithms for the Construction and Analysis of Systems* (Cham, 2021), J. F. Groote and K. G. Larsen, Eds., Springer International Publishing, pp. 303–325.

[116] Selsam, D., and Bjørner, N. Guiding high-performance sat solvers with unsat-core predictions. In *Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22* (2019), Springer, pp. 336–353.

[117] Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. Learning a SAT solver from single-bit supervision. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019* (2019), OpenReview.net.

[118] Si, X., Dai, H., Raghothaman, M., Naik, M., and Song, L. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems 31* (2018).

[119] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *Nature 529*, 7587 (2016), 484–489.

[120] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of go without human knowledge. *Nature 550*, 7676 (2017), 354–359.

[121] Smallbone, N. Twee: An equational theorem prover. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings* (2021), A. Platzer and G. Sutcliffe, Eds., vol. 12699 of *Lecture Notes in Computer Science*, Springer, pp. 602–613.

[122] Smallbone, N. Twee 2.4.1. `https://github.com/nick8325/twee/releases/download/2.4.1/twee-2.4.1-linux-amd64`.

[123] Suda, M. Improving enigma-style clause selection while learning from history. In *CADE* (2021), vol. 12699 of *Lecture Notes in Computer Science*, Springer, pp. 543–561.

[124] Sutcliffe, G. The CADE-27 automated theorem proving system competition - CASC-27. *AI Communications 32*, 5-6 (2020), 373–389.

[125] Sutton, R. S., and Barto, A. G. *Reinforcement learning: An introduction*. 2018.

[126] Torabi, F., Warnell, G., and Stone, P. Behavioral cloning from observation. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (2018), IJCAI'18, AAAI Press, p. 4950–4957.

[127] Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc. s2-42*, 1 (1937), 230–265.

[128] Urban, J. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning 37*, 1-2 (2006), 21–43.

[129] Urban, J., and Jakubův, J. First neural conjecturing datasets and experiments. In *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings* (2020), C. Benzmüller and B. R. Miller, Eds., vol. 12236 of *Lecture Notes in Computer Science*, Springer, pp. 315–323.

[130] Urban, J., Vyskočil, J., and Štěpánek, P. MaLeCoP: Machine learning connection prover. In *TABLEAUX* (2011), K. Brünnler and G. Metcalfe, Eds., vol. 6793 of *LNCS*, Springer, pp. 263–277.

[131] Veroff, R. Using hints to increase the effectiveness of an automated reasoning program: Case studies. *J. Autom. Reason. 16*, 3 (1996), 223–239.

[132] Voronkov, A. AVATAR: the architecture for first-order theorem provers. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings* (2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 696–710.

[133] Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. Sample efficient actor-critic with experience replay. *International Conference on Learning Representations* (2016).

[134] Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019* (2019), OpenReview.net.

[135] Zombori, Z., Urban, J., and Olšák, M. The role of entropy in guiding a connection prover. In *TABLEAUX* (2021), vol. 12842 of *Lecture Notes in Computer Science*, Springer, pp. 218–235.

Appendix A: Chapter 2

A.1 Congruence Closure + SAT

After the ML instantiator has instantiated the clauses in a problem, we remove all clauses that still have variables and send the remaining, ground part of the problem to Vampire. We use Vampire with the $--acc$ option that activates congruence closure and use a time limit of 30s. However, the vast majority of executions finish in under 1 second.

A.2 Comparison with Existing Provers

We compare with iProver, another system that in one of its modes relies mainly on instantiation to prove problems. We use the following flags for iProver, to confine it to a pure instantiation-based mode:

```
--schedule none
--superposition_flag false
--resolution_flag false
--inst_prop_sim_given false
--inst_prop_sim_new false
```

For our comparisons with CVC5, we used the default settings, with the newest version as of March 2023.

