APPLICATIONS OF MACHINE LEARNING TO AUTOMATED REASONING

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2021

Michael Rawson

Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

APPLICATIONS OF MACHINE LEARNING TO AUTOMATED REASONING
Michael Rawson
A thesis submitted to The University of Manchester
for the degree of Doctor of Philosophy, 2021

Systems that can automate some aspects of logical reasoning are now very strong, through years of theoretical and practical development. Some progress has been made to have such systems learn from past experience. One extremely general and theoretically-favourable approach is to integrate learned guidance inside a system, biasing its internal search routines to explore promising areas earlier. Unfortunately, the approach has some barriers to practicality, notably the penalty on raw, inferences-per-second performance. We explore a number of approaches around this area designed to circumvent these barriers while making other trade-offs. The setting considered is automated theorem provers for first-order logic, but work is broadly applicable outside this area.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

Many people in the automated reasoning world have helped me along enormously. In no particular order and with apologies to any omissions: Martin Riener, Ahmed Bhayat and Edvard Holden, for many interesting conversations; Chad E. Brown, who took the time to explain some higher-order topics to me and bounce some ideas around; Stephan Schulz, friendly, encouraging, and a source of good ideas explained well; Michael Färber for correspondence on topics of interest; Sean Holden, who is someone I encounter rarely but somehow always leave with something new; and Geoff Sutcliffe for "Michael, why haven't I received any of *your* systems yet?". Particular thanks are due to Martin Suda and the group he belongs to, who graciously provided a comfortable, interesting stay in their beautiful city, as well as many things to think about.

Giles Reger is an excellent supervisor, handling the difficulties I repeatedly get myself into with consummate ease. He patiently provides help and advice on everything I could possibly need, all without *instructing* me to do anything, not once. I have greatly enjoyed three-and-a-bit years of support and intellectual freedom under his supervision, and I look forward to more as his colleague.

Finally, Michaela. Somehow she *still* thinks what I do is worthwhile.

Chapter 1

Introduction

This work considers automated reasoning systems that learn from past experience to solve harder problems, in much the same way as human mathematicians. My approach inhabits a narrow intersection of two much-larger worlds, reasoning and learning, typically realised in computer science as *automated reasoning* and *machine learning*. Both are united in their ultimate goal of the *thinking machine*, but differ in many other respects, exist largely independently of each other, and resist combination.

I arrive at this area from interactive theorem proving. For my undergraduate degree, I made computer-checked arguments about some properties of fiddly mathematical objects[1]. The argument was written in *Isabelle*, a complete software environment for writing computer-checked formal proofs. As well as many other features, Isabelle exposes a few forms of automated reasoning to help dispatch easier steps, so that the user does not have to spell out each proof in minute detail.

There were many repetitive proofs, differing in important details but very similar in spirit and technique. I was repeatedly surprised by automated reasoning failing to find proofs for goals very similar to those it had managed to prove before. I was less surprised that I was unable to "teach" the system to prove statements by writing the proofs myself — after all, this was not a feature Isabelle advertised — but nonetheless it seemed that it ought to be possible. In this way I became interested in improving automated theorem provers, specifically to have fully-automatic systems learn from past successes and failures.

[1] *nominal binders*, for the initiated

1.1 Theorem Proving meets Machine Learning

Automated theorem proving is one of the oldest problems in computer science. The challenge is simple to state, but hard to pin down and harder still to solve: *given some precise conjecture, automatically write a proof, or refute the conjecture*. After some early success, logical reasoning, and by extension theorem proving, became a central pillar of the symbolic approach to artificial intelligence popular from the mid-1950s. The symbolic approach ultimately struggled to fulfil its promise and floundered, at least in part due to difficulty of theorem proving and the problems of rapidly-growing search spaces in general[2], but the theorem proving problem and the community around it survived the ensuing "AI winters" through the following decades. Theorem proving remains relevant today, with a growing number of successes and applications.

However, the spectre of large search spaces continues to haunt the field. It can be helpful to delineate two general approaches that aim to make progress towards proofs despite this fundamental difficulty. The first, which I will call *redundancy elimination*, can remove large swathes of search space altogether by identifying equivalent or unnecessary work. The second, *heuristic search*, identifies some desirable direction and explores there by preference, thereby avoiding exploring undesirable space. Inevitably, things are not as simple: most practical techniques incorporate some amount of both.

Clearly the former is preferable: there is no need for good heuristics to avoid useless search if redundancy elimination procedures already removed it, and designing good heuristics is very difficult in general, to the point where even successful heuristics are almost never a monotonic improvement. Eventually, unfortunately, the free lunch of redundancy elimination begins to run out, and the computational cost of identifying redundancies progressively eclipses the benefit of removing them. Somehow we must obtain good, general-purpose heuristics that point the way *intuitively*.

Machine learning aims to have computer systems "learn from experience", typically with the aim of automatically improving system performance on some task. It is often characterised as disjoint from symbolic approaches, taking a more numeric, statistical approach and occasionally shunning the "artificial intelligence" label altogether. The field has recently attracted a lot of attention, often solving problems that

[2]In the words of the Lighthill report, which *de facto* defunded artificial intelligence research in the UK, "[first-order automated theorem proving] is particularly an area where hopes have been disappointed through the power of the combinatorial explosion in rapidly cancelling out any advantages from increase in computer power" [Lig73].

would be very awkward otherwise, e.g. "sort these cat images from those dog images". Could a statistical approach plug the gap in automated reasoning? Perhaps what is missing from existing reasoning systems is the instinctive, fly-by-the-seat-of-your-pants *gut feeling* of heading in the right direction, the same feeling that allows humans to tell cats from dogs and write proofs without exhaustive search.

It is also becoming increasingly apparent that the new statistical approaches have their own shortcomings, and will need combining with other methods to reach their full potential. With these new tools, but also a growing awareness of their limitations, it seems that automated reasoning and machine learning are overdue for a merger.

1.2 Key Challenges

Naturally, it is not straightforward to achieve good results by applying machine learning to automated theorem proving. A small number of familiar theoretical and practical issues repeatedly hinder work in this area. Sometimes they can be subtle, manifesting only in a disappointing result or technical limitation in otherwise-promising work. I therefore hold the view that at least some, and preferably all, of the following must be solved before reaching a true "breakthrough moment" for learning-assisted reasoning.

1.2.1 Objective Functions and System Performance

There is an unclear relationship between the training procedure, the objective function optimised for during training, and performance of the assisted system. It is a difficult problem to decide how to improve decisions made within a theorem-proving system by learning, and conventional training procedures cannot always reflect this difficulty. Sometimes, a learned heuristic that matches the objective function well during training has little positive or even a negative effect on system performance.

For example, the problem of "premise selection" — selecting relevant facts from a large database for a certain conjecture — is, at first glance, a simple classification problem. But how best to emphasise that it is much more important that relevant facts are retained than irrelevant facts discarded? How best to deal with multiple proofs of the same conjecture that use different facts? If the learned model can indicate uncertainty in some way, how best to use this new information in the system?

To make matters worse, the current generation of automated reasoning systems are not in general good at recovering from bad states; as such, guided systems can inhabit an environment where excellent guidance yields merely good results, but even slightly-poor guidance is catastrophic.

1.2.2 Representing and Processing Syntactic Data

In order to use machine learning techniques to learn some desired function from inputs to outputs, at least the inputs (and sometimes also the outputs) must be represented in a suitable way for processing. The trend in machine learning is to move from hand-engineered "features" to learning more-or-less-directly from raw data: text, images and so on. This shift also applies to logical settings, but these new machine learning techniques have not always applied to, or worked well with, syntactic data. There are two axes on which representations are often compared: the amount of information that is lost when translating from one form to another, and the relative difficulty of learning functions from the representation.

Textual serialisation is an example of a representation which scores well on the former, but less well on the latter: the original can be read off directly, but it is quite difficult (informally speaking) to learn anything from a sequence of characters. On the other hand, feature-based representations score the opposite: we typically have no idea from a feature vector what the original might have looked like, but the vector is relatively easy to learn a function from — but only if we chose good features! Finding a representation technique that scores well on both scales is critical for effective learning.

1.2.3 Deceleration of Inference Rate

Integrating heuristic guidance into the internals of theorem-proving systems, especially neural networks, incurs a performance penalty. Systems often rely on a high rate of inference to find proofs within reasonable time. Unfortunately, this can mean that the performance penalty dominates any performance gained by means of learned guidance. This is especially pernicious if the system requires sequential execution of inference steps without backtracking, as a decision must be taken immediately and cannot be returned to. Recent software libraries typically use a coprocessor such as a commodity GPU to accelerate evaluation of large learned models, and therefore a distinction must be drawn between coprocessor *latency* and the total amount of compute required.

1.2.4 Proving Power versus System Suitability

It seems that some theorem-proving systems are more suitable than others for integration of learned guidance within the system itself, and unfortunately those that are more suitable tend to be less powerful unguided systems. It is possible to conceive learning systems that are agnostic to the system which consumes their output (such as premise selection), but these are typically limited in their decision-making power.

As mentioned in §1.2.3, performance penalties introduced by learned guidance can affect some systems more than others, but this is not the only distinction. Some redundancy eliminations and/or heuristics are effectively search *decisions* that increase system performance in the aggregate, but in specific cases can be harmful[3]. Removing these features allows learned guidance to make the correct decision for a particular problem, but also significantly decreases the proving power available for data generation during training, or to make up for deficiencies in learned guidance. Some way of resolving this conflict may be essential to future work in this area, so that systems can use learned guidance to make decisions while still exploring a reduced search space.

1.3 Approach

Strategy scheduling is a known area for combining machine learning and automated reasoning in which a list of system configurations (so-called “strategies”) are selected and/or prioritised by a machine-learned heuristic based on a set of features. Existing work typically only used “static” features derived from the problem, not the system’s *response*: this led me to investigate “dynamic”, search-time features for this problem, described in Chapter 3. This approach neatly avoids many pitfalls (although it arguably falls prey to a particularly-nasty case of the “objectional objective function”, §1.2.1), but it is limited by the lack of decision-making power common to strategy-based approaches: if a problem cannot be solved by any configuration in reasonable time, even perfect learned heuristics cannot help.

This frustration encouraged work on the system described in Chapter 4, in which a more semantic, intrusive approach is taken to intelligently guide preprocessing of a problem into sub-goals for dispatch by an existing automated theorem prover. The setting clearly highlighted all the challenges described in 1.2, and provided a starting point to address these directly. Representations of formulae used for learning (§1.2.2)

[3]For intuition’s sake, consider symbol precedence in a saturation context, or the regularity condition for connection tableau — few would suggest building a new system without these techniques today.

appeared promising and were developed further in Chapter 5, validated by empirical studies on benchmark tasks, and then used in subsequent chapters. Upon closer inspection, Chapter 4 began to look like a reinforcement-learning problem, where the reward received correlates *directly* to system performance (§1.2.1). I explore this idea in Chapter 6, where a theorem-proving reinforcement environment with graduated reward is constructed, and off-the-shelf algorithms for reinforcement learning are applied.

The practical problem of evaluation latency (§1.2.3) proved to be by far the most difficult to solve convincingly, but one possible solution is presented in Chapter 7. A backtracking system is presented which can evaluate learned heuristics asynchronously without reducing inference rate. Work on simplified ways to bias tree search (§1.2.1) and improve the performance of backtracking systems (§1.2.4) are also presented here.

1.4 Thesis Structure

Work is laid out in chapters, as follows.

1. Introduction
2. Background Material
3. Dynamic Strategy Priority (prioritising strategies during proof search)
4. Guided Preprocessing (intelligently preprocessing and splitting a problem)
5. Directed Graph Networks (representing and learning from syntactic data)
6. Reinforced Theorem Proving (a black-box reinforcement learning environment)
7. Asynchronous Policy Evaluation (integrated guidance without performance penalty)
8. Concluding Remarks

1.5 Pertinent Contributions

Here I list research contributions relevant to this work. Sometimes, early versions of the following appeared at events such as the *Conference on Artificial Intelligence and Theorem Proving* or the *Automated Reasoning Workshop*.

- *Dynamic Strategy Priority: Empower the strong and abandon the weak* [RR18]. Here we introduce the idea of prioritising strategies during proof search based on run-time features.

- *A Neurally-Guided, Parallel Theorem Prover* [RR19a], in which we present ideas and a prototype system addressing many of the issues raised in §1.2.

- *Directed Graph Networks for Logical Reasoning* [RR20b] provides simple solutions to the problem of representing syntactic data, such as those found in reasoning systems.

- *Reinforced External Guidance for Theorem Provers* [RBR20] constructs a reinforcement learning task to supply an existing unmodified system with helpful facts, and investigates solving the task.

- *lazyCoP: Lazy Paramodulation meets Neurally Guided Search* [RR21][4] in which we describe a system that uses expensive learned guidance to guide internal search routines without deceleration of inference rate.

[4]to appear

1.6 Other Contributions

I have made other tangential contributions that are not directly relevant to this work. Work on "age/weight shapes" in Vampire [RR19b] does not rely on learned guidance, although it does present encouraging evidence for the impact internal guidance could have on a state-of-the-art system. The ability to auto-encode sets of problems such that each problem can be mapped into a fixed-dimensional space is a corollary of the approach take in Chapter 5 and has some interesting applications [RR20a]. An as-yet-unpublished collaboration entitled "What can we learn from TSTP proofs?" mined information from a large set of proofs using both symbolic and statistical methods.

I am the author of lazyCoP, a connection-tableau system for first-order logic with equality, the first practical implementation of the calculus LPCT, and the research vehicle for Chapter 7. I also maintain a high-performance parser library for the TPTP problem format[5], with active users in the research community.

[5]https://github.com/MichaelRawson/tptp

Chapter 2

Background Material

Voluminous background literature could be presented as relevant to this work. This is due in part to the cross-disciplinary nature of the topic, and in part because of the long and successful history of both automated reasoning and machine learning as distinct fields. The chapter is left intentionally brief, and each subsequent chapter introduces specific literature as required. Each area of study is presented in isolation, then existing steps to apply machine learning to automated reasoning are discussed.

Another challenge to concise presentation of background material is recent popular and scientific interest in "artificial intelligence", particularly in the area of deep neural networks and their applications. As well as pure volume, research material is frequently not peer-reviewed, reproduced or even plausible; material presented in this chapter is either published or widely relied upon by others in the community.

In general, research progress in this area comprises combinations of existing well-known ideas from both fields to solve concrete problems, or viewing a known idea through the lens of the other field; rarely are completely-new approaches required or developed. This is encouraging for future work: each field is well-developed alone, but the combination is relatively unexplored.

2.1 Automated Reasoning

Automated reasoning is a broad area of study within artificial intelligence (see e.g. the *Handbook of Automated Reasoning* [RV01]), but what follows is framed in terms of *automatic theorem proving* in the symbolic tradition, specifically for classical first-order logic. Other automated reasoning settings (e.g. interactive theorem proving, counterexample finding, question-answering or program synthesis) are not considered, nor is automatic theorem proving for logics differing significantly from first-order logic (e.g. description, modal, intuitionistic, or higher-order logics) or special-purpose reasoning, such as methods for automatic reasoning in planar geometry. However, work presented in following chapters is somewhat generic over logic and problem setting.

2.1.1 Classical First-Order Logic

First-order logic (see e.g. *First-Order Logic* [Smu95]) provides syntax and semantics for statements about individuals, functions and predicates over individuals, propositional connectives and quantifiers $\forall, \exists$ over individuals. A binary equality predicate $s = t$ between terms can be included directly or axiomatised within the logic itself. *Classical* first-order logic admits the so-called "law of the excluded middle" asserting that a proposition must be either true or false.

Classical first-order logic is a popular logic for a variety of applications: it is sufficiently expressive for "ordinary mathematics", but not so expressive as to be completely intractable; it compactly encodes facts from a variety of domains [SSY94]; some other logics can be embedded in, or translated to, first-order logic; "theories" such as integer arithmetic can be included somewhat orthogonally [KV13]. This logic in particular also has a variety of theoretical niceties which aid both automation and expressivity: it is syntactically consistent, semantically complete, closed under negation, and admits powerful normal forms and cut-elimination. Lindström's theorem shows roughly that this is the strongest logic that retains some pleasant properties [Lin69].

2.1.2 Automatic Theorem Proving for First-Order Logic

Assuming a complete inference system, proofs of true statements in first-order logic system are known to exist and be of finite length. This property suggests *automatic theorem provers* (ATPs): software systems that explore a search space induced by an inference system to find a proof of some proposition. Unfortunately, proof *search* in first-order logic is known to be semidecidable in theory and computationally difficult in practice — but this has not stymied the efforts of authors of such systems!

Automatic reasoning in this setting has a long history [Dav01], with a number of success stories. A focus of research throughout this time has been to achieve reasoning *up to redundancy*: normal forms for formulae [BEL01] avoid duplicative reasoning up to isomorphism in the normal form; the resolution method [Rob65] avoids generating elements of the Herbrand universe unless required; the paramodulation and later superposition calculi [NR01] reduce the space of possible equality reasoning steps, to name a few. These techniques can reduce the size of the search space that must be explored and therefore accelerate finding proofs, bringing more difficult problems within reach.

Of course, other topics of interest are numerous: developing efficient proof calculi and search algorithms based on these techniques (such as the saturation and tableau families, below); reasoning over large axiom sets (e.g. [HV11]); the (partial) translation of problems expressed in other systems to first-order logic (e.g. [MP08, Urb06]); the integration of external reasoning tools such as SAT/SMT solvers (e.g. [Vor14]); efficient implementation techniques such as term indexing; decidable classes and incomplete algorithms; and meta-techniques such as strategy scheduling [WL99].

2.1.3 Benchmarks

Benchmark problem sets and competitions are a surprisingly-powerful force in driving research on automated theorem provers. Thousands of Problems for Theorem Provers (TPTP) [SSY94] is a curated set of problems from a variety of domains, expressed in a variety of logics and styles, which drives the annual CASC competition [Sut16]. MPTP [Urb06], on the other hand, is a translation of theorems from the Mizar Mathematical Library [GKN10] into first-order logic with equality. We sometimes use the *M40k* and *M2k* sets for evaluation, as in [KUMO18]. There are many other such benchmarks for different purposes, such as the SMT-LIB library [BST$^+$10] of Satisfiability Modulo Theories problems.

2.1.4 Superposition and Saturation

Many of the most high-profile modern systems are (partially) based on the superposition calculus, inferences from which are explored via *saturation* proof search. While the two are not inextricably linked, it is certainly a natural and popular pairing. Saturation algorithms aim to produce a *saturated set* of formulae: a set such that all inferences from the set up to redundancy are contained within the set [BG01a].

A number of different algorithms achieve saturation: *given-clause* algorithms are a popular clause-level approach in which a "given clause" is selected in some way and added to an initially-empty "processed" set by performing all necessary *generating* inferences with the existing members of the processed set [KV13]. There are various ways of further categorising not-yet-processed clauses; here it suffices to mention that this group of clauses grows rapidly as the processed set grows. *Simplifying* and *deleting* inferences may also simplify/delete available clauses at any point during search.

This framework encourages powerful redundancy elimination techniques such as *subsumption*, as well as inherent advantages compared to other methods: never exploring any part of the search space twice, for one. In exchange, memory use of saturation systems can be significant, although when combined with redundancy eliminations and modern hardware this is less of a problem than previously. Further, typical realisations of saturation such as given-clause algorithms enforce a temporal linearisation of inferences. This linearisation is restrictive for some developments, such as parallelisation at proof search level or learned heuristic guidance.

2.1.5 Tableau and Connection Methods

Tableau-style systems [Häh01] for first-order logic are often framed as distinct from or even in opposition to saturation/superposition systems, although nothing in principle prevents a tableau system from employing superposition[1]- or saturation- style reasoning [DV98, Gie06]. These systems aim to build closed *tableaux*, a kind of rooted tree presenting an argument by contradiction. Branches represent case distinction, and a branch is *closed* if a contradiction is obtained with respect to facts obtained earlier on the branch or its ancestors.

Many variants and refinements of tableau calculi exist. *Connection* tableau [LS01] are a particularly strong refinement: these add the constraint that extensions to a tableau must be directly *connected* to the current branch's leaf literal, rather than allowing the addition of axioms unrestricted. This allows an idiosyncratic goal-directed proof search, starting at the (negated) conjecture and working back toward axioms which refute it. Tableau with the connection refinement is not a proof-confluent system, but this is not a fatal problem as search in connection-tableau systems is typically backtracking in nature, such as by iterative deepening.

The current generation of (connection) tableau systems are typically less strong than current superposition/saturation systems, at least as measured by performance on large benchmark datasets such as TPTP. Explanations for this performance gap might include the difficulty of global search refinements such as subsumption, the

[1]although it should be noted that there are some problems with equality reasoning and tableaux, notably undecidability of rigid E-unification [DV96] and incompleteness of the obvious formulation of paramodulation with respect to the connection refinement [Pas08]

lack of specialised equality handling, the lack of proof confluence, or simple under-development relative to state-of-the-art systems. However, such systems have some compelling advantages which suggest unrealised potential, at least in combination with other systems. Proof search is goal-directed, adapts to at least modal and intuitionistic logics naturally [Waa01], and does not require sequential inference.

2.1.6 System Behaviour and Folklore

Significant "folklore" knowledge has accumulated about the design, implementation, behaviour and evaluation of first-order theorem provers, especially those of the saturation flavour. Some are "obvious", such as the idea that saturation systems typically find a proof quickly, or not at all — this follows from the quickly-growing sets of clauses present in saturation systems. Others are evaluated experimentally in literature, such as in experiments on clause selection [SM16, RR19b], although given the artificial nature of such experiments it is unclear how conclusive they can be. In following work folklore knowledge is acknowledged where used as such.

2.1.7 Modern Systems

There are a large number of systems capable of some amount of first-order reasoning currently available, all with different aims and advantages. We refer to, and occasionally use, a number of systems throughout what follows, either as motivation or as part of an experiment. In no particular order, and omitting many interesting and powerful systems, we mention here the classical superposition/saturation systems Vampire [KV13] and E [Sch02], the instantiation-based iProver [Kor08], the SMT solvers Z3 [DMB08] and CVC4 [DRK$^+$14], the model-elimination system SETHEO [LSBB92] and the "lean" connection-tableau system leanCoP [OB03].

2.2 Machine Learning

Machine learning is another branch of artificial intelligence which aims to have algorithms "learn from experience". This work uses techniques and results from the statistical tradition (see e.g. [Bis06, Mur12]) of machine learning, although other approaches exist. Traditionally the field is split into supervised learning, unsupervised learning, and reinforcement learning, although this distinction is not clear and there is considerable overlap between these areas.

Many different statistical techniques solve different problems with differing levels of complexity. Neural network models (see e.g. [GBC16]) in particular are the tool used in this work, and there are good reasons for this: neural networks are sufficiently general-purpose that they form a lowest common denominator; adapted models can process many different types of data; software libraries and hardware acceleration (e.g. [PGM$^+$19, ABC$^+$16, NBGS08]) exist to reduce development and training time; and implementing the forward pass of a network manually for integration into other systems is usually trivial.

However, neural models are by no means uniformly better than other techniques (and the current research focus on "deep learning" has its critics [Mar18]), so the decision to focus on applications of neural networks for theorem proving rather than exploring the large body of material on alternative statistical models is a source of some regret. Approximate Bayesian inference methods [BKM17] which allow the expression of uncertainty are of particular future interest.

2.2.1 Deep Neural Networks

An area of development within neural-network models concerns "deep learning": constructing neural networks from many processing layers, so that it is possible to learn hierarchical "features" [LBH15]. As input data passes through the network, features identified in lower levels are combined a new set of features for processing later. An interesting aspect of deep learning is the use of non-traditional layers and dynamic network topologies: examples include convolutional, recurrent and recursive networks [GBC16], and attention methods [VSP$^+$17].

Training such networks can require some artifice to avoid undesirable outcomes such as divergence, slow training, or overfitting. Many techniques have been developed which may help with training in one way or another, but it is not clear how best to categorise the resulting "box of tricks" according to how or what they achieve. Typical practical approaches may include some of the following non-exhaustive list: various non-linearities used between layers (including the popular rectified linear unit [NH10]); initialisation schemes (e.g. [GB10]); "regularisation" techniques such as norm penalties, parameter sharing (particularly as in convolutional and recurrent networks), data augmentation, label smoothing, early stopping, or dropout; stochastic gradient descent; purported improvements to the standard gradient descent algorithm such as gradient clipping, (Nesterov) momentum, adaptive learning rates (e.g. [DHS11, KB14]), learning rate schedules, or cyclic learning rates [Smi17, ST19]; batch normalisation [IS15] and related methods; performance optimisation methods which change behaviour such as integer quantisation or HOGWILD! [RRWN11]; a variety of possible loss functions or combinations thereof; and meta-techniques such as ensemble methods or transfer learning.

Changing the neural network's overall architecture is a recent successful trend, particularly to train deeper networks. Architectural styles such as Inception [SLJ$^+$15] (smaller sub-networks inside a larger network), Highway Networks [SGS15] (gated information flow), residual networks [HZRS16] (addition of inputs to outputs) and densely-connected networks [HLMW17] (all previous outputs available as inputs) tend toward "shortcut" connections for faster training of very deep networks.

Given the huge number of possible architectures and techniques, practitioners tend not to exhaustively attempt all combinations of these, instead relying on accumulated experience on similar learning tasks. The same is true of the many hyperparameters accumulated during the design of the training program. Some research aims to also automate these manual/grid-search tasks [BB12, EMH18].

2.2.2 Graph Convolutional Networks

Processing graphs in a neural context is of interest for this work. Graph convolutional networks have become a popular technique in this area, although there has been previous work on learning with graphs [ZTXM19]. Non-graphical convolutional networks are well-known for e.g. image processing tasks, but the graphical analogue is a more recent development. The original Graph Convolutional Network's learned convolution operator could be loosely described as "take the mean of the current node and its neighbours, transform the result by some learned weight matrix and activate" [KW16a].

Later developments have included more-powerful operators capable of better distinguishing certain graph structures (e.g. [XHLJ18]), novel graph network architectures (e.g. [KW16b, GJ19]) and improved techniques for localised pooling (e.g. [Die19]).

2.3 Heuristic Search, Planning and Reinforcement

These areas are grouped together here because all three consider the problem of exploring state spaces, albeit with different perspectives and varying amounts of information and freedom. The following work only uses a few results and techniques from these areas, but they are indispensable and disproportionately effective for theorem proving.

Heuristic search [ES11] is a cornerstone of "good old-fashioned AI" (e.g. [RN02]), providing a variety of techniques for exploring state space in various settings, such as *A* search* for informed search or *minimax* and variations for gameplay. *Planning* [LaV06], on the other hand, focusses on producing a plan — that is, actions leading from a start state to a goal state — for an agent. These first two are particularly difficult to disentangle [BG01b].

Reinforcement learning provides algorithms for maximising long-term *reward* received by an *agent* interacting with an *environment* (e.g. [SB18]). Many reinforcement algorithms call for some kind of function approximator, which in the modern age is typically a (deep) neural network. Some surprising results can be achieved via such "deep reinforcement learning" (e.g. [MKS$^+$15]), but the sub-field has suffered from some initial problems [Irp18]. There are several ways to classify work in reinforcement learning, to highlight a few: algorithms may assume the existence of a (learned) model of the environment ("model-based"), or not ("model-free") [AS97]; environments may have discrete or continuous state/action spaces; agent learning may be carried out during interaction with the environment ("online") or on logged interactions ("offline") [ASN19]; and environments may be deterministic or stochastic. Theorem proving recast as a reinforcement learning environment usually inhabits a discrete, deterministic environment with access to an obvious model, namely inference.

2.3.1 Bandits, UCT and MCTS

A fruitful line of questioning concerns so-called "bandit problems" [SB18]. The *multi-armed bandit* is a fixed number of machines, each of which has one lever (the "one-armed bandit"). Pulling a lever dispenses a reward, which is usually assumed to be an independent and identically-distributed sample from an unknown distribution, different between levers. Only pulling one lever at a time, how best to proceed?

It is possible ([ACBF02], based on earlier work [Agr95]) to achieve only logarithmic "regret" with a simple-to-compute algorithm UCB1: pull the lever i that maximises

$$\bar{x}_i + \sqrt{\frac{2 \ln n}{n_i}}$$

where $\bar{x}_i$ is the mean reward from lever i so far, n_i is the number of times i has been pulled, and n is the total number of lever pulls. Perhaps surprisingly, this technique can be adapted to search trees [KS06], which forms the basis for *Monte-Carlo Tree Search* [CBSS08], famously combined with neural networks to play Go [SHM$^+$16].

2.4 Machine Learning for Automated Reasoning

Integrating machine learning techniques is a natural research direction for automated reasoning: there has been much research on reducing redundancy in proof search, but comparatively little about choosing a good “direction” for proof search. Some manual developments could be viewed as symbolic approaches toward this goal: goal-directed methods such as SInE [HV11]; directed rewrite approaches such as ordered paramodulation; or heuristics such as clause weight; but as with all fixed heuristics their power and flexibility, while significant, are limited to what has been programmed.

Learned heuristics have no such limits, although their practical applications are only just emerging — the most popular guidance methods are still the manual, symbolic variety. As learning techniques increase in both power and capabilities, learned approaches may augment or even replace manual heuristics in existing systems [Goe20]. Unfortunately, some recurring issues have have historically kept the area little-known compared to its two enormous parents, as discussed in §1.2.

Nevertheless, intrepid research has achieved steady progress, with occasional high-profile success on large projects. Work published in 2014 combined classical theorem proving systems with learned premise-selection systems to prove 39% of theorems from the Flyspeck project [Hal06] completely automatically [KU14]. This was later improved to 47% by 2015 [KU15b]. More recent work saw the large-scale development of deep neural networks for premise selection [ISA$^+$16] and to guide existing systems [LISK17].

Systems continue to improve into the present. In the most recent CADE ATP System Competition (CASC) [Sut16], a competition for automatic theorem proving systems including various logics and divisions, systems making use of machine learning were in contention. The MaLARea system [Urb07], a meta-system again combining classical theorem proving techniques and learning techniques, took first place in the “Large Theory Batch” [Sut08] division. Meanwhile, ENIGMA [CJSU19], a learned guidance package for the mature, saturation-based E theorem prover [Sch02], outperformed the unguided host system in the “First-Order Formulae” division.

2.4.1 Features and Representations

Representing formulae for processing by machine-learning algorithms has been a research area for some time. The majority of classical algorithms take aim at fixed-size, typically real-valued vectors, and so effort has been made to both engineer good features (e.g. [BHP14]), or to embed formulae algorithmically into a feature vector (e.g. [JU17, TUGH11, GJU19]). The symbols that occur in a given input can be used in various ways to generate these vectors, but this can result in very high-dimensional feature vectors when many symbols are present in a problem set. This problem can be tackled using a distributed representation of features [KK18]. An interesting approach to augmenting feature engineering is the "abstract nonsense" of MaSh [KBKU13], which uses both manual features and the working hypothesis that "facts with similar features are likely to have similar proofs". The above techniques tend towards *syntactic* features, but *semantic* features are also possible [KUV15, USPV08]

Feature engineering in this way is very successful, but requires manual effort and necessarily discards at least some information from input data. End-to-end approaches (see [ATCF20] for an overview and experiments) have included character- or token-level approaches in which structured data is rendered to a textual representation (e.g. [ISA+16, PUBK19]), recursively-evaluated syntax-tree structures (e.g. [Chv18, Gau20]), recurrent methods [PU20], attention-based approaches [UJ20] and — recently and with apparent success — graphical approaches in which formulae are treated as directed syntax graphs, possibly sharing terms (e.g. [WTWD17, OKU19, PLR+20]).

2.4.2 Indirect Guidance

Learning approaches which do not modify the proof search procedure directly have several advantages: no performance penalties during proof search, no modification of complex search systems required, and often more clear-cut development and evaluation as a result of treating the underlying system as a black-box. A possible limitation of such approaches could be that no matter the performance of such systems, if the underlying system cannot prove the result it is of no use. However, empirically indirect guidance is very successful and there is limited data to support the idea that there are such problems which cannot be solved without learned intervention [RS19].

Premise selection (see e.g. [KLT+12] for an overview) is particularly prevalent, aiming to select only relevant facts and thereby reduce the search space. Strategy selection (e.g. [BHP14, HK19]) aims to select good options for a concrete system for a given problem, a technique known to be surprisingly effective from work on strategy scheduling. It is possible to conceive many different such problem settings, such as learning symbol precedences for term orderings [BS20] or conjecturing intermediate lemmas [UJ20].

2.4.3 Direct Guidance

The other major approach is the opposite: do not modify factors external to proof search, but instead guide search internally through algorithms parameterised by learning systems. An argument could be made that this approach *subsumes* indirect guidance: premise selection can be achieved simply by avoiding those premises, strategy selection is not required if the system *is* the strategy, conjecturing can be achieved directly by searching for intermediate lemmas, and so on. In any case, the direct guidance problem is very hard, and in practice the two approaches may be complementary.

This approach is not new: early work in this area includes perceptron guidance [ESS89, SE90] for branches in SETHEO [LSBB92]. Tableau-style systems have remained popular in such research, including naïve-Bayes guidance [UVŠ11, KU15a] for leanCoP [OB03] and later developments with reinforcement learning [KUMO18, ZCM+19]. Saturation-style provers initially resisted learned guidance, exhibiting performance problems [LISK17], but some work has gone into remedying this problem [CJSU19].

Chapter 3

Dynamic Strategy Priority

Material from the following chapter appeared in *Practical Aspects of Automated Reasoning 2018* under the title "Dynamic Strategy Priority: empower the strong and abandon the weak" [RR18]. The paper introduced a new method for guiding portfolio ATP systems by inspecting run-time portfolio information and demonstrated improved performance on a concrete ATP system, Vampire.

Automated theorem provers are often used in other tools as black-boxes for discharging proof obligations. One example where this has enjoyed a lot of success is within interactive theorem proving. A key usability factor in such settings is the time taken for automatic systems to complete. Automated theorem provers typically run lists of proof strategies sequentially, which can cause proofs to be found more slowly than necessary if the ordering is sub-optimal. We show that it is possible to predict strategies likely to succeed while they are running *using an artificial neural network. We implement a run-time strategy scheduler in the Vampire system which exploits the trained neural network to improve average proof search time, and hence increases usability as a black-box prover.*

Many modern automated theorem provers (e.g. E [Sch02], iProver [Kor08], Vampire [KV13], CVC4 [DRK$^+$14], leanCoP [OB03]) for first-order logic rely on *portfolio modes* [RSV14], which use tens to hundreds of distinct *strategies*. Of these strategies, only a few might solve a hard problem, often rapidly. Typically, a portfolio of strategies has a predefined execution order: the prover will execute strategies in this order, running each for a bounded length of time until a strategy succeeds or the system exhausts time or strategies.

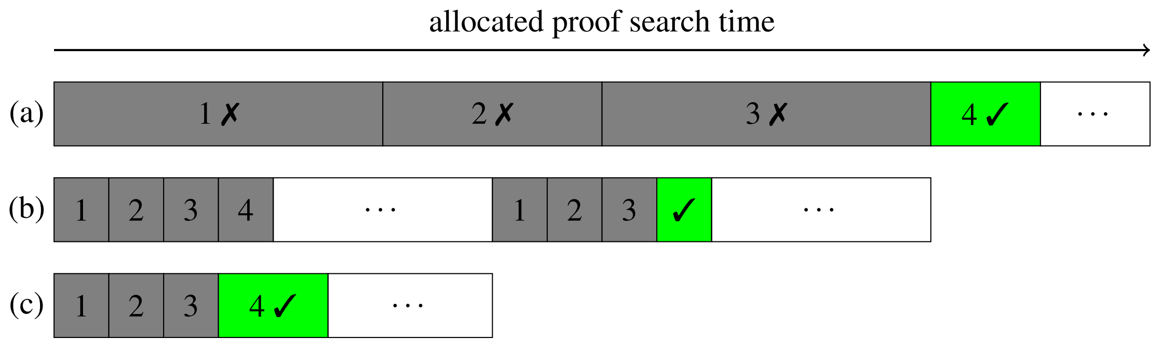


Figure 3.1: Different strategy scheduling schemes and their effects on proof search:
(a) Failing strategies might block a successful strategy.
(b) Round-robin scheduling can help mitigate this problem.
(c) This approach is improved if promising strategies run for longer.

Portfolio modes are important as, in practice, there is no best strategy. Furthermore it is uncommon that two hard problems are efficiently solved by the same strategy. However, portfolio execution is not without problems: deciding the optimal ordering and time allocation is hard in general [RS17], and produces overly-rigid, brittle engineering when applied to specific domains, such as those found in the TPTP problem set. Moreover, for any particular problem, some lengthy strategies that are successful on other problems are doomed to failure from the outset — illustrated in Figure 3.1 — but are left to run unchecked, wasting time that could be spent on other strategies.

This chapter makes two contributions. We first demonstrate correlation between trends in dynamic properties of proof search, and the pending success or failure of a strategy (§3.2,§3.3). We then use this to implement strategy scheduling, prioritising those strategies most likely to succeed (§3.4). This approach differs from previous work [Sch02, MW97, KSU13] which attempts to predict successful strategies *a priori* from static features of the input problem; instead we tip running strategies for success based on dynamic, run-time features and use this information to make decisions at runtime. Our experiments (§3.5) show that guiding scheduling in this way can significantly speed up portfolio-based approaches.

3.1 A Brief Introduction to Vampire

Vampire is a saturation-based first-order automatic theorem prover. This section reviews its basic structure and components relevant to the rest of this chapter. We will use the word *strategy* to refer to a set of configuration parameter values that control proof search, and *proof attempt* to refer to a run of the system using such a strategy.