A.3 Random Instantiator

In Figure A.3.1 we include a plot of the number of training set proofs cumulatively found by the random instantiator on the M2k subset of the training set. For the full dataset, the random instantiator cumulatively solves 9923 training

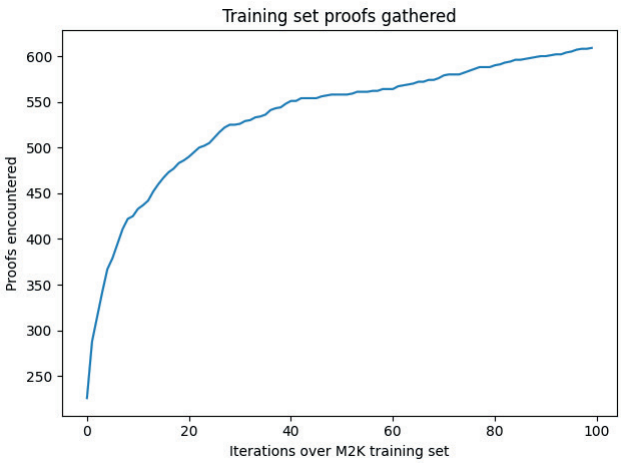

Figure A.3.1: Cumulative M2k solutions by random instantiation limited to the training set.

problems after 100 passes with two instantiation levels.

We have also attempted to extract a training set of instantiation proofs from iProver, but encountered difficulties extracting training data. Exactly tracing all the steps as they happened, transforming them so that the ML system can replicate them, is difficult, especially because of differences in handling of equality. As we expected similar difficulties with other systems, we chose to start from random instantiations, in the exact same setting as our ML predictor.

A.4 Training Curves

Figures A.4.1 and A.4.2 correspond to the model training on the full (non-M2k) training dataset. We can see that the model can reach very high performance on the training set (even achieving 0.95 median accuracy), but stops improving on validation after about 40 epochs.

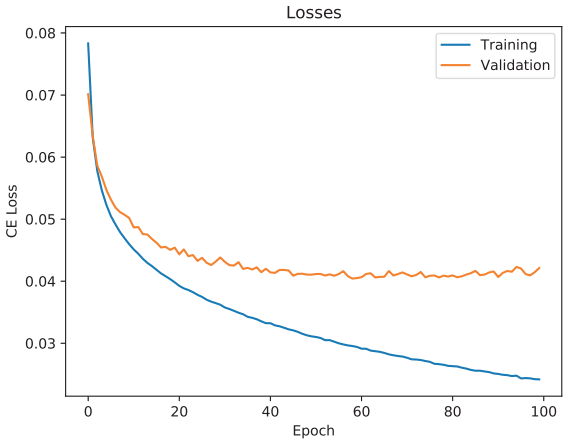

Figure A.4.1: Loss curve for a model trained on the 25-5 random instantiator data.

A.5 Training on Generated Proofs

In some cases, during the looping experiments, multiple different proofs of the same problem can be generated. In that case, we keep the proof with fewer levels of instantiation (if there is a difference) to train on. The underlying idea is that we should train the instantiation system to be as concise as possible and that it should prefer proofs with shallower terms.

A.6 On Keeping General Clauses & Training Data Alignment

When a clause is instantiated, we can either keep the parent clause of the instance around in our set of clauses, or the parent can be deleted. However, deletion would correspond to the assertion that no more instances of that parent clause are necessary anymore. As it is possible for the system to miss crucial instantiations, the parent clauses are kept so that they may be instantiated on the next level again. Deleting the parent clauses was also briefly tested, but led to worse preliminary performance.