3.1.1 Input and Preprocessing

Vampire accepts problems in classical first-order logic with equality and a predefined set of first-order *theories*: arithmetic, arrays, datatypes and so forth. Problems are parsed and transformed into a set of *input clauses* before proof search begins, referred to as *preprocessing*.

This process converts the problem to *clause normal form*, and may also apply a number of other possible preprocessing techniques. Preprocessing can alter certain properties of the problem such as its size and distribution across the signature of the problem. For example, the E.T. system [KSUV15] implements a preprocessing step for E which selects small sets of axioms from a larger set, for reasoning in large theories. This suggests that any prediction method that relies solely on characteristics of the input problem, rather than search-level characteristics such as post-preprocessing properties, will find it harder to predict the success or failure of strategies.

3.1.2 Saturation Algorithms

Saturation systems such as Vampire use *saturation algorithms with redundancy elimination* (§2.1.4). They work with a search space consisting of a set of clauses and use a collection of generating, simplifying and deleting inferences to explore this space. Generating inferences, such as *superposition*, extend this search space by adding new clauses obtained by applying inferences to existing clauses. Simplifying inferences, such as *demodulation*, replace a clause by a simpler one. Deleting inferences, such as *subsumption*, delete a clause, typically when it becomes redundant [BG01a]. Simplifying and deleting inferences must be well-behaved to preserve completeness.

The goal of saturation algorithms is to *saturate* the clause set with respect to the inference system. If the empty clause is derived then the input clauses are unsatisfiable. If no empty clause is derived and the search space is saturated then the input clauses are guaranteed to be satisfiable, but *only if* a complete strategy is used. A strategy is complete if it is guaranteed that all inferences between non-deleted clauses in the search space will be applied. Nevertheless, Vampire includes many incomplete strategies, as they can be more efficient at detecting unsatisfiability.

All saturation algorithms implemented in Vampire belong to the family of *given* *clause algorithms*, which can achieve completeness via a fair *clause selection* process that prevents the indefinite skipping of old clauses. These algorithms typically divide clauses into three sets: *unprocessed*, *passive* and *active*. Clauses are considered *processed* if they are contained within *passive* or *active*. Algorithms follow a simple *saturation loop*:

1. Add non-redundant *unprocessed* clauses to *passive*. Redundancy is checked by attempting to *forward simplify* the new clause using processed clauses.

2. Remove processed clauses made redundant by newly processed clauses, i.e. *backward simplify* existing clauses using these clauses.

3. Select a given clause from *passive*, move it to *active* and perform all generating inferences between the given clause and all other active clauses, adding generated clauses to *unprocessed*.

Later we will show how iterations of this saturation loop from different proof attempts can be interleaved. Vampire implements three saturation algorithms:

1. *Otter* uses both passive and active clauses for simplifications.

2. *Limited Resource Strategy (LRS)* [RV03] extends Otter with a heuristic that discards clauses that are unlikely to be used with the current resources, i.e. time and memory. This strategy is incomplete but also generally the most effective at proving unsatisfiability.

3. DISCOUNT uses only active clauses for simplifications.

A recent development in Vampire is AVATAR [Vor14, RSV15] which integrates with the saturation loop to perform clause splitting. The general idea is to use a SAT solver to select a subproblem by naming each clause sub-component by a propositional variable, running a SAT solver on these abstracted clauses, and using the subsequent propositional model to select components to include in proof search. At the end of each iteration of the loop we check whether the underlying subproblem has changed. AVATAR can occasionally make loops run a little longer, but no more than other steps such as backward subsumption. Otherwise, the notion of saturation loop remains the same when using AVATAR.

There are also other proof strategies that fit into the above loop format and can be interleaved with the core superposition-based proof attempts. For example, *instance* *generation* [GK03] saturates the set of clauses with respect to the instance generation rule and *finite model finding* [RSV16] iteratively checks larger model sizes. These loops tend to be much longer than those from other algorithms.

3.1.3 Strategies in Vampire

Vampire includes more than 50 parameters. By only varying parameters and values used by Vampire at the last CASC competition, we obtain over 500 million strategies. These parameters control aspects of proof search such as

- Preprocessing steps (24 different parameters)
- Choice of saturation algorithm and related behaviour, such as clause selection
- Inferences available (16 different kinds with variations)
- Splitting via AVATAR

Even restricting these parameters to a single saturation algorithm and straightforward preprocessing steps, the number of possible strategies is vast. For this reason, Vampire implements a portfolio *CASC mode* [KV13] that categorises problems based on syntactic features and attempts a sequence of approximately 50 strategies over a five minute period (this number can vary significantly). These strategies are the result of extensive benchmarking and have been shown experimentally to work well on unseen problems, i.e. those not used for training.

The syntactic features used by the current portfolio mode are coarse-grained and include the rough size of the problem (bucketed into tiny, small, medium, large, huge), the presence of features such as equality or certain theories, and whether the problem is effectively propositional [PdMB08] or Horn [Hor51]. Not all combinations of these are considered. Portfolio mode is created by considering a set of training data over a list of strategies and attempting to greedily cover as much of it as possible by splitting the set of problems into smaller groups based on these features until all solutions fit into a given time limit. This process places the strategy that solves the most problems during training, then the next best strategy after removing the problems solved by the first, and so on.

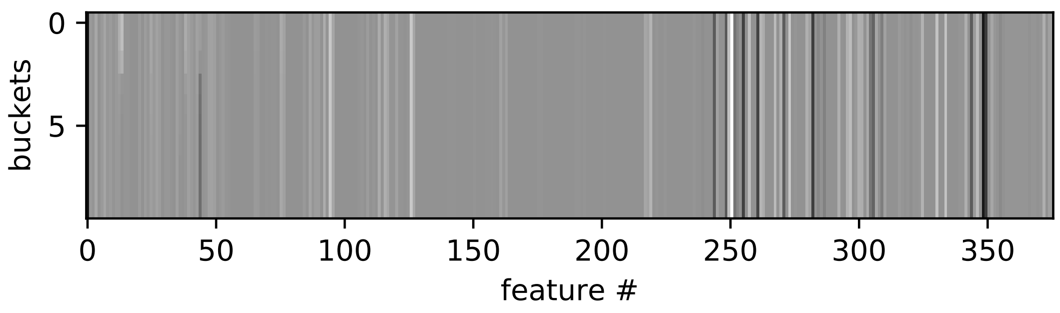


Figure 3.2: A trace displayed as a colourmap, after preprocessing. The vertical axis represents (normalised) inferences, horizontal axis features in no particular order. While most features remain almost the same over the lifetime of this trace, some can be seen to change on the left-hand side of the map.

3.2 Feature Engineering and Collection

This section discusses features extracted from Vampire for prediction and how they are extracted. Modifying Vampire to log execution data (general metrics such as memory usage, or prover-specific metrics such as the number of generated clauses) for different strategies obtained from its primary portfolio mode is straightforward, but some data-collection decisions were made:

- Only numerical data immediately available in the prover are collected, but there is scope here for both non-numeric and derivative data sources, which may provide greater insight into the proof state in future work. Suppose that quantities A and B are measured directly, but are more often discussed in terms of categories C that all A,B pairs fall into. Data could then include a suitable embedding of C as well as, or instead of, A and B.

- Data are collected at intervals of a fixed number of internal saturation steps (in experiments presented in this chapter, this value is 10). This may not necessarily correspond to fixed time intervals, as each step may be more or less expensive, depending on the strategy and the problem.

- All available data are collected, even if it appeared to be constant or unhelpful. This allows an agnostic approach to learning in which the neural network training procedure selected relevant features.

In all, 376 features are recorded, including the number of active clauses, passive clauses, and subsumptions, for instance. All appropriate TPTP problems are used

model	mean accuracy (%)	standard deviation (%)
MLP	81.5	2.0
1D CNN	82.4	3.1
GRU	83.9	1.9

Table 3.1: k-fold cross-validation classification accuracy on a balanced dataset of around 10,000 total successful/failed execution traces. $k = 5$.

with the modified Vampire to generate *execution traces*. The execution traces produced are difficult to work with, however: some are short or non-existent, others are extremely lengthy. Mean execution trace length is 536 (standard deviation: 1208), with the longest trace 9971 recorded steps long. Some features also exhibit high variance.

To deal with these problems, preprocessing is applied. Each feature is scaled to zero mean and unit variance. Data that are too short (fewer than 10 steps) are discarded, as the strategy likely did not take very long in any case. The remaining traces are sliced in the time domain into 10 evenly-sized “buckets”, then a mean over each bucket is taken to produce 10 values for every trace. This results in fixed data dimensions, normalised over trace length and normalised feature values.

However, even now these data are not representative of the classification problem desired: these traces show *completed* runs of Vampire, whereas the classifier will be used to predict the success or otherwise of runs of Vampire that are still in progress. Hence, we take “snapshots” at various stages (in these experiments, at every quarter) of the trace, discarding the rest of the data, then preprocess the remaining trace as described above. Conveniently, this also provides 4 times the original number of training examples. Figure 3.2 shows a processed trace.

3.3 Predicting Successful Strategies

Predicting which proof search attempts will succeed in time and which will fail may seem unlikely, especially when predictions are based solely on information in the execution trace. However, it is known that the “slowly-growing search space” maxim is an effective heuristic for finding good strategies in saturation-based theorem proving [RS17]. The maxim states that strategies which minimise the number of derived clauses over time are more likely to succeed.

Since the data we use includes the number of derived clauses, among many other

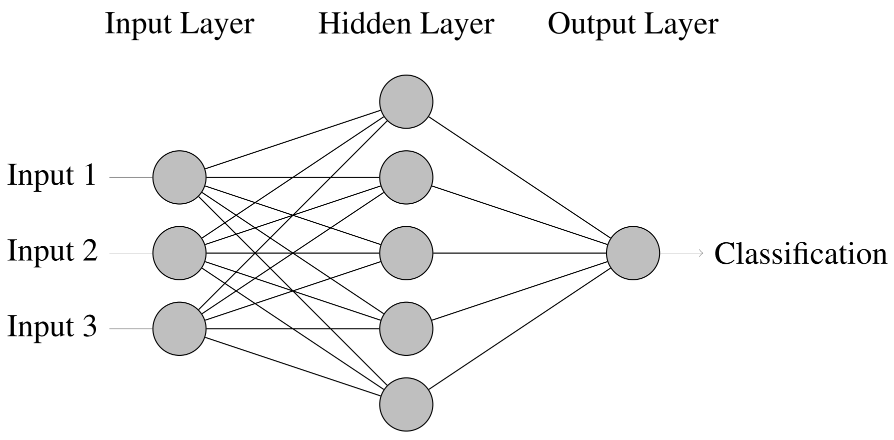


Figure 3.3: A multi-layer perceptron with an input layer, a single densely-connected hidden layer, and a single binary output neuron. A version of this model with a greater number of input and hidden neurons is used to predict strategy success at runtime.

features, it appears plausible that this approach might work at least as well as the slow-growth heuristic alone. Engineering a prediction algorithm that attempts to partition traces into "succeeding" and "failing" classes is possible with the use of machine-learning techniques (§2.2). Conveniently, some of these methods do not produce a binary output, but instead some real-valued output $f(\mathbf{X}) \in [0,1]$ which we use as the "level of confidence" in success of the trace, $\mathbf{X}$. This *success score* can be used to apply an ordering to executing strategies, allowing "smart" dynamic scheduling of strategies.

Three neural-network models are trained and evaluated: a simple fixed model and two models with an inductive bias for series data.

1. A simple multi-layer perceptron (MLP) with one input for each (bucket, feature) pair in the trace and a single hidden layer.
2. A convolutional network performing a 1-dimensional convolution pass along the inference axis per-feature, before the hidden layer.
3. A recurrent network, feeding feature series along the inference axis into a gated recurrent unit [CGCB15] before a hidden layer.

Results for this classification task are shown in Table 3.1. The consistent level of accuracy achieved is encouraging. Both of the more-advanced classifiers performed better than the simple neural network. However, the simple MLP model was chosen for integration into Vampire for implementation simplicity and for performance — it is "good enough" while also suitably fast (§1.2.3).

3.4 Intelligent Scheduling for Vampire

We show that this abstract predictor can be used in a concrete implementation for the Vampire prover. In the modified prover, it is used to run several strategies from Vampire's portfolio in a modified scheduler: strategies self-report their own execution data to the supervisor process and pause for re-evaluation at regular intervals. When a strategy halts, the scheduler then decides whether to re-schedule the strategy for some more time, or to swap it out for a different strategy.

An interesting problem is how to choose between starting new strategies and continuing to run older strategies. Additionally, if too many strategies are started (but not necessarily running) concurrently, this can consume significant memory. A tradeoff must be found between the ideal of constantly running and evaluating all available strategies in the schedule and the memory and compute cost of doing so. We implemented such a tradeoff in which strategies yet to start have a neutral *static priority* compared to running strategies, evaluated by neural network. The precise algorithm used here is as follows, taking as input a list of strategies to run:

1. Initialise an empty "run pool" of processes, with a maximum size equal to the desired number of workers (e.g. CPU cores available). Also initialise an empty priority queue of paused processes.

2. Whenever the pool is under capacity, either:

    (a) If the best process in the paused queue has a priority greater than the static priority p_{static}, wake it and move it into the running pool. In tests, a static priority of around 0.5 appeared to work reasonably well.

    (b) Otherwise, take the next strategy from the input list and start a new process to run that strategy.

3. When a strategy pauses to be re-evaluated:

    (a) Remove the process from the pool.

    (b) Re-evaluate the process priority using the neural network and the data it provided.

    (c) Insert the process into the queue with the computed priority.

4. When a strategy terminates, check if it succeeded. Otherwise, remove it from the pool.

variation	solved	new	unique	wall (s)	CPU (s)
baseline	12,899	-	420	3.03 ± 9.91	10.53 ± 33.60
no-prediction	11,827	56	33	2.86 ± 8.33	10.38 ± 32.24
prediction	12,425	111	88	2.59 ± 9.55	8.74 ± 31.03

Table 3.2: Raw results for the scheduling variations.

5. If the pool, the queue, and the set of input strategies are all depleted, all strategies have failed. Exit.

To integrate the neural network into Vampire, we took the trained network weights from our Python-based experiments, and generated a C source file with these weights expressed as a large array. The neural network's architecture was then re-implemented manually in Vampire, using the network weights compiled into the new version. This approach has several advantages: while perhaps not as efficient as using external libraries such as PyTorch [PGM+19] or TensorFlow [ABC+16] — which may use available hardware acceleration — our approach is *reasonably* efficient, low-latency, does not incur any additional dependencies, and does not add to program start-up time.

3.5 Experiments

We evaluate whether prediction results carry through to improved performance in Vampire. Note that in this setting, "success" is not easy to quantify: ideally Vampire's time-to-proof would be reduced while still proving most problems it could solve previously.

3.5.1 Setup

We take all relevant problems for Vampire from TPTP 6.4.0 (17,281 problems in total) and run three variations of Vampire 4.2.1's CASC portfolio mode:

baseline is Vampire's standard sequential portfolio mode

no-prediction uses the dynamic scheduling architecture without prediction. This effectively produces a round-robin scheduling of strategies.

prediction uses the trained neural network to predict whether a strategy will be successful as previously described.

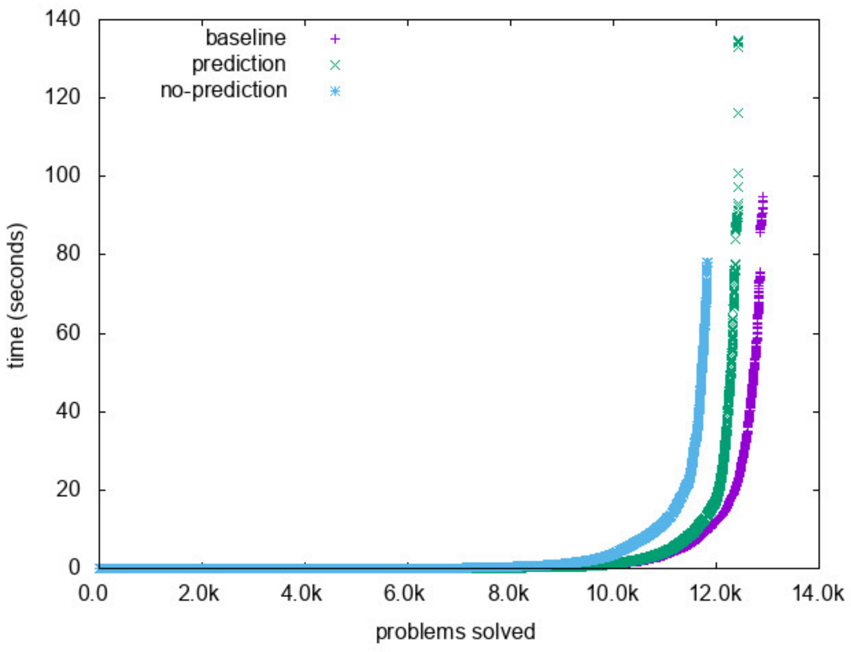


Figure 3.4: Problems solved against time.

	no-prediction		prediction	
	wall	CPU	wall	CPU
# problems worse	1,895	2,518	642	940
mean slowdown (when worse)	29.02	78.82	18.83	44.52
# problems better	451	788	650	1,297
mean speedup (when better)	7.02	28.19	3.47	10.46

Table 3.3: Amount of speedup or slowdown compared to the baseline solver.

The default 3GB memory limit and 300s time limit from the standard portfolio mode are kept but in addition each variation runs in a *multicore* setup distributing strategies over 4 cores. Experiments were run on the StarExec cluster [SST14] where each node contains an Intel Xeon 2.4GHz processor.

3.5.2 Results

Table 3.2 gives some raw numbers. Here *new* refers to the number of problems solved by this variation that were not solved by *baseline*, *unique* refers to the number of problems solved by a variation but not by the other two, and mean and standard deviation solution times are given for wall-clock and CPU time. Figure 3.4 illustrates the overall results by plotting the number of problems solved against time taken.

Table 3.3 gives further statistics comparing the two scheduling variations with the baseline. For the purposes of this table we only compare the results on problems that both the given variation and baseline solve and where the difference between solution times is greater than 1 second. This second part is important as it allows for small variations in solution times due to natural non-determinism.[1] Speedup is a multiplicative factor (a speedup of 2 means a proof was found in half the time), similarly for slowdown. "Better" means that the given variation was faster than the baseline, whereas "worse" means that the variation was slower.

3.5.3 Discussion

An immediate observation is that the overall number of problems solved is slightly worse for the variations performing scheduling. A possible explanation for this could be that the strategies in the portfolio mode are quite fragile; their performance degrades with the small overhead of context-switching and additional memory/CPU contention. Further experiments could explore this by using more generic strategies with longer time limits: many strategies in CASC mode run for less than 1 second. However, the overall number of problems solved is still high enough to make the result useful.

New problems solved by scheduling variations are an oddity, perhaps explained by introduced noise and non-determinism in the scheduling process. We note that one explanation for this could have been if the strategy schedule was longer than the time given and prediction moved a strategy into the allowed time. This is a behaviour we might expect for short time limits, but for this experiment all strategies ran.

The average solution times are improved with the two variations and the variation using prediction achieves the best solution time on average. When looking at the more detailed results of Table 3.3 we see that in the majority of cases the prediction variation was faster than the baseline, which was not the case when no prediction was applied. Furthermore, the impact of "getting it wrong" was larger without prediction: that is, the resulting slowdowns are larger.

[1]This allowance is often not made but it can heavily skew results. Without it, all variations look almost identical, as cases where variations behave in the same way dominate. This does not indicate that the variations are not an improvement, but there are many cases where there is no difference. The most likely explanation for this is that solutions are quick and no scheduling is required e.g. if the problem is solved during preprocessing.

3.6 Summary

We are aware of work in premise selection [ISA$^+$16, KBKU13, Urb07, WTWD17], static strategy selection [BHP14, KU15c, KSU13], and more recently, direct proof guidance [UVŠ11, KU15a, LISK17]. However, we are not aware of any previous work in the area of strategy selection during search for conventional theorem provers. Work on static strategy selection focuses on static properties of the input problem, rather than dynamic properties of the proof search space, with the exception of Bridge et al [BHP14]. Bridge considers various dynamic features of the search space after 100 steps of the saturation algorithm in the default mode of E. SATzilla [XHHLB08] famously and very successfully incrementally constructs a portfolio — partially based on a learned model — for SAT problems.

The aim of our experiments was to improve Vampire's overall performance, if not in the number of total theorems proved, but in the average time taken to prove problems. This approach has been shown to produce a significant increase in speed without an excessive penalty in the number of problems solved.

There are several routes that could be explored in order to further improve performance. As well as improving predictor performance by use of more sophisticated data curation, processing, and machine-learning techniques, it may also be possible to improve the naïve scheduling algorithm. Further research might include designing scheduling algorithms which keep predictions as up-to-date as possible, maximise processor utilisation, minimise memory usage/swapping, reduce context-switching overhead, or even minimise the number of required calls to the prediction algorithm.

This form of optimisation for Vampire is relatively novel: historically the aim of the team has been to prove as many theorems as possible, rather than to improve the speed of moderately-hard problem solving. As immediate impact, these developments may be useful in improving Vampire's performance in the new SLH division in the CASC competition, as well as improving the overall usability of interactive theorem proving via quicker "hammer" results, such as those reported via Sledgehammer [MP08].

This approach avoids most of the challenges discussed in §1.2 altogether by not interfering with the internal search decisions under the level of strategies, although in retrospect we suffer from an unclear objective function. This strength of the approach is also a limitation: if the set of strategies cannot solve a problem, no amount of strategy scheduling will help. Learning to schedule strategies from internal system statistics is also quite coarse-grained, well removed from the decisions made by a strategy: it is not clear at what point this approach will run out of sufficient information or decision-making power to obtain improved results.

Chapter 4

Guided Preprocessing

Material from the following chapter appeared in *Frontiers of Combining Systems 2019* as “A Neurally-Guided, Parallel Theorem Prover” [RR19a]. The paper introduced a theorem proving system which used a learned model to manipulate problems for an existing ATP. The design of the system started from a clean slate: while it is not yet particularly practical, it allows unusual features we feel may be useful in future.

We present a prototype of a neurally-guided automatic theorem prover for first-order logic. The system uses a neural network trained on previous proof search attempts to evaluate subgoals based directly on their structure, and hence bias proof search toward success. An existing first-order theorem prover is employed to dispatch easy subgoals and prune branches which cannot be solved. Exploration of the search space is asynchronous with respect to both the evaluation network and the existing prover, allowing for efficient batched neural network execution and for natural parallelism within the prover. Evaluation shows that the system can improve with learning.

Recent advances in neural network systems for logical reasoning allow for processing structured data such as terms and formulae in a neural context. This advance suggests a new breed of *neural ATP* in which proof search is guided by a neural black-box acting as “mathematician’s intuition”. However, in practice there are several implementation issues [RR19c] which must be worked around in order for neural guidance to integrate with traditional systems (§1.2). With a fresh take on neural guidance, our prototype LERNA[1] takes an alternative step toward useful neural automatic theorem provers.

[1]**Le**arning to **R**eason with **N**eural **A**rchitectures. Lerna is also the lair of the mythical many-headed beast HYDRA. Source code available at `https://github.com/MichaelRawson/lerna`.

4.1 Neural Networks for Formulae

Work on integrating machine-learned heuristics into automatic theorem provers has relied on hand-engineered features or other embedding methods [JU17, KH17], which have the advantage of simplicity and relative efficiency, but do not fully encode the syntactic structure of proof state and therefore lose information. By contrast, a method which takes into account all information should allow for greater precision in proof guidance systems. Deep Network Guided Proof Search (DNGPS) [LISK17] is an example of previous work in this area, which integrated a deep neural guidance system into the saturation-based prover E [Sch02]. DNGPS achieved successful results, but suffered from the latency introduced into the system by the neural heuristic: despite processing only a reduced amount of the available proof state, the reduction in throughput necessitated a two-phase approach in which the prover was neurally-guided in the first phase, before falling back to traditional proof search in the second (§1.2.3).

Processing structured data such as logical formulae is a relatively new domain for neural networks (§1.2.2). Some work attempts to use unstructured representations of such formulae, such as text, or build entirely-new models for a specific logic [ESA+18], whereas others attempt to re-use neural techniques for structures such as trees [BPM15]. A promising direction in this area is recent research on neural methods working with graphs [DBV16, KW16a, SKB+18], which have already been applied to premise selection [WTWD17].

4.2 Design

In order to achieve the goal of a neural theorem prover without the disadvantages associated with neural approaches, a new design of theorem prover may be required. Popular calculi used in existing ATPs tend to be unfriendly to neural guidance (§1.2.4). For such a system, we desire the following from the calculus:

Proof state must be reasonably-sized. Attempting to evaluate large proof states structurally requires a lot of computation and resources. Popular saturation-style systems produce particularly large proof states by nature.

Evaluation of states must be possible in parallel. Machine-learning algorithms can be accelerated more efficiently in batches. Top-down approaches lend themselves to this technique, whereas saturation provers are more sequential.

Subgoals must be independent and self-contained. If the prover has a notion of subgoals which must be dispatched (such as in tableau provers), these should be independent of the rest of the search space, without e.g. shared rigid variables. Otherwise, the learning system is trying to learn while ignoring the wider context of the search. For example, it can be the case that one way of closing a branch of a connection tableau makes a different branch impossible to close.

Subgoals must be intelligible and freely chosen. Destructive transformations such as the use of normal forms can obscure the original intuition behind a goal, at least for human observers. While this is not necessarily the case for learned guidance, it seems likely that removing structure will reduce model performance. Further, such transformations can have significant effect on proof search: we want the learned model to make these choices for itself.

We therefore implement a system based on a first-order tableau calculus "without unification" (i.e. variables are not treated as *rigid*), working on non-clausal, general formulae. Each goal in this case is the set of formulae present on the tableau branch. In this context, proof state is small (only the current branch), evaluation of states is possible in parallel, each branch is independent and contains all information required, and all available structure from the original problem is kept.

4.2.1 Proof Search

In the calculus (see §4.3) for the system, there are two branching factors: each goal has a set of possible inferences, and each inference contains a set of possible sub-goals. To prove a goal, at least one inference must be proved. To prove an inference, all the inferences' sub-goals must be proved (i.e. shown to be unsatisfiable). A simple optimisation is that sub-goals may be shared between inferences, so search becomes a directed acyclic graph alternating between goals and inferences, as shown in Figure 4.1.

Now the search graph can be explored: in each step, a leaf goal is selected for expansion, and all resulting inferences and sub-goals are added to the graph. If a goal has no possible inferences, it is satisfiable and can be removed from the search space. On the other hand, if a goal is trivial (i.e. contains a contradiction), it is unsatisfiable and can be marked as proven. This idea is lifted to inferences: if an inference contains any satisfiable sub-goal, it too is satisfiable, whereas if an inference contains all unsatisfiable goals, it is unsatisfiable. Proof search continues until the timeout is reached or the root goal is shown to be (un-)satisfiable. In order to dispatch trivial sub-goals quickly,

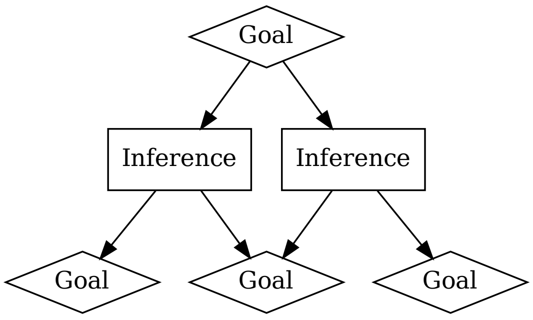


Figure 4.1: Proof search, showing shared sub-goals.

an existing fast *oracle* ATP is used (§4.4). This may mark goals as (un-)satisfiable, at which point no further exploration is required.

Search is biased by heuristic evaluation. The neural heuristic function (§4.5) evaluates each goal and assigns a score corresponding to whether the network believes that the goal is satisfiable or unsatisfiable. In order to balance exploitation of promising directions and exploration of all parts of the search space, a UCT-based [KS06] search algorithm is used (§2.3.1). This is similar to the Monte-Carlo search used in monteCoP [FKU17] but without the randomised aspect. At each sub-goal g, the prover chooses the inference i with subgoals s according to

$$\max_{i \in g} \left[\underbrace{\min_{s \in i} (\mathsf{score}(s))}_{\text{exploitation}} + c \times \underbrace{\sqrt{\frac{\ln \mathsf{visits}\,(g)}{\mathsf{visits}\,(i)}}}_{\text{exploration}} \right]$$

where score gives the heuristic score, visits gives the total number of visits to that node so far, and c is an exploration parameter ($\sqrt{2}$ in the original UCB1 definition). The sub-goal with the *minimal* score is selected: this may be counter-intuitive at first. Selecting sub-goals with minimal scores encourages attacking the "hardest" part of a goal first, and prioritises subgoals considered possibly satisfiable by the heuristic: satisfiable subgoals allow large parts of the search space to be pruned.

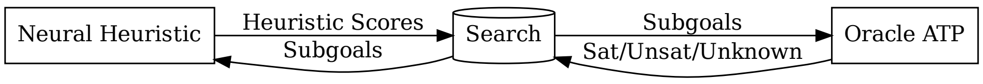


Figure 4.2: Information flow between subsystems.

4.2.2 Architecture and Prototype

The system aims to consume all available CPU and GPU resources as efficiently as possible. To that end, proof search is asynchronous: the search algorithm generates new sub-goals, which are placed on two separate queues: one for the oracle ATP, another for heuristic evaluation. Proof search then continues elsewhere, while the oracle ATP is called in parallel on each sub-goal, consuming all available CPU cores, while the heuristic consumes batches of subgoals, consuming all available coprocessor resources. As information flows backwards from these processes, the search process updates information about a given sub-goal and propagates information upwards, influencing future proof search: see Figure 4.2.

The heuristic is implemented as a server, communicating with the main prover via a TCP socket. In principle this allows for the heuristic to be a shared resource with a centralised heuristic server, or a load-balanced cluster.

4.3 Calculus

The proof calculus used in the above architecture may be extremely general: in fact, any function from goals to a finite set of possible inferences (themselves finite sets of sub-goals) will suffice, as long as each goal remains independent of any other such that the heuristic function can process all available information. If the inference system is complete, there are no additional constraints such as fair selection to maintain completeness of the system, as the balanced search algorithm ensures this (§4.2).

LERNA implements a refutation tableau calculus [Häh01] without unification. The calculus described is deliberately naïve in order to easily satisfy the design constraints given above, but could be replaced by a stronger calculus in the future. An inefficient calculus is not a problem in principle, as the heuristic should select promising areas to explore and ignore uninteresting sub-goals, and the oracle system also helps with this. However, a more efficient calculus improves performance where the heuristic fails.

$$\textsc{contradiction}\ \frac{}{\phi,\neg\phi,\Gamma} \qquad \textsc{equal}\ \frac{t=s,\phi\,[t/s]\,,\Gamma}{t=s,\phi,\Gamma} \qquad \textsc{implies}\ \frac{\neg\phi,\psi,\Gamma}{\phi\Rightarrow\psi,\Gamma} \qquad \textsc{equivalent}\ \frac{\neg\phi,\neg\psi,\Gamma \qquad \phi,\psi,\Gamma}{\phi\equiv\psi,\Gamma}$$

$$\textsc{conjunction}\ \frac{\phi_1,\phi_2,\ldots,\phi_n,\Gamma}{\phi_1\wedge\phi_2\wedge\ldots\wedge\phi_n,\Gamma} \qquad \textsc{disjunction}\ \frac{\phi_1,\Gamma \qquad \phi_2,\Gamma \qquad \ldots \qquad \phi_n,\Gamma}{\phi_1\vee\phi_2\vee\ldots\vee\phi_n,\Gamma}$$

$$\textsc{instantiation}\ \frac{\forall x_1,x_2,\ldots x_n.\phi[f(x_1,x_2,\ldots x_n)/x],\forall x.\phi,\Gamma}{\forall x.\phi,\Gamma} \qquad \textsc{exists}\ \frac{\phi[k/x],\Gamma}{\exists x.\phi,\Gamma}$$

Figure 4.3: A complete inference system for LERNA. Rules for negation are as usual and not shown here for brevity. In rule INSTANTIATION, f is a function symbol of arity n in the conclusion's signature and $x_1 \ldots x_n$ are fresh for the conclusion. In rules NON-EMPTY and EXISTS, k is fresh for the conclusion. $\phi[t/s]$ is a capture-avoiding substitution replacing t for s in ϕ.

4.3.1 Refutation Tableaux

In order to show a conjecture C from a set of axioms A_i, it suffices to negate C and then show that the resulting conjunction $A_1 \wedge A_2 \wedge \ldots \wedge \neg C$ is unsatisfiable. A set of inference rules of the form

$$\frac{\Gamma_1 \qquad \Gamma_2 \qquad \ldots \qquad \Gamma_n}{\Delta}$$

where Γ_i, Δ are sets of formulae and $\neg(\Gamma_1 \wedge \Gamma_2 \wedge \ldots \Gamma_n) \Rightarrow \neg\Delta$ is an unconditional tautology, form a refutation calculus. Proofs in this calculus can be expressed by closed trees of inference rules.

4.3.2 Complete Inferences

The inference rules in Figure 4.3 form a complete inference system, by analogy with a first-order tableau calculus without unification. A difference and point of interest is the rule for instantiating universal quantifiers: instead of instantiating a variable with any possible term t — an infinite space — it is instantiated with one function symbol (or constant) at a time, quantifying over new variables as needed. Note that no free variables are permitted in formulae: variables are always quantified. This allows

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{P(c),Q(c),\neg P(c)}\qquad\overline{P(c),Q(c),\neg Q(c)}}{P(c),Q(c),\neg P(c)\vee\neg Q(c)}\,\textsc{Disjunction}}{P(c),Q(c),\neg(P(c)\wedge Q(c))}\,\textsc{Negation}}{P(c),Q(c),\forall x.\neg(P(x)\wedge Q(x)),\neg(P(c)\wedge Q(c))}\,\textsc{Weaken}}{P(c),Q(c),\forall x.\neg(P(x)\wedge Q(x))}\,\textsc{Instantiation}}{P(c),Q(c),\neg\exists x.P(x)\wedge Q(x)}\,\textsc{Negation}$$

Figure 4.4: Suppose we are trying to prove $\exists x.(P(x)\wedge Q(x))$, given $P(c)$ and $Q(c)$, which has an obvious constructive proof via $x = c$. After pushing a negation in, x is instantiated once to c, after which the universally-quantified formula can be (optionally) dropped with a WEAKEN rule. The DISJUNCTION rule produces two subgoals that can be attacked independently.

for instantiating any term by repeated application of the instantiation rule (effectively enumerating the Herbrand universe for the goal: as usual, if there is no constant in the signature, one is introduced), but without an infinite number of possible inferences at any point. Equality is handled by a rule rewriting classes of equal ground terms. These rules are trivially complete and astonishingly inefficient, but are hoped to be used only a few times in order to provide enough of a "hint" to the oracle system.

4.3.3 Weakening

A weakening rule is an important part of LERNA's calculus, since the INSTANTIATION and EQUAL rule can produce a large number of formulae, some of which must be removed to help the oracle to prove the goal. Each application of the rule removes some amount of information from the goal in order to simplify it — this is sound and corresponds to removing an axiom from proof search. The rule is merely

$$\overset{\textsc{Weaken}}{\dfrac{\Gamma}{\phi,\Gamma}}$$

4.3.4 Simplifications

Before each inferred goal is added to proof search, it is simplified. Simplifications remove tedious inferences such as eliminating double negations, and generally reduce

$$\frac{\phi,\Gamma}{\neg\neg\phi,\Gamma}\textsc{double-neg}\qquad \frac{\phi\wedge\psi\wedge\pi,\Gamma}{\phi\wedge(\psi\wedge\pi)}\textsc{conj-assoc}\qquad \frac{\phi,\Gamma}{\phi\vee\bot,\Gamma}\textsc{disj-prop}\qquad \frac{\top,\Gamma}{t=t,\Gamma}\textsc{refl}\qquad \frac{\phi,\Gamma}{\forall x.\phi,\Gamma}\textsc{free}$$

Figure 4.5: Some simplification rules implemented in LERNA. In rule FREE, x does not occur in ϕ. Several other rules are implemented.

the search space available. Simplification rules are not intended to make choices about proof search: otherwise, using a normal form would be a reasonable step here. Figure 4.5 gives some example simplification rules.

4.4 Oracle System

One problem with the calculus as described is that proofs can be quite lengthy, even if the goal is relatively trivial. To rectify the situation, new goals generated by ongoing proof search are enqueued for attempted proof by an existing *oracle* ATP system, as described in §4.2. In our implementation we use the mature SMT solver Z3 [DMB08], which supports quantified first-order logic via a combination of decision procedures for decidable fragments (such as the Bernays-Schönfinkel class of formulae), and heuristic quantifier instantiation routines [GDM09]. Z3 is attractive for this application due to its low startup times and its ability to produce both satisfiable and unsatisfiable results.

LERNA uses Z3 as an external system (it could be replaced by an alternative ATP), running it with its Model-Based Quantifier Instantiation heuristic for 20 milliseconds. This was chosen as the shortest time in which the oracle can dispatch a reasonable amount of trivial goals (and in fact Z3 is so strong it dispatches some goals immediately, as shown in §4.6). Longer oracle runtimes might produce better performance in future, but for this approach longer runtimes begin to conflate the performance of the oracle and the performance of the system as a whole.

This application is unusual for ATP systems: very short runtimes, and a mix of true and false problem statements. Running in this setting also amounts to fuzz-testing: in development of this system a bug was rediscovered in Z3[2] which resulted in non-termination of the prover: happily, the bug was already fixed in a newer version.