This also leads to an interesting issue, as our training data is constructed to

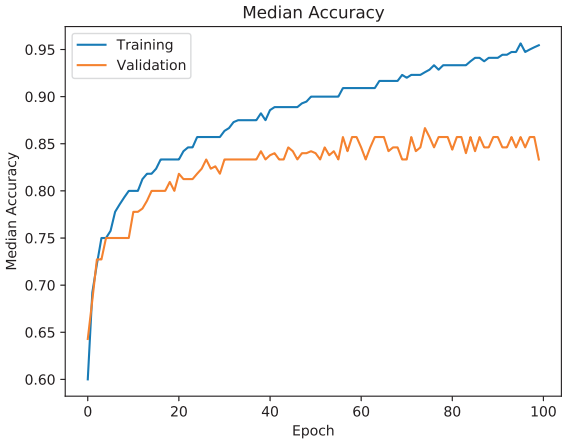

Figure A.4.2: Median accuracy for a model trained on the 25-5 random instantiator data.

not include the parent clauses anymore, when the instances have been created (the training data is cleared of any superfluous steps). However, keeping the parent clauses in the training data did not lead to better results. This remains to be further investigated.

Another potential issue is that during inference time, the system produces unnecessary instantiations, which are then part of the input at the next instantiation level. This means that the input graphs start looking more and more dissimilar from the training data. Preliminary attempts to remedy this difference by also training on the data as it is during inference time however did not lead to better results. There may be a balance to strike in the relative importance of training on the various types of data.

A.7 Scaling & Decomposition of the Task

The GNN part of the approach scales at a maximum quadratically with the number of nodes (the sum of the number of clauses, literals and terms in the parsed problem), in the case of a complete graph. However, our graphs are not close to complete graphs. For the RNN part, the memory requirements scale linearly according to the number of clauses with variables in the problem

C and also linearly with the number of required samples per clause (i.e. 25 or 5). The RNN time requirements in theory scale linearly with the size of the largest instance needed, measured in the number of variables. However in practice we cap the RNN at 12 iterations. The scaling of the method as a whole is dependent on the size (in terms of levels needed to generate the needed terms) of typical proofs and on the accuracy of the instances generated. If the probability distribution that is learned is very focused, only a few unnecessary instances are generated, which keeps the growth of the clause set under control.

The looping experiments, where the network learns from a starting dataset to prove more and more theorems, can take days to complete. On the M2k dataset, one looping iteration (including training procedures) takes around 5 minutes (with 1000 attempts per iteration), whereas it takes around 50–60 minutes for the experiments on the full dataset (with 10 000 attempts per iteration). The M2k looping experiment was run with softmax temperature 2, and for the full dataset experiment the temperature was set to 1. The higher the temperature, the more unique instances are created. With 5 levels of instantiation, too many instances can slow down the procedure.

The total space of possible instantiations is impossible for us to examine. We have therefore chosen to use our level-based approach, where we construct the necessary terms from top to bottom. Other decompositions of the task are possible. For example, new terms could be constructed bottom-up, starting with already existing ground terms.

Having chosen to use the level-based approach, there are still other choices to make. For example, does the current state of the variable-symbol mapping in *other clauses* influence our choice of a symbol for a particular variable in our *current clause*? When considering the task itself, of course the instantiations of other clauses impact the viability of instantiations in the current clause. However, we have chosen to decompose the process in two parts: the graph neural network (GNN) can see the entire graph, but the recurrent neural network (RNN) that produces the instances can only see the variable embeddings for 1 particular clause (and to get instances for all clauses, we run the same RNN in parallel). For the system to work, the coordination of which instantiation strategy to pursue in the RNN must be happening in the GNN part, when the clause, symbol and variable embedding vectors are determined. We found this to be an acceptable trade-off, as it allowed us to consider each clause somewhat separate from the other ones, simplifying and speeding up our instantiation sampling procedure. A probability distribution that considers the combinatorial number of choices made in all the other clauses at all times would be very expensive to evaluate.

A.8 Using other Ground Solvers

We tested whether using CVC5 as our ground solver would change the results. We used three levels of neural instantiations on the test set problems, and removed all clauses with variables. Then, we ran either Vampire (with *acc* argument) or CVC5 (default parameters). We found no difference, with both systems solving the exact same amount of problems: 904, 1909, 2146 at the respective levels. This indicates that the precise choice of ground solver, as long as CC + SAT is present, does not influence our results.