LERNA might also be seen as an intelligent preprocessor for existing ATPs in this setting: existing theorem provers are known to be sensitive to small changes in their

[2]`https://github.com/Z3Prover/z3/issues/2101`

input [SM96], and generally make little attempt to split their input into smaller sub-goals, for parallelism [SS94] or otherwise. The system can therefore act as an adaptor for any existing ATP, adding parallelism opportunities and "smoothing out" sensitivity to input syntax.

4.5 Learned Heuristic

A suitable heuristic function for the system must predict a value between 0 and 1 for a given formula ϕ, where 0 represents a satisfiable goal and 1 represents unsatisfiability, based on a set of tagged formulae seen in previous proof search. Although the data is collected by running the system itself and might be considered *reinforcement* learning, for this approach data collection and learning were considered separately and hence forms a classic supervised-learning problem.

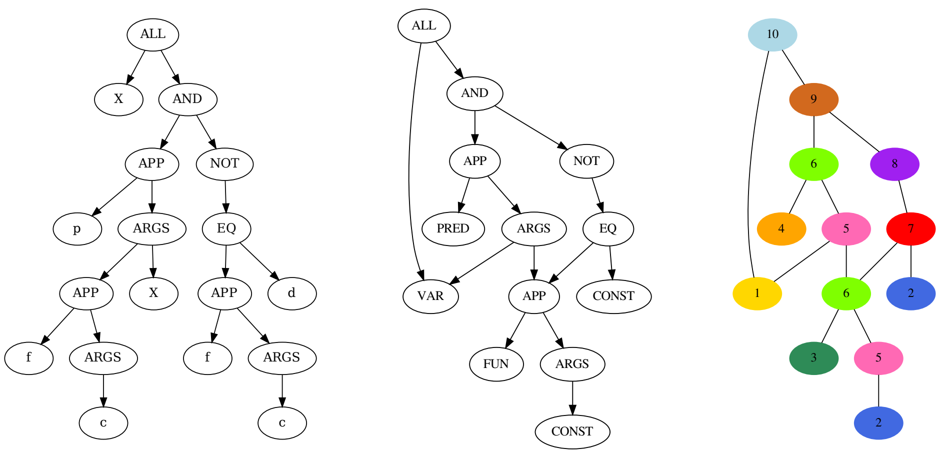
4.5.1 Data Collection

A large dataset of satisfiable and unsatisfiable goals were collected by running the unguided prover on the *M40k* dataset for 10 seconds. As soon as the prover determines the satisfiability of any sub-goal, the formula it represents and its status is recorded. This resulted in 18,340 unsatisfiable examples and 1,845,267 satisfiable examples, occupying 6GB of disk space. The dataset is imbalanced, due to a combination of weakening rules producing a large number of trivially-satisfiable examples, and to immediate prover termination after the goal is shown to be unsatisfiable. The ratio of satisfiable to unsatisfiable goals is around 100:1.

4.5.2 Translation to Graphs

Wang et al. [WTWD17] give a translation from higher-order formulae to directed graphs, and a similar scheme is used here. Constants, function symbols, predicate symbols, and bound variables are given their own node. Applications of functions and predicates to arguments are represented as an "application node" with two children: the symbol node and an "argument list" node representing the list of arguments. Propositional connectives and equality have the obvious representation, while quantifiers have two children: the variable they bind and their sub-formula.

To produce an input graph from a formula F, the formula is first parsed into an abstract syntax tree. Common sub-trees up to α-equivalence [Bar84] are merged, then the



(a) AST. (b) Nameless DAG. (c) Final result.

Figure 4.6: The translation process for $\forall x. [p(f(c),x) \wedge \neg(f(c) = d)]$ to a graph, as seen by the neural network.

resulting directed acyclic graph has any named-symbol nodes replaced with an opaque, nameless label such as “predicate” or “variable”: since distinct symbols remain as distinct nodes under this scheme, no information is lost other than the natural-language semantics of the symbol name. In practice, undirected graphs improved model performance, so the graph is made undirected before encoding nodes as integer labels to produce the final input graphs. Undirected graphs lose some information but allow bi-directional information flow between nodes.

4.5.3 Augmentation

One possible solution [SWK09] to the problem of classification on imbalanced domains is to synthesise new data for under-represented classes (in this case unsatisfiable formulae) from existing data by modifying it, c.f. augmenting image data by cropping, flipping or adding noise to existing images. There are many possible ways to augment formulae graphs: here we add a small number of random edges to the input graphs. We note that this does not necessarily preserve unsatisfiability or even syntactic validity, but it does appear to help. This augmentation requires the network to be robust to noise, which might plausibly improve performance, e.g. recognising a theorem with additional axioms. As with all such techniques, benefits are limited, and after some point increasing the amount of regularisation simply reduces performance.

4.5.4 Neural Architecture

In a typical convolutional network architecture for images [KSH12], there are a series of pixel-level stages, followed by a densely-connected neural network. Each pixel stage either combines data from local features (via *convolution*), or reduces the dimensions of the image (via *pooling*) for the next stage. Graph neural networks have analogous node-level operators, *convolution*: combining information from neighbouring nodes; and *pooling*: merging nodes to reduce the size of the input graph. Pooling may be localised to a subset of nodes (e.g. [GJ19]), or global pooling in which all nodes are collapsed into one. A brief period of experimentation with these operators yielded the following network architecture, shown in Figure 4.7.

Input. A graph G consisting of one-hot encoded nodes N and edges E.

Embedding. Each node is mapped to an embedding vector of size 64 via a trained dense embedding.

Initial Convolution. 4 convolution layers are applied to the graph with rectified linear activations. This yields a graph of the same size, but with information exchanged between nodes.

Convolution/Local Pooling. Similar convolution layers are then passed through top-k [GJ19] layers, retaining $k \approx 60\%$ of the graph's nodes. This is repeated 3 times, reducing the size of the graph considerably.

Convolution/Global Pooling. A final convolution layer feeds into a max-pooling layer, combining all remaining node data into one datum, and dropping the edge data.

Output. Fully-connected hidden and output layers produce binary classification.

It is not claimed that this is the optimal configuration, and no grid search took place to optimise the network architecture or hyperparameters. To reduce over-fitting, dropout is applied in convolutional and fully-connected layers, $p = 0.1$.

4.5.5 Training

The dataset is split into a large training set and a smaller test set (200 balanced examples), since unsatisfiable examples were time-consuming to obtain in this setting. The unsatisfiable training data were then augmented as described in §4.5.3 to produce a

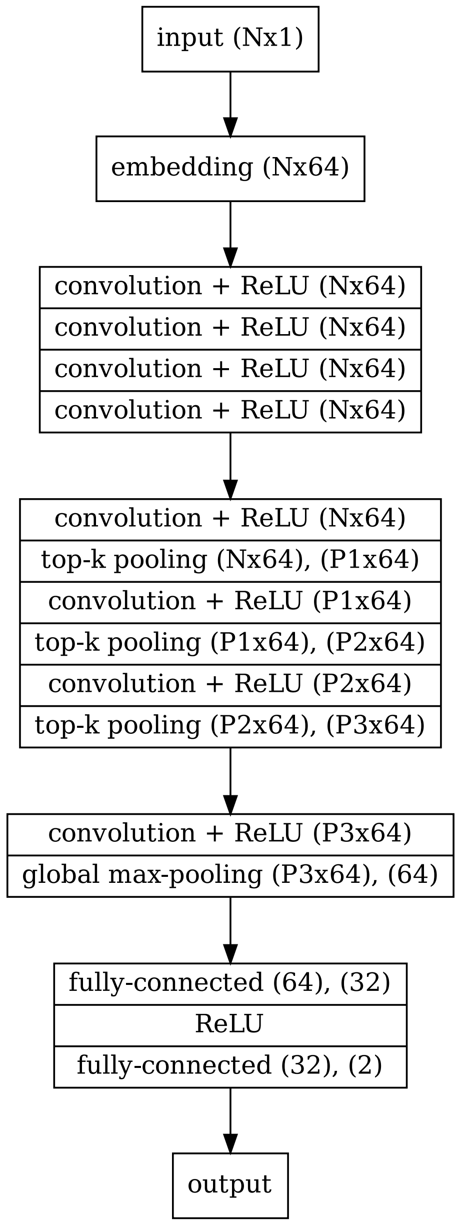
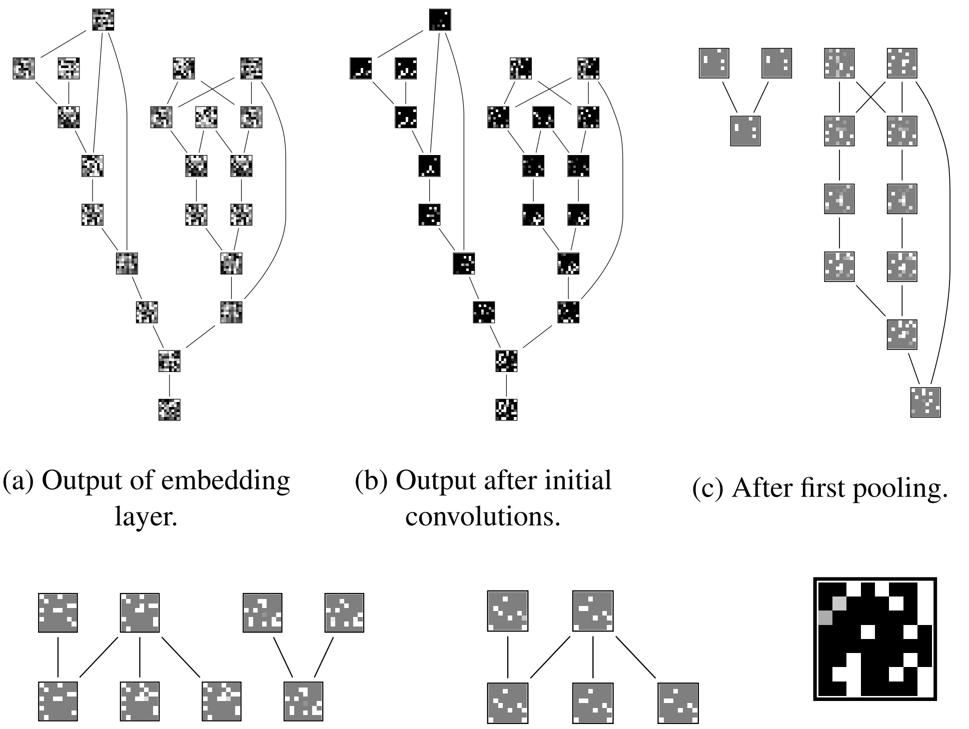

input (Nx1)
embedding (Nx64)
convolution + ReLU (Nx64)
convolution + ReLU (Nx64)
convolution + ReLU (Nx64)
convolution + ReLU (Nx64)
convolution + ReLU (Nx64)
top-k pooling (Nx64), (P1x64)
convolution + ReLU (P1x64)
top-k pooling (P1x64), (P2x64)
convolution + ReLU (P2x64)
top-k pooling (P2x64), (P3x64)
convolution + ReLU (P3x64)
global max-pooling (P3x64), (64)
fully-connected (64), (32)
ReLU
fully-connected (32), (2)
output

Figure 4.7: The neural network architecture. Initially there are N nodes, then after pooling there are P_1, P_2, P_3 nodes. Node-level embedding layers are shown per-node, graph-level convolutional and pooling layers are shown per-graph.



(a) Output of embedding layer.

(b) Output after initial convolutions.

(c) After first pooling.

(d) After second pooling.

(e) After third pooling.

(f) After global pooling.

Figure 4.8: Computation in the neural network, showing intermediate values involved in the network (correctly) predicting the satisfiability of an input formula.

balanced total training set of 3.5 million examples. The network trained on commodity desktop hardware for 8 epochs/24 hours, after which both validation and training loss appeared to cease improving significantly.

4.5.6 Network Evaluation

The network was evaluated on the balanced test set of 200 examples, as described. Various metrics for accuracy are shown in Table 4.1. While these results are very promising, we note that it is unclear how effective a train/test split is in this setting (similar subgoals may occur in both sets, even with proper data hygiene), and that this network is not attempting to determine the satisfiability of arbitrary formulae, merely those that occur in proof attempts on the *M40k* dataset. Regardless, network performance is surprising and improves proof search in practice.

metric	score
accuracy	0.930
precision	0.990
recall	0.884
F_1	0.934

metric	score
true positive	99
true negative	87
false positive	13
false negative	1

Table 4.1: Accuracy metrics for the subgoal classifier.

configuration	solved
Z3 (10s, as baseline)	1216
Z3 (20ms, as oracle)	711
LERNA, unguided (10s, with oracle)	969
LERNA, guided (10s, with oracle)	1023

Table 4.2: Total solved problems on the *M2k* dataset.

4.6 Experimental Results

To show that neural guidance can improve the performance of LERNA the system was run with and without guidance for 10 seconds on all available CPU cores. All results were collected on commodity desktop hardware.

Table 4.2 shows the total number of theorems proved using various configurations of Z3 and LERNA on the *M2k* dataset. Z3 ran for a full 10 seconds to establish baseline performance, then as an oracle for 20 milliseconds to determine the number of "trivial" problems. LERNA ran on an identical dataset, first without guidance from the neural heuristic, then with guidance. With neural guidance LERNA was able to solve an additional 54 problems and overall LERNA was shown to be complementary to Z3, proving 114 problems that Z3 was unable to solve on its own, and 40 that neither unguided LERNA nor Z3 could solve. Conversely, Z3 was able to solve more problems in total, which is unsurprising given the maturity of the tool. These results show that LERNA is able to learn from experience and complement an existing ATP.

4.7 Research Directions

Given the prototype nature of this approach, we have included a detailed discussion on possible extensions. As LERNA is a very new system, there is likely much to be gained by simple engineering and tuning: for example, the exploration parameter c has been left at its original value $\sqrt{2}$.

4.7.1 Proof Search

Heuristic search in the system has strengths and weaknesses. The amount of information ("confidence") in the system grows over time, as a result of a growing number of oracle invocations and neural network evaluations. A weakness is the heuristic search method: UCT-based methods are not an ideal fit for theorem proving, particularly with the ability to split goals. It also appears that the system sometimes determines that many directions are all equally promising, and spends time investigating all of them when one would suffice. "Can this branch be closed?" is perhaps less interesting than "what is the shortest way to close this branch?" (§1.2.1).

Another direction for proof search is an incomplete mode where branches deemed sufficiently uninteresting by the heuristic are pruned, perhaps in response to resource constraints as in limited resource strategies [RV03]. This approach, while clearly incomplete, would significantly accelerate proof search in the direction of more promising search within the available resources.

4.7.2 Prover Calculus

The calculus currently employed is deliberately naïve and extensions could be explored to reduce redundant search. As one possible view of this approach is as an intelligent preprocessor for an existing ATP, more aggressive and/or weakening inferences might be included in the calculus. For instance, *prenexing* (or conversely *miniscoping*) formulae can have a significant effect on proof search for some theorem provers, so including suitable quantifier-manipulation rules might prove to be a useful extension. *Definition introduction* for sub-formulae or terms is another similar direction.

Ideas from other tableau calculi could well be suitable for this system, such as an adapted connection rule from the non-clausal connection calculus [Ott11], as used in nanoCoP [Ott16]. Finally, this prover architecture can support other logics without excessive modification. Given that Z3 is already capable of supporting many *theories*, such as arithmetic or datatypes, a many-sorted first order logic such as those described by SMT-LIB or the TFF0 dialect of TPTP seems appropriate.

4.7.3 Oracle System

While Z3 is a strong theorem prover in its own right and performs well here, it remains to be seen if it is the best for this application. Other ATPs (or counter-example-finding systems) could be investigated. A *portfolio* of several oracle systems working in tandem might also be considered, although of course this will eventually retard proof search linearly in the number of systems present.

Reducing the number of oracle invocations is another area for optimisation. Currently, the system calls an oracle for every new sub-goal generated. It seems unlikely in some cases that the sub-goal is materially easier to dispatch than its parent — for example, in the case of propositional inferences that do not split the goal — so reducing such subgoal invocations by heuristic or randomised means is a possible area for improvement. LERNA does not currently use any information from the oracle beyond its status: using auxiliary information such as satisfying models or unused formulae could well aid proof search.

4.7.4 Machine-Learned Heuristic

Many other graph-based neural architectures are possible. Neural models specifically for theorem proving are relatively under-studied. Different approaches to formula-to-graph translation, symbol embeddings, data augmentation, and model integration may also be explored.

4.8 Summary

The introduced LERNA system implements a theorem prover with a neural heuristic processing the entire proof state, structured as a graph. After training on data automatically generated by the prover system (even where it fails to find a proof), the neural network approach is shown to be practically useful for improving proof search performance. A number of approaches (batching, oracle invocations, parallelism) are employed to improve system efficiency. While the system is not a state-of-the-art ATP, it has some unique desirable properties, among them simplicity, parallelism, parametricity with respect to calculus/oracle/heuristic, and introspection of proof state. The general approach is flexible and presently unexplored.

We now leave this exploratory work to investigate some specific areas in depth: the representation of formulae as graphs in Chapter 5, the idea of theorem proving as reinforcement in Chapter 6, and asynchronous heuristic search for tableau-style systems in Chapter 7.

Chapter 5

Directed Graph Networks

Material from the following chapter will be published in the proceedings of *Practical Aspects of Automated Reasoning 2020* as “Directed Graph Networks for Logical Reasoning”. The paper introduced neural models that work over directed graphs, allowing lossless encoding of a variety of logical data, such as formulae in first-order logic.

We introduce a neural model for approximate logical reasoning based upon learned bi-directional graph convolutions on directed syntax graphs. The model avoids inflexible inductive bias found in some previous work on this domain, while still producing competitive results on a benchmark propositional entailment dataset. We further demonstrate the generality of our work in a first-order context with a premise selection task. Such models have applications for learned functions of logical data, such as in guiding theorem provers.

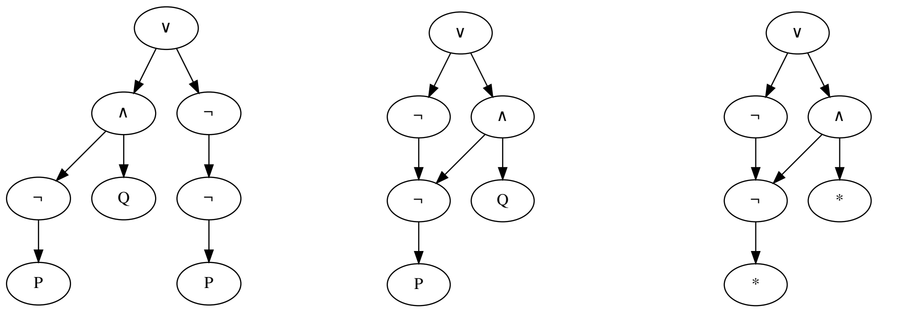
Neural networks are ubiquitous in tasks in which features must be extracted from unstructured data — tasks such as computer vision, or natural language processing. However, learning in a similar way from data that are already highly-structured is only beginning to be studied, but is sorely needed in fields such as program synthesis or automated reasoning. We approach this area from guidance of automatic theorem provers for first-order logic: an undecidable setting that nevertheless might benefit from heuristic guidance, as strategies for a subset of “useful problems” can be learned this way. It should be noted that we do not aim to *solve* known computationally-hard or undecidable problems with this approach, merely approximate these functions for practical purposes. In this chapter we explore the use of neural models for heuristic tasks on logical data, first in a propositional context, then progressing to a first-order setting.

5.1 Propositional Task

Evans et al. [ESA+18] introduce a dataset for studying the ability of neural networks to perform tasks which are "primarily or purely *about* sequence structure". The dataset consists of tuples of the form (A, B, y) where A and B are propositional formulae and y is the binary output variable. The task is to predict logical entailment: whether or not $A \models B$ holds in classical propositional logic. A and B use only propositional variables and the connectives $\{\neg, \wedge, \vee, \Rightarrow\}$ with the usual semantics. The dataset provides training, validation and test sets, with the test set split into several categories: "easy", "hard", "big", "massive" and "exam". The "massive" set is of particular interest to us as it contains larger entailment problems, similar in size to those found in real-world first-order problems.

PossibleWorldNet is introduced alongside the dataset [ESA+18] as a possible solution to the task: an unusual neural network architecture making use of algorithmic assistance in generating repeated random "worlds" to test the truth of the entailment in that world, in a similar way to model-based heuristic SAT solving or semantically-guided premise-selection systems such as SRASS [SP07] or MaLARea-SG1 [USPV08]. The PossibleWorldNet approach performs exceptionally well, but does suffer from inflexible inductive bias: it is unclear how this model would perform on harder tasks without a finite number of possible worlds, or tasks where model-based heuristics don't perform as well. Tending instead toward a purely-neural approach, Chvalovský introduces TopDownNet [Chv18], a recursively-evaluated neural network with impressive results on this dataset.

Graphical representations have been used with some success for logical tasks: Olšák et al. introduce a model based on message-passing networks working on hypergraphs [OKU19], while Paliwal et al. [PLR+19] use undirected graph convolutions for a higher-order task. Crouse et al. show that particular form of subgraph pooling [CAC+19] improves the state of the art on two logical benchmarks for graph neural networks. An interesting effort related to the propositional task is that of NeuroSAT [SLB+19], a neural network that learns to solve SAT problems presented in conjunctive normal form. We are aware of other similar work [GAA+19] developed concurrently: the work achieves good results by adding a fixed number of edge labels and edge convolutions to the model, in exchange for additional complexity and artificially limiting e.g. function arity.



(a) syntax tree (b) DAG - named (c) DAG - nameless

Figure 5.1: Producing a DAG representation of $(\neg P \wedge Q) \vee \neg\neg P$.

5.2 Approach

Our main contribution is a graph neural network model working directly on logical syntax that performs well on benchmark datasets, while remaining simple and flexible. Suitably-designed representations retain enough information such that an equivalent input (up to renaming) can be reconstructed from the representation, a property we will call *lossless*. Lossless representations have no direct practical benefit, but eliminate the possibility of poor model performance due to insufficient information (§1.2.2).

To achieve this goal we use a bi-directional convolution operator working over directed graphs and experiment with different architectures to accommodate this approach, which achieves strong performance on the propositional entailment dataset discussed in §5.1. Progressing to first-order logic, we also demonstrate a lossless first-order encoding method and investigate the performance of an identical network architecture on a first-order dataset.

5.3 Input Encoding

Directed acyclic graphs (DAGs) are a natural, lossless representation for most types of logical formulae the authors are aware of; including modal, first-order and higher-order logics, as well as other structural data such as type systems or parsed natural language. A formula-graph is formed by taking a syntax tree and merging common sub-trees, followed by mapping distinct named nodes to nameless nodes that remain distinct: an example is shown in Figure 5.1. Such graphs have previously been used for problems such as premise selection [WTWD17] or search guidance of automatic theorem provers (Chapter 4). It should be noted that the acyclic property of these graphs does not seem to be important — it just so happens that convenient representations happen to be acyclic. This representation has several desirable properties:

Compact size. Sufficiently de-duplicated syntax DAGs have little to no redundancy, and in pathological cases syntax trees are made exponentially smaller.

Shared processing of redundant terms. Common sub-trees are mapped to the same DAG node, so models that work on the DAG can identify common sub-terms trivially.

Bounded number of node labels. By use of nameless nodes, a finite number of different node labels are found in any DAG. This allows for simple node representations and does not require a separate textual embedding network, although this could be used if desired.

Natural representation of bound variables. Representing bound variables such as those found in first-order logic can be difficult [Pit01] — this representation side-steps most, if not all, of these issues and naturally encodes α-equivalence.

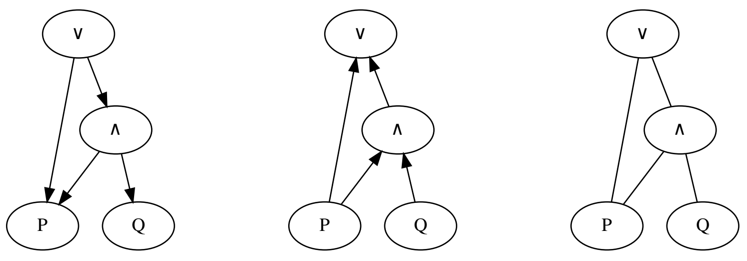
One drawback of such DAGs as a representation for logical formulae is that they lack ordering among node children: with a naïve encoding, the representation for $A \Rightarrow B$ is the same as $B \Rightarrow A$, but the two are clearly not equivalent in general. The same problem also arises with first-order terms: $f(c,x)$ is indistinguishable from $f(x,c)$. However, this problem can be removed by use of auxiliary nodes and edges such that an ordering can be retrieved, as shown in §5.6. For the propositional dataset, the classical equivalence $A \Rightarrow B \equiv \neg A \vee B$ is used to rewrite formulae, avoiding ordering issues. We also recast the entailment problem $A \models B$ as a satisfiability problem: is $A \wedge \neg B$ unsatisfiable? These methods reduce the total number of node labels used (4 in total — one for propositional variables, and one for each of $\{\neg, \wedge, \vee\}$), and allow the network to re-use learned embeddings and filters for the existing operators.

5.4 Model

We introduce and motivate a novel neural architecture for learning based on DAG representations of logical formulae. Unusual neural structures were found to be useful,

	valid	easy	hard	big	massive	exam
mean node count	23.5	23.7	47.5	51.4	80.2	9.1
maximum node count	37	39	65	86	102	13
standard deviation	4.6	4.7	5.9	11.0	6.7	2.1
mean symbol count	5.2	5.3	5.8	5.2	18.5	2.4

Table 5.1: Encoding statistics



(a) top-down only (b) bottom-up only (c) undirected

Figure 5.2: Information flow in a formula DAG representing $P \wedge Q \vee P$.

and are described first, before these blocks are then combined into the model architecture.

5.4.1 Bi-directional Graph Convolutions

We assume the input DAG is a graph $(\mathbf{X}, \mathbf{A})$ where $\mathbf{X}$ is the node feature matrix and $\mathbf{A}$ is the directed graph adjacency matrix. Various graph convolution operators — denoted $\text{conv}(\mathbf{X}, \mathbf{A})$ here as an arbitrary operator — have enjoyed recent success. These generalise the trainable convolution operators found in image-processing networks to work on graphs, by allowing each layer of the network to produce an output node per input node based on the input node's existing data and that of neighbouring nodes connected with *incoming* edges. This can be seen as passing messages around the graph: with k convolution layers, a conceptual "message" may propagate k hops across the graph. Here, we use the standard convolutional layer found in Graph Convolutional Networks [KW16a]. This operator suffers from a shortcoming (illustrated in Figure 5.2) on DAGs such as those used here: information will only pass in one direction through the DAG, as messages propagate only along incoming edges. Unidirectional messages are not necessarily a problem: bottom-up schemes such as TreeRNNs [TSM15, Gau20] exist, Chvalovský uses a top-down approach [Chv18], and cyclic edges are another possible solution. However, to play to the strengths of the graphical approach the ideal would have messages passed in both directions, with messages from incoming and outgoing edges dealt with separately. It is possible to simply make the input graph undirected, but this approach discards much of the crucial encoded structure and was not found to perform much better than chance on the propositional task. Instead, a bi-directional convolution is one possible solution:

$$\text{biconv}(\mathbf{X},\mathbf{A}) = \text{conv}(\mathbf{X},\mathbf{A})||\text{conv}(\mathbf{X},\mathbf{A}^{\mathsf{T}})$$

where the $||$ operator denotes feature concatenation. By convolving in both edge directions (with disjoint weights) and concatenating the node-level features produced, information may flow through the graph in either direction while retaining edge direction information. A concern with the use of bi-directional convolution in deep networks is that each unidirectional convolution must decrease the size of output features by a factor of at least 2 in order to avoid exponential blowup in the size of feature vectors as the graph propagates through the network. Due to the use of a *DenseNet*-style block with feature reduction built-in, this was not an issue here.

We use a directed adjacency matrix and the convolution operator

$$\text{conv}(\mathbf{X},\mathbf{A}) = \sigma\left(\hat{\mathbf{D}}^{-1}\hat{\mathbf{A}}\mathbf{X}\mathbf{W}\right)$$

where $\hat{\mathbf{D}}$ is the degree matrix of $\hat{\mathbf{A}} = \mathbf{A}+\mathbf{I}$. This is a row-normalisation of the adjacency matrix: we did not find symmetric normalisation employed in the original graph convolution operator to be useful, probably due to the lack of undirected edges.

5.4.2 *DenseNet*-style blocks

Recent trends in deep learning for image processing suggest that including shorter "skip" connections between earlier stages and later stages in a deep convolutional network can be beneficial [HZRS16]. DenseNets [HLMW17] take this to a logical extreme, introducing direct connections from any layer in a block to all subsequent layers. We found a graphical analogue of this style of architecture very useful. Suppose that $\mathbf{X}_{i-1}$ is the input of some convolutional layer H_i. Then, by analogy with DenseNets, H_i should also be given the outputs of previous layers as input:

$$\mathbf{X}_i = H_i\left(\mathbf{X}_0||\mathbf{X}_1||\ldots||\mathbf{X}_{i-1},\mathbf{A}\right)$$

However, in later layers this node-level input vector becomes very large for a computationally-expensive convolutional layer such as H_i. DenseNets also include measures designed to reduce the size of inputs to convolutional layers, such as 1×1 convolutions. We include an analogous "compression" fully-connected layer h, which reduces the input size before convolution by allowing the network to project relevant node features from previous layers:

$$\mathbf{X}_i = H_i \left(h \left(\mathbf{X}_0 \| \mathbf{X}_1 \| \ldots \| \mathbf{X}_{i-1} \right), \mathbf{A} \right)$$

5.4.3 Graph Convolution Operators and Pooling

It has been shown that the standard graph convolution layer is incapable of distinguishing some types of graph, motivating Graph Isomorphism Networks [XHLJ18]. Since logical reasoning is almost entirely about graph structure and is known to be computationally hard, it was expected that the more powerful convolution operator would produce better results, but the isomorphism operator did not outperform the baseline graph convolution operator in experiments. Similarly, localised pooling is well-known to be useful in image processing tasks, and its graphical analogues such as top-k pooling [GJ19] and edge contraction pooling [Die19] also perform well on some benchmark tasks. These also appear useful here, perhaps corresponding to the human approach of simplifying sub-formulae. However, these also did not improve performance, possibly due to the lack of redundancy in formula graphs. Further investigations into integrating these powerful methods is left as future work.

5.4.4 Architecture

A simplistic neural architecture is described. Batch normalisation (BN) [IS15] is inserted before convolutional and fully-connected layers, and rectified linear units (ReLU) [NH10] are used as nonlinearities throughout, except for the embedding layer (no activation) and the output layer.

Embedding. An embedding layer maps one-hot input node features into node features of the size used in convolutional layers.

Dense Block. DenseNet-style convolutional layers follow, including the fully-connected layer (FC) so that each layer consists of ReLU-BN-FC-ReLU-BN-BiConv. Only one block is used, with each layer using all previous layers' outputs.

network		training	
input features	4	batch size	64
convolutional features	16	momentum	0.9
convolutional layers	48	weight decay	0.0001
		initial min. learning rate	0.01
		initial max. learning rate	0.1
		learning rate decay factor	0.99995
		learning rate cycle length	8000

Table 5.2: Network and training hyperparameters

Global Average Pooling. At this point the graph is collapsed via whole-graph average pooling into a single vector. Passing forward outputs from all layers in the dense block to be pooled was found to stabilise and accelerate training significantly.

Output Layer. A fully-connected layer produces the final classification output.

A relatively large number of convolutional layers — 48 — are included in the dense block, for both theoretical and practical reasons. Theoretically, if information from one part of the graph must be passed to another some distance away in order to determine entailment or otherwise, then a greater number of layers can prevent the network running out of "hops" to transmit this information. Practically, more layers were found to perform better, particularly on the larger test categories, confirming the theoretical intuition. In principle there is no limit to the number of layers that might be gainfully included.

5.5 Experimental Setup and Results

Source code for an implementation using the PyTorch Geometric [FL19] extension library for PyTorch [PGM$^+$19] is available[1].

5.5.1 Training

Training setup generally follows that suggested for DenseNets [HLMW17]: the network is trained using stochastic gradient descent with Nesterov momentum and weight

[1]`https://github.com/MichaelRawson/gnn-entailment`

model	valid	easy	hard	big	massive	exam
PossibleWorldNet	98.7	98.6	**96.7**	**93.9**	73.4	96.0
TopDownNet	95.5	95.9	83.2	81.6	83.6	96.0
contribution	**99.4**	**99.3**	91.2	88.3	**89.2**	**97.0**

Table 5.3: Accuracy scores for the propositional entailment task

decay, with the suggested parameters. Parameter initialisation uses PyTorch's defaults: "Xavier" initialisation [GB10] for convolutional weights and "He" initialisation [HZRS15] for fully-connected weights. A cyclic learning rate [Smi17] was found to be useful for this model — we applied a learning rate schedule ("`exp_range`" in PyTorch) in which the learning rate cycles between minimum and maximum learning rates over a certain number of minibatches, while these extremes themselves decay over time. Training continued until validation loss ceased to obviously improve. See Table 5.2 for training parameter details.

5.5.2 Augmentation

No data augmentation is used as the dataset is relatively large already, and further it is unclear what augmentation would be applied: the "symbolic vocabulary permutation" approach [ESA+18] is not applicable here due to the nameless representation, but randomly altering the structure of the graph does not seem useful as it could well change the value of y unintentionally. One could imagine a *semantic* augmentation in which A is made stronger or B weaker — this would produce data augmentation without invalidating the value of y.

5.5.3 Reproducibility

Results are reproducible, but *caveat implementor*. Training runs performed on a CPU are fully deterministic, but slow. Conversely, training runs performed on a GPU are not fully deterministic[2], but are significantly accelerated. Results reported here are obtained with a GPU, but produce comparable results on repeated runs in practice.

[2]An unfortunate consequence of GPU-accelerated "scatter" operations. See `https://pytorch.org/docs/stable/notes/randomness.html`

5.5.4 Results

Experimental results are shown in Table 5.3. Results reported from PossibleWorldNet and TopDownNet ($d = 1024$) are also included verbatim, without reproduction, for comparison. Test scores of the best model on each data split are highlighted. Results show that our model is competitive on the test categories, both with algorithmically-assisted approaches (PossibleWorldNet), and with a pure neural approach (TopDownNet). The model significantly outperforms on the "massive" test category.

5.5.5 Discussion

We conjecture that our model generalises to some degree the approach taken with TopDownNet. In our model arbitrary message-passing schemes within the entire DAG are permitted, rather than TopDownNet's strict top-down/recurrent approach, which may go some way to explaining the difference in performance. However, the relationship with PossibleWorldNet is less clear-cut, and this is reflected in results: PossibleWorldNet remains unbeaten on the "hard" and "big" categories, but is surpassed on all others.

5.6 First-Order Logic

We demonstrate the flexibility and generality of our approach by also applying the same model without adaptation or tuning to a different dataset expressed in first-order logic. First-order logic is significantly richer than propositional logic, which allows us to demonstrate naturally encoding argument order with directed graphs (§5.6.2).

5.6.1 Dataset

We employ the Mizar/DeepMath premise-selection dataset [KU15b] used in the evaluation of the distributed feature representations of Kucik and Korovin [KK18], the subgraph-pooling models of Crouse et al. and the hypergraph model of Olšák et al. The task is to predict whether or not a given premise is required for a given conjecture, both expressed in full first-order logic. The dataset asserts a baseline score of 71.75%, Kucik and Korovin achieve 76.45%, while the best subgraph-pooling model achieves 79.9%. Olšák et al. report 80% but do not reserve a validation set, and further employ clausification. It is unclear to what extent clausification helps or hinders machine learning approaches on this dataset.

5.6.2 Representation

A similar input representation to that in the propositional case is used here. However, argument order in function and predicate application must be preserved in order to maintain a lossless representation. This is achieved by use of an auxiliary "argument node" for each argument in an application, connected by edges indicating the order of arguments, shown in Figure 5.3. Quantifier nodes have two children: the variable which they bind, and the sub-formula in which the variable is bound. More space-efficient or otherwise performant graph representations are a possibility left as future work. 17 node types are used in total.

5.6.3 Training and Results

We used an identical configuration as with the propositional case: it is possible that with some tuning better performance can be produced. We do however note that using fewer layers (down to around 24 — half of the original number), did not seem to hurt performance for this benchmark and significantly reduced computation requirements and GPU memory usage. Data was split as suggested[3] at the conjecture level into 29,144 training conjectures and 3252 testing conjectures (we reserve a validation set of 128 conjectures). The model achieves a classification accuracy of 79.8% on the unseen test set.

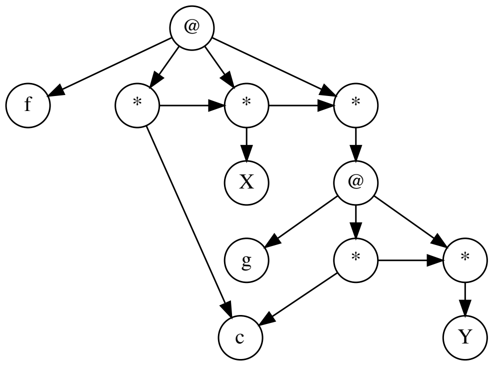
5.6.4 Discussion

The network achieves performance significantly above the baseline, comparable with Crouse et al. and Olšák et al., without task-specific tuning. We consider this a good result, suggesting that the network architecture is able to perform without adaptation on more complex tasks expressed in different logics.