Appendix B: Chapter 3

B.1 Server Settings

We run the server with 48 state workers, 32 GPU workers, and the query max size is 2048. The context is created from a random sample of at most 512 given clauses and all negated conjectures (may be among given clauses). In no eval mode we run the server with the `--zero_scores` option.

The evaluations were run on NVIDIA DGX-1 with dual 20-core Inter E5-2698 v4, 512 GB RAM, and 8 NVIDIA Tesla V100 GPU cards.

B.2 iProver Settings

We compile iProver (branch `2022_sockets`) with `STATIC=true z3=false` in the debug mode (`let dbg_global_flag = true`), which is useful for extracting proofs for further iterations. For every problem, we run
`tptp4X -t noint -u machine -N` to rename integers to constants.

We always run 16 iProvers in parallel on the same machine as we run the server and they communicate via the loopback network interface. The server overhead is relatively small (overall load is usually 22–23) with a GPU utilization between 10–20% even with our biggest models ($d = 32$ and $l = 11$). Time limits (in real time) are always 15 seconds.

The detailed settings are as follows.

Non-interactive mode

```
iproveropt \
 --interactive_mode false \
 --inst_learning_loop_flag false \
```

```
 --schedule none \
 --preprocessing_flag false \
 --instantiation_flag true \
 --superposition_flag false \
 --resolution_flag false \
 --time_out_real 15 \
 --inst_unprocessed_bound 1000
```

Ignore eval and no eval mode

```
iproveropt \
 --interactive_mode true \
 --external_ip_address "127.0.0.1" \
 --external_port "12300" \
 --inst_learning_loop_flag false \
 --schedule none \
 --preprocessing_flag false \
 --instantiation_flag true \
 --superposition_flag false \
 --resolution_flag false \
 --time_out_real 15 \
 --inst_unprocessed_bound 1000
```

Solo mode

```
iproveropt \
 --interactive_mode true \
 --external_ip_address "127.0.0.1" \
 --external_port "12300" \
 --inst_learning_loop_flag false \
 --schedule none \
 --preprocessing_flag false \
 --instantiation_flag true \
 --superposition_flag false \
 --resolution_flag false \
 --inst_passive_queue_type priority_queues \
 --inst_passive_queues_freq "[1]" \
 --inst_passive_queues "[[+external_score]]" \
 --time_out_real 15 \
 --inst_unprocessed_bound 1000
```

Coop mode

```
iproveropt \
 --interactive_mode true \
 --external_ip_address "127.0.0.1" \
 --external_port "12300" \
 --inst_learning_loop_flag false \
 --schedule none \
 --preprocessing_flag false \
 --instantiation_flag true \
 --superposition_flag false \
 --resolution_flag false \
 --inst_passive_queue_type priority_queues \
 --inst_passive_queues_freq "[27;25;2]" \
 --inst_passive_queues "[[+external_score];
                         [-conj_dist;+conj_symb;-num_var];
                         [+age;-num_symb]]" \
 --time_out_real 15 \
 --inst_unprocessed_bound 1000
```

B.3 GNN Settings

We trained models with node, symbol, and clause embeddings of size 16 with 10 layers. In the last iteration, we also tried models with embeddings of size 32 and with 11 layers. We have not performed other hyperparameter searches. Layer normalization is used after every message passing step to ensure training stability.

The models were trained on various servers with NVIDIA GTX 1080 Ti, Tesla V100, and A40 GPUs. We trained each model for roughly two days (with a limit of 100 epochs). The only exception was iteration 1 where we trained models just for one day to get more proofs quickly. The Adam optimization algorithm with a learning rate 0.0005 was used for training.

Research Data Management

This thesis research has been carried out under the research data management policy of the Institute for Computing and Information Sciences of Radboud University, The Netherlands[1]. The chapters themselves contain links to repositories associated with the research projects. Here, those links are collected, along with links to archived versions.