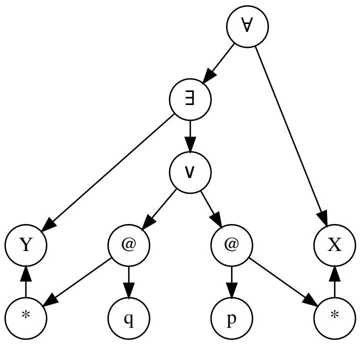
5.7 Summary

We explore directed-graph representations and a new architecture for logical approximation tasks and show that they have good performance characteristics and a number of advantages, notably simplicity. The approach can work over many different logics in principle, and practical experiment suggests this is true in practice. The network

[3] `https://github.com/JUrban/deepmath`



(a) $f(c, X, g(c, Y))$



(b) $\forall x.\exists y.P(x) \vee Q(y)$

Figure 5.3: First-order graph encodings, showing (a) argument ordering and (b) variable binding.

does not use any algorithmic assistance as PossibleWorldNet does, yet achieves competitive performance — this allows the network to process similar tasks which do not have a useful concept of "possible worlds". Combining our work with the best of other approaches, such as using the densely-connected network architecture with hypergraph methods, is a promising direction.

In some applications, such as guiding automatic theorem provers, network prediction throughput is crucial. High-performance automatic theorem prover internals typically use a graphical representation [SRV01], so graphs are a natural choice for these structures. Additionally, graph neural networks parallelise [FL19] somewhat more naturally than previous approaches such as TreeNets, suggesting that this style of network may be more applicable to these domains.

Much future work is possible, especially to improve efficiency. In particular, we suspect that multiple dense blocks might use fewer parameters or perform better than one large block. A hybrid skip-connection approach, such as connecting smaller dense blocks with residual connections, is of particular interest to us as it may reduce computational cost significantly. We used versions of this method in the following chapters as a general-purpose learning tool, and were pleased to find that it has generally-good performance without much tuning, and can adapt easily to encode extra-logical data such as actions or metadata into the graphical representation.

Chapter 6

Reinforced Theorem Proving

Material from the following chapter was presented at *Practical Aspects of Automated Reasoning 2020* under the title “Reinforced External Guidance for Theorem Provers”. At the virtual conference, we introduced a reinforcement-learning setting for automated theorem proving, discussed its merits and presented some attempts to solve it in the context of Vampire.

We introduce a reinforcement learning environment for deriving useful “lemma” facts to aid existing automated theorem provers: agents receive reward for making deductions which reduce system effort. This forms a challenging reinforcement task with applications for practical theorem proving. We present and train an exemplar deep neural agent for the environment and demonstrate deduction of helpful formulae for unseen, harder problems once trained on similar, easy problems. The environment is fully general and can accommodate any automated theorem prover, deduction system or reinforcement algorithm.

We begin with the foibles of ATP systems. Modern ATP systems can explore a large search space quickly while eliminating redundancies, but even sophisticated provers can get “stuck” in explosive search areas or fail to exploit promising directions. Such conventional systems are also sensitive to their input: adding unhelpful axioms or using a different formulation of the same problem can result in losing a proof (§1.2.1). However, the opposite is also true: adding useful lemmas — perhaps not found otherwise until significantly later in proof search — can cause a system to find a proof much more quickly.

To exploit the situation we train agents to select helpful formulae for an existing ATP from a set provided by some inference system. During training episodes on easier problems, agents are provided the problem statement combined with previously-selected formulae, and are presented with a set of new formulae derived from existing formulae by the inference system. After the agent selects a formula, a reward or punishment signal is then administered depending on whether the ATP found the deduction helpful, or unhelpful. The relationship is symbiotic:

1. The agent learns to select helpful inputs for the ATP
2. The ATP provides an approximate reward signal to guide agent learning

The former is practically useful for improving the performance of existing theorem provers: if an agent provides sufficiently useful input a proof can be brought within the the resource limits of the user, turning a timeout into a proof. The latter is not as directly useful — it is unlikely that the agent makes a good ATP by itself — but it makes an interesting tradeoff. The agent must learn to deal with the quirks of the underlying prover, but the learning signal dispenses reward or punishment more readily than simple proof/no-proof reward schemes. It further appears that learning to help an existing system find proofs is a good proxy for learning to find proofs: during evaluation we found that agents occasionally found a proof independently of the ATP.

Well-trained agents may be of enormous practical significance: as well as improved performance on problems in the domain of interest, the generality of this approach means that some amount of engineering effort can be removed. For example, the importance of parameter tuning for a given problem domain may be reduced, or some *ad-hoc* heuristic can be learned effectively.

6.1 Reinforcement Learning and Theorem Proving

Reinforcement learning (RL) deals with agents learning to take optimal actions in an environment that dispenses some reward [SB18]. There are a number of algorithms that solve variations of this problem, such as Q-learning. In Q-learning, agents learn to approximate a function $Q(s,a)$ that represents the long-term reward expected from taking action a in state s. Reinforcement learning can use deep neural networks as function approximators, as in the DQN algorithm used to play classic Atari games [MKS+15]. A recurring issue in *online* RL algorithms (the state of the art for some time) such as DQN is the amount of computation required to continuously generate relatively few training examples. This is exacerbated in our approach with computationally-expensive ATP system invocations and data-hungry deep neural networks. Offline RL [ASN19] has received less attention historically, but the appeal of collecting a dataset of environment interactions ahead of time and training without interaction drives a modern resurgence in interest.

Initial efforts have been made to apply reinforcement learning techniques to guiding theorem proving, notably by biasing Monte-Carlo tree search with learned approximations [KUMO18]. One problem observed by the reinforcement FLoP system [ZCM$^+$19] is the sparsity of reward available in theorem-proving environments.

6.2 Motivation and Learning Task

Research applying RL techniques to automated theorem proving has encountered obstacles, both in the sparsity of reward in theorem proving (either a proof is found, or it is not — no reward can be reasonably dispensed until the agent finds it) and with relatively weak initial performance of the ATP/agent hybrid (§1.2). By separating agent and ATP, our approach partially side-steps these issues in exchange for agents specialising to particular system behaviour: instead of attempting to learn some policy for a formal calculus, agents learn to help a concrete ATP system. This represents a different problem to that found in previous work directly combining RL and theorem proving. We now codify the new environment.

6.2.1 Necessity of Reinforcement Learning

At first this might appear to be a supervised learning problem: simply try a number of deduced lemmas, observe their effect on the prover and classify them as helpful or unhelpful. Unfortunately, deductions do not exist in isolation: consider two deductions that are mildly unhelpful alone but when combined produce a rapid proof, or two deductions which are mildly helpful alone but produce explosive proof search when combined. Further, the ordering of deductions matter: provers typically process inputs sequentially in some order with effect on proof search. Therefore, the task must be approached as a reinforcement learning exercise, maximising the *long-term* reward possible to achieve from making some deduction in the here-and-now.

Algorithm 1: RL environment allocating reward from an existing ATP

```
Parameter: conventional ATP system $\mathcal{A}$
Parameter: 1-step inference system $infer$
Parameter: maximum steps $T$
Input: learned (stochastic) agent policy $\pi$
Input: training problem $P$, easily solved by $\mathcal{A}$
Output: selected actions $a_t$ receiving rewards $r_t$ at step $t$
begin
    load and optionally preprocess $P$, obtaining $axioms$ and $conjectures$
    $selected \leftarrow conjectures$
    $actions \leftarrow axioms \cup infer(selected)$
    run $\mathcal{A}(axioms \cup selected)$ to get initial score $s_0$
    for $t \leftarrow 1$ to $T$ do
        sample action $a_t \in actions$ according to $\pi$
        $axioms \leftarrow axioms \setminus \{a_t\}$
        $selected \leftarrow selected \cup \{a_t\}$
        $actions \leftarrow axioms \cup infer(selected)$
        $s_t \leftarrow \mathcal{A}(axioms \cup selected)$
        compute $r_t$ from $s_t$, then report $(a_t, r_t)$
    end
end
```

6.2.2 Rules of Play

A naive approach for listing possible deductions takes all available input formulae and enumerates all possible 1-step deductions between them. While this works, with $O(n^2)$ possible binary deductions it is prohibitive on some problems. Instead, we take an approach similar to a given-clause loop: formulae are split into a *selected* set (initially empty, or the negated conjecture for refutational provers) and an *actions* set (remaining axioms and 1-step deductions from *selected*). Agents may select a formula from the *actions* group to move to the *selected* group, updating *actions* with any new inferences. After each step, the ATP system runs on the remaining axioms and the *selected* set in order to establish a reward. The RL task is therefore defined by:

- current state: disjoint sets of remaining axioms and *selected*
- possible *actions*: not-yet-selected axioms and 1-step deductions from *selected*
- a reward function based upon the underlying ATP system

6.2.3 Black-Box: Vampire

This work is deliberately agnostic to the particular theorem-proving technology used, but we use the archetypal superposition system Vampire [KV13] as a research vehicle. The system can prove many hard theorems in first-order logic, such as those found in the widely-used TPTP [SSY94] problem set. Recently, the system has been extended to support problems expressed in a higher-order logic [BR20].

In the specific case of Vampire, all formulae are converted to clause normal form in a preprocessing step first, so all formulae mentioned are clauses. Tracking which clauses are derived from the conjecture allows for the separation of conjecture and axiom clauses: this has the pleasant side-effect of making the environment somewhat goal-directed in this case. Clauses are fed into Vampire in such a way that clauses in *selected* are processed first: this allows for clauses selected by the agent to have a much larger impact (in either direction) on proof search.

6.2.4 Reward Function

Intuitively, we would like to measure the amount that a deduction contributes to a prover finding a proof more easily. There are many possible reward functions, such as the difference in the total number of formulae generated, number of proof search loops, or dispensing reward when the deduction appears in proof output. All of these are easily measured and consistent across multiple runs. However, any fixed statistic is susceptible to the complexity of the ATP system: it is for example possible for a deduction to reduce the number of formulae generated during search while simultaneously slowing the prover down due to more-involved search steps. Another possible unfortunate case is rewarding formulae which appear in the final proof, where a deduction which made a large section of search space redundant — but did not appear in the proof — is not rewarded.

Instead we measure the number of user-space instructions executed by the ATP to find a proof, as reported by the Linux utility `perf` [DEKB16]. The negation of this can be used as a scoring function: the maximum score is 0 (no instructions required to find a proof), and one score is better than another if it took fewer instructions. In practice it is helpful to subtract a constant number of instructions representing ATP start-up: this can be measured by proving a trivial statement. While there is some "jitter" incurred by using this method, this is comparatively small with respect to the difference caused by input changes. A reward function is defined in terms of this score: if a set of agent

t	*selected*	*axioms*	*infer*(*selected*)	*choice*	s_t	r_t	cumulative reward
1		1,2,3,4,5,6		2	-7	0.3	0.3
2	1	2,3,4,5,6		5	-8	-0.1	0.2
3	1,5	2,3,4,6	10	10	∞	-1.2	-1
1		1,2,3,4,5,6		3	-9	0.1	0.1
2	3	1,2,4,5,6		1	-8	0.1	0.2
3	3,1	2,4,5,6	8	6	-9	-0.1	0.1
4	3,1,6	2,4,5	8	8	-2	0.7	0.8
...							

Figure 6.1: Example training episodes illustrating the environment. Baseline score s_0 is -10.

deductions scores s_t and after a new deduction is made it scores s_{t+1}, a possible reward is $s_{t+1} - s_t$. To normalise rewards across different problems, the initial score s_0 is used as a baseline, so that the final reward function is

$$r_t = \frac{s_{t+1} - s_t}{-s_0}$$

This definition can cause some confusion due to the negative scoring. For example, if initially a problem takes 10 instructions and after taking the first action takes 5, $s_0 = -10$ and $s_1 = -5$, so therefore $r_0 = 0.5$. If instead the first action now takes 12 instructions, $s_1 = -12$ and therefore $r_0 = -0.2$.

While the maximum episode reward is now +1, it is possible for agents to produce unboundedly negative episodes with unwise deductions. Some RL algorithms used in our initial experiments were unstable in the presence of this kind of unbounded punishment. To restrict this, if the cumulative episode reward at any point is less than -1 (meaning that the ATP needed twice as many instructions as originally), the episode can be terminated. This early termination was found to stabilise training with misbehaving algorithms as episode rewards are now limited to the range [-1, 1]. Timeouts therefore result in an episode reward of -1, finding a proof outright by deduction results in +1. However, this restriction was not used with our final approach: there we allow arbitrary negative reward.

6.2.5 Example

Consider the following problem in (very) classical first-order logic to illustrate the environment. We have a knowledge base:

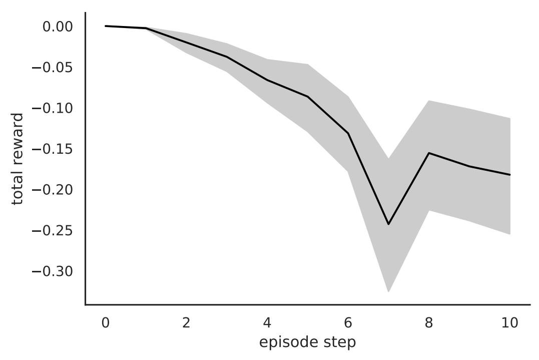
1. All men are mortal.
2. Socrates is a man.
3. Mortals eat olives.
4. Meletus is a man.
5. Mortals wear sandals.
6. Hemlock is unpleasant.

and we wish to prove the conjecture "Socrates eats olives". Only 1, 2 and 3 are required for the proof: 4 and 5 are related but unhelpful and 6 is completely unrelated. A fictional inference system might come up with the following valid deductions given some subsets of these axioms:

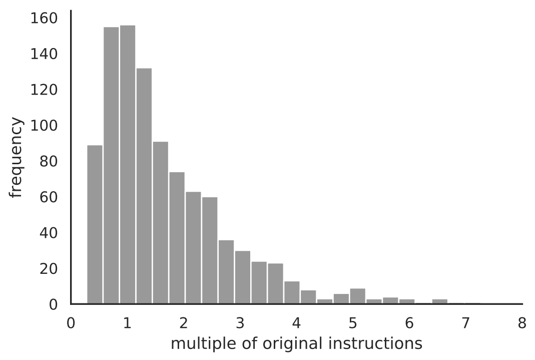
7. Socrates is mortal.
8. All men eat olives.
9. Meletus is mortal.
10. All men wear sandals.
11. Hemlock is not pleasant.
12. Mortals eat olives and wear sandals.

Among these deductions, some are helpful and others are not. 7 is the first step of the proof, while 8 is a helpful lemma: these are useful individually, but are not *more* useful when combined. 9 and 10 are meaningful but irrelevant, while 11 and 12 are redundant. Developers and users of ATP systems are no doubt familiar with more advanced but similar deduction behaviours.

Figure 6.1 shows some example episodes with a fictional (and terrible) ATP. In the first, the agent selects 1, a helpful axiom. Processing this axiom early reduces the amount of work required and produces a positive reward. The agent then selects 5, which is unhelpful but does not affect the prover too much, returning a slight negative reward. This allows the deduction of 10, which the agent then selects, thinking that this seems like a helpful lemma. Unfortunately, this causes an explosion of facts related to sandals inside the ATP, resulting in a timeout. The agent might learn to avoid sandal facts in the future when proving conjectures about olives. After some training, the agent returns to this problem for another go. This time it does much better: it successfully deduces 8, while not selecting anything too unhelpful.



(a) Running total reward up to 10 steps. Mean and 95% confidence interval after 1000 episodes.



(b) Distribution of ATP instructions required to solve derived problem after 50 steps. 1000 samples.

Figure 6.2: Illustrating task difficulty with a uniform random policy on `GRP001-1`.

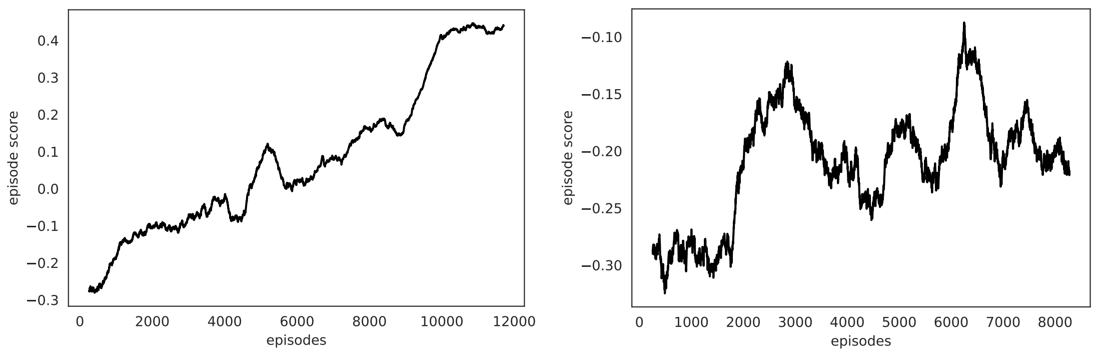
6.2.6 Environment Properties

This setting creates an episodic RL task with a discrete, deterministic action space and somewhat stochastic reward. Any algorithm for solving it must be highly sample-efficient, as each sample requires running an ATP system to completion [PU18]. The task is also relatively difficult when compared to other benchmark RL tasks: even returning zero reward requires following a narrow set of actions, mistakes are punished harshly, and delayed reward is common.

Some properties of this environment set it apart from typical RL tasks. Training and testing are distinguished meaningfully in this environment, since in testing we cannot in general receive rewards: if we could solve the problem, we would be done. Another difference is the ability to backtrack: while this is occasionally possible in other environments, it is unusual.

Figure 6.2 shows the episode reward obtained by running a random policy repeatedly on a single problem from the TPTP set, `GRP001-1`[1]. The general trend is downward, and progressively steeper as the growing number of random formulae start to interfere with proof search. Figure 6.2 also shows the distribution of instructions required from running a random policy for 50 steps, then running the prover again. A small number of random explorations improve on the original problem, but the majority are unhelpful and damage ATP performance. This demonstrates a key feature of this environment: making these selections for the ATP can be helpful, but a random

[1]Reward is computed from the Vampire ATP. Its configuration and an inference system based upon Vampire are described in §6.4.1.



(a) Single-problem reward

(b) Multi-problem reward

Figure 6.3: Total reward received with Q-learning over time on: (a) a single problem; (b) a set of problems. In the single-problem case online learning converges to a good, but not optimal, solution. With a set of problems reward still improves, but fluctuates and does not achieve positive reward.

policy is harmful more often than not. Can an agent learn to do better?

6.3 Solving the Reinforcement Task

At this point there are two practical problems to solve: train an agent to perform well on this RL task, and subsequently use the trained agent in some way to improve ATP performance.

6.3.1 Online, Model-Free Learning

We implemented a number of different conventional online, model-free RL algorithms in search of performance, but none were particularly successful. Significant engineering effort, manual tuning and reducing the task to that of learning a policy for a single problem produced very few positive results. DQN [MKS+15] (value function estimation), REINFORCE [Wil92] (policy gradients) and A3C [MBM+16] (actor-critic with parallelism) were all implemented, but would not reliably converge in reasonable time on the single-problem case and performed poorly when required to generalise. DQN with a large replay buffer was the best-behaved among these, shown in Figure 6.3. Even when algorithms behaved well, learned policies were prone to "forgetting" good policies as training continued. Impractical amounts of computation were required to gain any measurable episode reward in all cases.