- **Chapter 2:**
  - `https://github.com/JellePiepenbrock/neural-synthesis`
  - DOI: `https://doi.org/10.5281/zenodo.14616518`
- **Chapter 3:**
  - `https://github.com/JellePiepenbrock/iprover-gnn-server`
  - DOI: `https://doi.org/10.5281/zenodo.14616489`
- **Chapter 4:**
  - `https://github.com/JellePiepenbrock/mlcvc5-LPAR`
  - DOI: `https://doi.org/10.5281/zenodo.14616456`
- **Chapter 5:**
  - `https://github.com/JellePiepenbrock/neural_rewriting`
  - DOI: `https://doi.org/10.5281/zenodo.14616272`

[1] `https://www.ru.nl/en/institute-for-computing-and-information-sciences/research`, last accessed January 15, 2025.

Summary

Mathematics can be formalized to allow a computer to execute the proof steps and recognize whether a statement has been proven. Computer systems that automate mathematics are called automated theorem provers (ATP). A collection of assumptions, called axioms, is entered into the system in combination with a goal statement or conjecture. The system has inference rules, which prescribe how the axioms and conjecture may be handled during a search for a proof.

Depending on the exact ATP system, the allowed inference rules may vary. However, the number of possible choices at each point in the process can become very large, which makes the search space towards a proof difficult to navigate. It would be useful to have a system that can predict which choices are better than other ones, so that a proof may be reached earlier in the search procedure.

The field of machine learning is concerned with creating mathematical models that approximate relationships between quantities that are expressed in example data. The process has the following steps: showing the system the expected output quantity that is associated with a particular input, measuring the difference between the expected output and the current prediction of the mathematical model and then modifying the system so that the difference will be smaller. At the end of the procedure, one ends up with a model that predicts the value of the expected output quantity.

In this thesis, automated theorem proving is combined with machine learning. The machine learning component is tasked with learning to predict which choices in the proving process are useful and which are not. This information can be learned from already finished proofs. The overall goal is to improve the performance of the automated theorem proving systems in terms of theorems proven in a particular time period. In particular, a class of machine learning models called graph neural networks is used to process mathematical data and the state information of the automated provers at particular times.

The first research project develops a system that replaces variables in first-order logic problems with more concrete terms in a process called instantiation. This is accomplished with two ML approaches: a graph neural network that creates representations for the symbols and subcomponents of the mathematical problem and a recurrent neural network that predicts which symbols should be used to create the instantiation terms. The system can start learning from scratch to prove 19.7% of the unseen problems in our dataset.

The second research project integrates a graph neural network with the clause selection component of the automated theorem prover iProver. This prover uses an instantiation calculus and the clause selection guides which clauses will be used for instantiation. The predictions of the graph neural network improve the performance of iProver in its instantiation mode.

The third research project in this thesis concerns the integration of a graph neural network into the satisfiability modulo theories (SMT) solver cvc5. The ML predictor can control the instantiation process within cvc5 by choosing the subformula to instantiate, as well as by ranking the terms with which to instantiate that subformula. In a dataset of first-order logic problems, the ML predictor improves the performance of the enumerative instantiation system within cvc5.

In the fourth research project, a tree neural network is used to learn a rewriting policy for a dataset of loop theory problems. The system learns how to solve more than half of the unseen problems from scratch by rewriting using the rewriting rules until the goal is reached.

The work in this thesis shows that guiding automated theorem proving with machine learning is possible for instantiation-based and rewriting-based proving systems. Different methods to integrate machine learning into the provers are explored. The lessons learned can be used in the further development of machine learning heuristics for automated theorem proving systems. Future work is possible in the direction of improving the speed of the predictors and in exploring whether particular kinds of proof are especially useful to train the predictor in comparison with other kinds.

Samenvatting

Wiskunde kan worden geformaliseerd zodat een computer bewijsstappen kan uitvoeren en kan herkennen wanneer een stelling bewezen is. Computersystemen die wiskunde automatiseren worden *automated theorem provers* (Nederlands: automatische stellingbewijzers, afgekort: ATP) genoemd. Een collectie van aannames, axioma's genoemd, wordt ingevoerd in combinatie met een doelstelling. Het systeem heeft regels voor geldige gevolgtrekkingen, die voorschrijven hoe de axioma's en doelstelling mogen worden behandeld gedurende de zoektocht naar een bewijs.

Afhankelijk van het specifieke ATP systeem kunnen de geldige vormen van gevolgtrekking verschillen. Het aantal mogelijke keuzes op ieder moment in het proces kan zeer groot worden, wat het moeilijk maakt door de zoekruimte richting een bewijs te navigeren. Het zou nuttig zijn om een systeem te hebben dat kan voorspellen welke keuzes beter zijn dan andere, zodat eerder in de zoekprocedure een bewijs kan worden gevonden.

Het onderzoeksveld *machine learning* (Nederlands: machinaal leren, afkorting: ML) houdt zich bezig met het creëren van wiskundige modellen die verbanden tussen variabelen benaderen, die kenmerkend zijn in data. Het proces bevat de volgende stappen: Het systeem het verwachte antwoord op basis van bepaalde invoerdata laten zien, het verschil meten tussen het verwachte antwoord en de voorspelling van het model en tot slot het systeem aanpassen zodat dit verschil kleiner wordt. Aan het einde van deze procedure heeft men een model dat waardes voorspelt op basis van de invoerdata.

In dit proefschrift wordt automated theorem proving gecombineerd met machine learning. De machine learning component heeft als taak te leren voorspellen welke keuzes in het bewijsproces nuttig zijn en welke niet. Deze informatie kan worden geleerd uit reeds voltooide bewijzen. Het algemene doel is om het vermogen van de automated theorem proving systemen te verbeteren, uitgedrukt in het aantal stellingen die kunnen worden bewezen in een bepaalde peri-

ode. Er wordt een klasse van machine learning modellen gebruikt die *graph neural networks* (graaf neurale netwerken) heet voor het verwerken van wiskundige data en de informatie over de staat van de automated theorem provers op specifieke punten in het proces.

Het eerste onderzoeksproject in dit proefschrift ontwikkelt een systeem dat variabelen in eerste-orde logicaproblemen vervangt door meer concrete termen in een proces dat *instantiation* (instantiatie) heet. Dit wordt bereikt met twee verschillende ML methodes: een graph neural network dat representaties creëert voor de symbolen en subcomponenenten van het wiskundige probleem en een recurrent neural network (recurrent neuraal netwerk) dat voorspelt welke symbolen gebruikt moeten worden om de instantiatietermen te construeren. Het systeem kan beginnen met leren vanuit het niets totdat het 19.7% van de ongeziene bewijzen in onze dataset kan bewijzen.

Het tweede onderzoeksproject integreert een graph neural network met de *clause selection* component van de automatische bewijzer iProver. Deze bewijzer gebruikt een instantiatiecalculus en clause selection bepaalt welke feiten worden gebruikt voor instantiatie. De voorspellingen van het graph neural network verbeteren de prestaties van iProver in de instantiatiemodus.

Het derde onderzoeksproject behandelt de integratie van een graph neural network met de *satisfiability modulo theories* (SMT) probleemoplosser cvc5. De ML voorspeller kan het instantiatieproces in cvc5 controleren door de subformule om te instantieren te kiezen en door de termen waarmee deze subformule kan worden geïnstantieerd te rangschikken. De ML voorspellingen verbeteren de prestaties van het enumeratieve instantiatiesysteem in cvc5.

In het vierde onderzoeksproject wordt een *tree neural network* gebruikt om een strategie voor het herschrijven van problemen uit een dataset over quasigroepen te leren. Het systeem leert meer dan de helft van de ongeziene problemen in de dataset op te lossen door de problemen te herschrijven tot het doel bereikt is.

Het werk in dit proefschrift laat zien dat het besturen van automated theorem proving met machine learning mogelijk is in het geval van systemen gebaseerd op instantiatie en herschrijven. Verschillende methodes om machine learning te integreren met de bewijzers worden verkend. De lessen die geleerd zijn kunnen worden gebruikt in het verder ontwikkelen van heuristieken gebaseerd op machine learning voor automated theorem proving. Toekomstig werk kan zich richten op het verbeteren van de snelheid van de voorspellers en op het onderzoeken of er een specifieke stijl van bewijs bestaat waar de voorspellers beter van leren dan andere stijlen.

Contributions & Publication List

The introduction to the topics of automated theorem proving and machine learning serving as **Chapter 1** of this PhD thesis was written specifically by me for this purpose and has not appeared anywhere else before. Tom Heskes, Mikoláš Janota, and Josef Urban helped proofread the text.

Chapter 2 is based on a journal paper "Invariant Neural Architecture for Learning Term Synthesis in Instantiation Proving" published in the *Journal of Symbolic Computation*. The implementation of the system, the experimental evaluation, and the writing of the initial text of the paper were done by me. Josef Urban provided the initial random instantiations dataset. Josef Urban and Mikoláš Janota advised the implementation and experimental work. Miroslav Olšák assisted in changing the first-order parser to fulfill the requirements needed for the system. Josef Urban, Konstantin Korovin, Mikoláš Janota, and Tom Heskes helped proofread and improve the text of the paper.

Chapter 3 is based on a conference paper "Guiding an Instantiation Prover with Graph Neural Networks" published at *LPAR 2023*. I implemented the graph neural network and initial integration with the machine learning server code. Konstantin Korovin created a communication API within iProver to interact with the ML server. Karel Chvalovský did the final training and experimental evaluation of the system with the iProver integration. The initial writing of the text of the paper was mostly divided along these lines. Josef Urban advised and proofread the paper.

Chapter 4 is based on a conference paper "First Experiments with Neural cvc5" published at *LPAR 2024*. I implemented the system, did the experimental evaluation of the system, and wrote the initial text of the paper. Mikoláš Janota advised conceptually and also practically on how to integrate with the *cvc5* codebase. Jan Jakubův helped migrate the codebase to a newer version of *cvc5*.

Josef Urban advised on project direction and the Mizar dataset. Everyone helped proofread and improve the paper text.

Chapter 5 is based on a conference paper "Guiding an Automated Theorem Prover with Neural Rewriting" published at *IJCAR 2022*. I implemented the system, did the experimental evaluation, and wrote the initial text of the paper. Tom Heskes, Mikoláš Janota, and Josef Urban advised and helped improve the final text of the paper.

Below is a list of works that I co-authored during the PhD that were accepted and published in scientific venues:

1. Jelle Piepenbrock, Josef Urban, Konstantin Korovin, Miroslav Olšák, Tom Heskes, Mikoláš Janota: Invariant Neural Architecture for Learning Term Synthesis in Instantiation Proving. Journal of Symbolic Computation **(Chapter 2)**

2. Jan Jakubův, Mikoláš Janota, Jelle Piepenbrock and Josef Urban: Machine Learning for Quantifier Selection in cvc5. ECAI 2024

3. Jason Rute, Miroslav Olšák, Lasse Blaauwbroek, Fidel Ivan Schaposnik Massolo, Jelle Piepenbrock, Vasily Pestun: Graph2Tac: Online Representation Learning of Formal Math Concepts. ICML 2024

4. Jelle Piepenbrock, Mikoláš Janota, Josef Urban, Jan Jakubův: First Experiments with Neural cvc5. LPAR 2024 **(Chapter 4)**

5. Pedro Orvalho, Jelle Piepenbrock, Mikoláš Janota, Vasco M. Manquinho: Graph Neural Networks for Mapping Variables Between Programs. ECAI 2023

6. Karel Chvalovský, Konstantin Korovin, Jelle Piepenbrock, Josef Urban: Guiding an Instantiation Prover with Graph Neural Networks. LPAR 2023 **(Chapter 3)**

7. Jelle Piepenbrock, Tom Heskes, Mikoláš Janota, Josef Urban: Guiding an Automated Theorem Prover with Neural Rewriting. IJCAR 2022 **(Chapter 5)**

8. Zarathustra Amadeus Goertzel, Jan Jakubův, Cezary Kaliszyk, Miroslav Olšák, Jelle Piepenbrock, Josef Urban: The Isabelle ENIGMA. ITP 2022

9. Mikoláš Janota, Jelle Piepenbrock, Bartosz Piotrowski: Towards Learning Quantifier Instantiation in SMT. SAT 2022

Acknowledgements

Many people have been part of my adventure these last few years. Here I want to express my gratitude to them.

First I would like to thank my supervisors during this PhD journey: Tom Heskes, Mikoláš Janota and Josef Urban. I was very lucky to get a group of supervisors with whom I got along on both a personal and research level. All of you have been inspiring for me. I would also like to say something about each of you separately.

I would like to thank Tom Heskes for his calm stewardship during my time as a PhD student: always approachable, even when in another country, for both advice and encouraging words. I always felt that I was in good hands.

I would like to thank Mikoláš Janota for his trust and willingness to integrate me into his project. I knew that I could always walk into his office and ask for technical explanations of complicated software or for an enthusiastic whiteboard session on concepts that were new to me.

I would like to thank Josef Urban for his limitless interest and motivation for advancing artificial intelligence in theorem proving. I learned a lot from him and I admire his ability to be an enthusiastic driving force in the community. He shaped, in Prague and through the AITP conference, a great environment for research.

I would also like to thank the people in and associated with the AI department at CIIRC and the broader community for making the research a lively activity: Thibault Gauthier, Martin Suda, Filip Bártek, Mirek Olšák, Lasse Blaauwbroek, Bartosz Piotrowski, Chad Brown, Karel Chvalovský, Jan Hůla, Jan Jakubův, Pedro Orvalho, Zar Goertzel, Michael Rawson, Jason Rute, Yutaka Nagashima, Konstantin Korovin, Cezary Kaliszyk, David Cerna and many others.

I would like to thank Herman Geuvers, Moa Johansson and Stephan Schulz for reviewing my manuscript and their useful suggestions on how to improve it.

I would like to thank Teven Le Scao, Barbora Hudcová, Hugo Cisneros and Kateryna Zorina for making my stay in Prague so good that I wanted to stay longer!

I would like to thank the people in and around the Data Science group at ICIS, where I felt at home: Charlotte Cambier van Nooten, Paulus Meessen, Emma Gerritse, Nik Vaessen, Marc Hermes, Gabriel Bucur, Parisa Naseri, Roel Bouman, and many others as well.

I would like to thank Alex Kolmus, Simon Brugman, Mick van Hulst and Roeland Wiersema for sharing with me very interesting and formative moments.

I would like to thank my mother Jacqueline and father Fred for all the support along the way! They gave me the freedom and opportunity to pursue my interests. With their encouragement, they laid the foundation on which I could build. I would also like to thank my sister Diede for her enthusiasm and support, even if sometimes she was half a world away. I know that I can always count on her.

And, of course, I also want to thank Mirthe for everything: you were there for all the highs and the lows. The knowledge that I could always return home to you made the whole process a lot more enjoyable.

Curriculum vitae

Education

- **PhD in Computer Science**, Radboud University 2020 – 2024
- **MSc. in Computing Science**, Radboud University 2018 – 2020
- **MSc. in Molecular Life Sciences**, Radboud University 2016 – 2020
- **BSc. in Molecular Life Sciences**, Radboud University 2012 – 2015

Employment

- **Promovendus**, Radboud University 2020 – 2024
- **Junior Research Scientist**, Czech Institute for Informatics, Robotics and Cybernetics, Czech Technical University in Prague 2020 – 2023
- **Co-Founder**, GraphKite 2017 – 2021

Radboud Universiteit