While these shortcomings are known difficulties in deep reinforcement learning, we also suspect that current online methods are not well-suited for this task. The relatively large amount of compute required for assigning reward means that algorithms must learn quickly from relatively small amounts of data in order to deliver visible progress. DQN's use of a replay buffer allows re-use of samples: this may explain why it outperformed other methods for this task. Moreover, the performance of these algorithms is also typically evaluated on control and game-playing tasks such as MuJoCo [TET12], presumably very different from this one.

6.3.2 Offline Learning

To reduce the amount of computational effort required, offline RL techniques are promising. Offline approaches are known to work well on heuristic search tasks such as two-player strategy games [SHM$^+$16], but it is not clear how to adapt these techniques to our task, especially if expensive backtracking search at test time is avoided. At this point applying algorithms from literature was abandoned in favour of an *ad-hoc* practical approach which works well on this task.

First, a large tree of possible states is explored from a training problem. Each state is evaluated by the target ATP, and the results propagated upwards through the tree to provide an estimate of the long-term discounted reward from taking a given action in a given state. Toward the leaves of the tree, this estimate becomes myopic: this is accepted as a limitation for now. In practice this seems not to matter, perhaps the better-quality estimates for actions taken at the start of an episode have more impact on long-term performance. We leave placing greater emphasis on better estimates during training as future work.

Once this estimate for long-term reward is obtained, we produce a target Boltzmann distribution $\hat{\pi}(s)$ over the possible actions in a given state s from these rewards[2]: the better the long-term reward, the more likely the action. Finally, we train a function approximator to estimate this distribution as $\pi(s)$ by minimising the KL-divergence $D_{KL}(\hat{\pi}||\pi)$. We again emphasise the lack of theoretical backing for this approach, but note that this appears to be better in practice than merely training the agent to select the action with the greatest estimated reward. In the event that the agent picks an action incorrectly, this biases training so that the "best of the rest" is preferred rather than treating all non-optimal actions equally. Such a stochastic policy also fits well with the approach taken in 6.3.6.

[2]merely a convenient way to map rewards into a probability distribution [SB18]

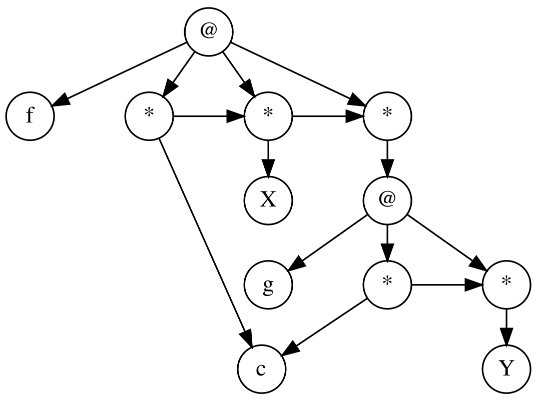
6.3.3 Data Generation

A fundamental problem of generating training data for offline learning is matching the distribution of data to "real-world" data, that seen by fully-trained agents during testing. If an agent sees only data from good situations, then it is incapable of getting itself out of a bad one. Conversely, given only bad situations, it is unlikely to fully exploit good choices. Therefore, simple exploration techniques such as best-first search (too optimistic) or breadth-first search (too pessimistic) are unlikely to produce good training distributions. Instead, we use a UCT-maximising search [KS06] (§2.3.1) similar to that employed in AlphaGo and rlCoP, without random playouts. Playouts (and associated leaf value estimates) are replaced by an ATP invocation to compute reward for the leaf's state. This information is propagated upwards to estimate maximum discounted rewards for branches. Leaves which exceed resource limits or have no possible inferences are considered *closed*, and branches are closed if all their children are.

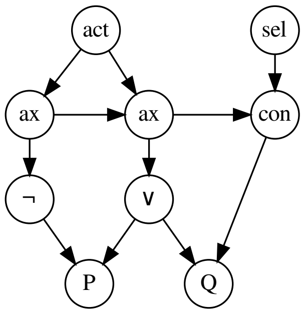
This approach balances *exploration* (expanding rarely-visited parts of the search tree) and *exploitation* (expanding promising parts of the search tree), and therefore provides a somewhat better distribution of training examples for the agent. After a round of training, a tree of states and corresponding discounted reward estimates for their children is generated. Previous approaches have used the number of visits to child states to induce a probability distribution on actions. However, this is not particularly efficient in our case: leaf nodes are all visited once but have significantly different rewards. We take a Boltzmann distribution over a state's estimated discounted rewards (with manually-tuned temperature τ) and use the resulting probabilities.

6.3.4 State Encoding

In order to represent the structure of states and actions, we use a graphical representation, as in Chapter 5. Available actions, axioms and conjectures are tagged as such with auxiliary nodes and edges. Additionally, the order in which formulae are given to the ATP can make significant difference to system performance: this is encoded by adding directed edges between top-level formula nodes. No attempt was made to add further information to the graph, although in future auxiliary data might be used to communicate prover-level information such as term orderings or formula parents to the agent. Figure 6.4 shows a fragment encoding a first-order term, and a complete graph encoding initial state of a propositional problem: if $\neg P$, and $P \vee Q$, then Q.



(a) Fragment encoding $f(c, X, g(c, Y))$.



(b) Complete state for a propositional problem.

Figure 6.4: Encoding states with directed graphs. Note auxiliary nodes and edges encoding function argument order, and "marker" nodes differentiating formula roles in the overall state.

6.3.5 Network Architecture

We use a residual, rather than dense, version of the network described in Chapter 5. The only difference is at the output: node-level features for action nodes are projected out of the graph, then fed through a fully-connected layer to produce the required per-action scalars. The neural network architecture employed to process and learn from data of this kind is not important, and we use it here only as a practical instrument. Training used mini-batch stochastic gradient descent with Nesterov momentum, L_2 regularisation and a cyclic learning rate with fixed triangular policy [Smi17]. See §7.5.2 for a more detailed discussion of a similar approach.

6.3.6 Application of Learned Policy to ATP Systems

The end result of a successful training routine in our environment is a learned policy π selecting an action a for the given state s — or, in the stochastic case, producing a probability distribution over available actions. Although seemingly useful, it is not immediately obvious what to do with this object to improve ATP performance on problems not solved within resource limits.

To avoid the difficult engineering and inevitable performance penalties experienced when adding learned guidance to existing theorem provers, we take a simple fire-and-forget approach: a fixed number of deductions are made one after another according to the policy, then the agent's work is finished and the unmodified ATP runs on the derived

Table 6.1: Tunable parameters for various parts of the system.

Generation		**Network**		**Training**	
parameter	value	parameter	value	parameter	value
exploration factor c	1	residual modules	16	min. learning rate	0.0001
temperature τ	5	layers per module	2	max. learning rate	0.001
discount factor γ	0.99	node channels	64	cycle period	20000
		hidden layer neurons	1024	batch size	64
				L_2 weight penalty	0.0001
				momentum	0.95

problem unhindered by guidance penalties. An effect similar to portfolio modes can be achieved by repeating this process with a stochastic policy so that from an original problem P a random process of derived problems $P_1, P_2, P_3 \ldots$ can be produced for the ATP to attempt. We do not claim that this is optimal, and leave variations such as best-first search or e.g. UCT-based methods as future work.

6.4 Experiments

We perform three experiments of increasing difficulty to exercise the environment, test our solution method and investigate practical uses for this approach. The first is a pilot study training and testing on a single easy problem: this is of no practical use, but provides a sanity check for the system. We then show that on a very limited problem domain (synthesising Church numerals), the system can learn to identify patterns so that harder problems can be brought within reach by pointing the system in the correct direction. Finally, an entire TPTP domain (`GRP`, group theory) is selected to provide a realistic use case. An online repository[3] contains scripts and data that (i) were used to produce these experiments, and (ii) can be adapted for similar work.

6.4.1 Experimental Setup

We set up our experiments in the context of the Vampire system. A deterministic configuration[4] of Vampire is used as the underlying ATP, and a special mode is used as the inference system, although in full generality this could employ a separate inference

[3] `https://github.com/MichaelRawson/paar20`

[4] no portfolios (§3), DISCOUNT in place of LRS [RV03], no AVATAR [Vor14], otherwise default — AVATAR is in principle deterministic, but caused more jitter than we would like here

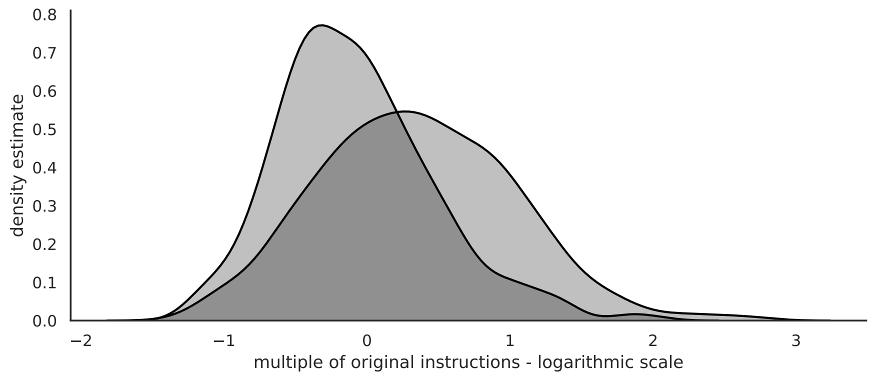


Figure 6.5: Kernel density estimates showing distribution of relative instructions required for Vampire to solve problems derived from `GRP001-1` with a learned (left) and random (right) policy. 1000 samples per distribution, Gaussian kernel, natural logarithmic scale for clarity.

system. The mode performs all 1-step inferences in Vampire's calculus from a set of clauses efficiently, while removing redundancies. Clauses are kept in textual form until conversion is required for processing by the neural network. The neural network implementation and training uses the PyTorch [PGM+19] library for GPU acceleration. Tunable parameters for the system are listed in Table 6.1.

6.4.2 Pilot: `GRP001-1`

We trained an agent as described above on approximately 5000 samples generated from `GRP001-1`, a relatively easy problem for Vampire. The experiment was a success: convergence was assured, compute requirements significantly reduced, and the resulting learned policy is markedly better than a random policy. To quantify the performance of different policies, we first run the learned stochastic policy for 50 steps to generate a derived problem, then measure the number of instructions required for Vampire to solve the derived problem, relative to that required to solve the original problem. The distribution generated by our learned policy compared to a uniform random policy is shown in Figure 6.5. The best episode with learned policy produced a derived problem requiring only 28% of the instructions required to solve the unmodified problem.

6.4.3 Small Domain: Synthesising Church Numerals

The Vampire theorem prover has been extended to support higher-order logic (HOL). This is achieved by implementing a recently-developed modification of first-order superposition which is complete for HOL [BR20]: the calculus represents an unfolding of higher-order unification into superposition. This differs from standard first-order superposition mainly in the addition of an inference rule, *narrow*. In essence, *narrow* involves rewriting with a combinator equation left-to-right. The approach is a new method for higher-order theorem proving and is yet to be fully evaluated, but preliminary investigations suggest that on problems requiring complex higher-order unification, it is at a disadvantage in comparison to calculi based on λ-calculus.

Church numerals are representations of natural numbers using higher-order functions [Chu36]: problems involving such numerals often require the synthesis of complex unifiers. Problems that posit the existence of a particular numeral require unification (or narrowing in case of Vampire) to synthesise this numeral. An example is "there exists a Church numeral n such that $n \times n = n + n$": clearly $n = 2$ is a constructive proof for this problem, but this is not trivial for systems attempting problems involving larger values of n. With recurring structures and an explosive search space, this is a promising domain for application of our learning method. We curate a set of training problems which synthesise the numeral 2 (which Vampire solves easily), and attempt to learn to improve performance where larger numerals are required.

While the overall approach here is the same, some extra details appear in the formulae (types, function application operators and a finite set of combinators necessary for the HOL calculus), which should be treated semantically in the state-encoding graph. For example, explicit function applications are treated as normal first-order function applications, and the combinators each receive a special node label of their own.

Training on this domain was more difficult than initially expected: as well as the significant technical complexity involved, environment dynamics are unusual compared to typical first-order problems. We found that it was relatively rare for the ATP to solve a modified problem (possibly due to the explosive nature of some derived clauses), but that occasionally the learning system found a proof outright during training or testing. This somewhat negates our system's main advantage of incrementally dispensing reward, and makes it difficult to generate large training datasets.

Results are therefore somewhat mixed: trained policies solved the training problems significantly more frequently than a uniform random policy. Both trained and random policies were capable of reducing time taken for problems test problems involving synthesis of the church numeral 3, but it was more difficult to show that training helps here: we report the result anyway to demonstrate the strengths and weaknesses of the approach. We expect that tuning system options or behaviour to produce a more graduated reward will yield an easier task: we have at least shown that modifying the input problem can significantly help the ATP system.

6.4.4 Large Domain: Group Theory

We attempt the ultimate task for this kind of approach, generalising from easier problems to related harder problems: we envisage a future ATP tuning system in where a large corpus of easier problems allow training to attempt a few remaining unsolved problems, a problem of significant interest for the community [Reg19].

The TPTP `GRP` domain is used for this study, containing a large number of first-order problems with a range of difficulty. The configuration of Vampire used for the pilot study can solve 384 of the problems in this domain in under 1 second: these were used as training problems while the remaining 706 problems form the test set. It is unclear to what extent the problems within `GRP` are actually similar from the ATP perspective: this is a reasonable criticism, but we expect that in practical problem sets similarity is equally unclear. Sufficient data can be easily generated from these training problems: we generated approximately 100,000 data in under a day with modest parallelism on commodity desktop hardware. Network training continued until loss on a small held-out validation set failed to improve.

We note at this point the difficulties of evaluating any new ATP feature or modification [RSV14]. In our case the situation is only worsened by the necessary presence of non-determinism, the relatively strong effect of starting afresh in first-order theorem proving (c.f. portfolio modes and Chapter 3) and the very large differences in performance that can be caused by modifying problems. As a compromise, we evaluate four configurations of Vampire on the test set of problems in `GRP`:

1. 6 runs of a uniform-random policy for 10 steps, producing 6 derived problems for Vampire to attempt within a time limit of 10 seconds.

2. 6 runs of the learned stochastic policy for 10 steps, as with the uniform policy.

3. Vampire running as usual with a time limit of 10 seconds. This forms a lower baseline point of comparison against policy configurations.

4. Vampire running continuously for 60 seconds as the upper point of comparison.

Table 6.2: Results for 4 Vampire configurations on harder problems in `GRP`.

configuration	solved	unique
baseline (10s)	191	0
uniform random policy (6×10s)	206	0
learned stochastic policy (6×10s)	234	7
baseline (60s)	275	26

Results are shown in Table 6.2. First, a learned policy improves by some margin over a uniform random policy, although it is difficult to make arguments about significance here without prohibitive computational cost. Manual inspection of random/learned policy behaviour on a few test problems tends to show that learned policies produce fewer timeouts, but where solutions are found the distribution of instructions required is not significantly different. Further, it appears that problems where the learned policy performs well are similar to problems in the training set: this is discouraging from the point of view of learning general heuristics for Vampire, but encouraging for the idea of learning from easy problems in a hard problem set.

Overall, the 60-second baseline is stronger in terms of total number of problems solved than either of the 6×10 second runs, but the learned policy solves 7 problems that the 60-second run cannot, lending some credence to the idea that generating useful lemma clauses is worthwhile for first-order theorem proving. Even without learning, this approach is potentially useful for some sort of portfolio mode.

6.5 Summary

A novel reinforcement learning task for aiding practical automated reasoning systems by deducing helpful lemmas is motivated and implemented. It has some advantages over other methods for combining automated reasoning and reinforcement learning, in exchange for learning a policy specialised to concrete ATP systems rather than a formal proof calculus. Successfully solving such a task in general has significant potential practical benefit for ATP systems.

In this chapter, we set out the RL task and show that it is at least partially tractable for specific domains by means of practical experiment. We develop an offline RL method for this task to combat the significant computational cost incurred, then show that it trains to a useful policy within a single-problem environment. Subsequently, we show that this approach has good initial performance on more difficult practical reasoning settings. When suitably trained, our exemplar deep neural agent can select lemma facts which reduce the time required sufficiently to bring a difficult proof within resource limits that would not otherwise be found.

This environment allows many possible developments from the basic task. In our experiments, we focus on specialising to a particular problem domain, leaving the bias to a particular ATP system implicit. Human engineers have spent significant time engineering good heuristics for existing systems that work well across many different domains: future experiments might also include a very large, cross-domain experiment to focus on the ATP system, rather than the domain of application. Developments in interpretable models [GMR$^+$18] may even allow for automatic discovery of good ATP heuristics for manual implementation.

The reinforcement task laid out seems to have many positive aspects, but persuading existing RL techniques to make any progress is very difficult. Additionally, the ad-hoc offline method developed to solve this task in reasonable time and with reasonable convergence is somewhat suspect from an RL standpoint: we have no theoretical guarantee that our agent is learning anything beyond optimising for a distribution we thought relevant (although practical results show that it is at least partially *useful*). Further detailed research is required to either shore this approach up theoretically, or find an existing method in the RL literature that has better performance.

Chapter 7

Asynchronous Policy Evaluation

Material from the following chapter is due to appear in *TABLEAUX 2021* as “lazyCoP: Lazy Paramodulation meets Neurally Guided Search”. The paper introduces a system, lazyCoP, designed from scratch to use expensive learned guidance without slowing inference rate. It is also the first known implementation of the “lazy paramodulation” calculus LPCT, an approach to equality handling in connection calculi.

State-of-the-art automated theorem provers explore large search spaces with carefully-engineered routines, but do not learn from past experience as human mathematicians can. Unfortunately, machine-learned heuristics for theorem proving are typically either fast or accurate, not both. Therefore, systems must make a tradeoff between the quality of heuristic guidance and the reduction in inference rate required to use it. We present a system that is completely insulated from heuristic overhead, allowing the use of even deep neural networks with no measurable reduction in inference rate. Given 10 seconds to find proofs in a corpus of mathematics, the system improves from 64% to 70% when trained on its own proofs.

The great majority of automatic theorem provers use some kind of heuristic search. This could be simple, such as the use of iterative deepening on a certain property to achieve completeness [OB03]; complex, as in hand-engineered schemes [GS20]; or even learned in some way [UVŠ11]. Such heuristics are critical for system performance: an excellent heuristic could find a proof in linear time, while a poor heuristic increases search time drastically. Historically these routines have been engineered, rather than learned, resulting in fast yet disproportionately-effective heuristics like the age/weight schemes [SM16] used in systems like Vampire [KV13].

Learning a good heuristic from previous proof attempts has become more popular recently, and can achieve good results [JU17]. Techniques from machine learning can approximate complex functions that are difficult to discover or write down, but this comes at computational cost (§1.2.3). This cost can result in an unfortunate outcome where a learned heuristic that appears promising during testing actually *degrades* performance when included in a concrete system, due to reduced inference throughput.

Even assuming a heuristic is both fast and accurate, it is not always clear how to gainfully include predictions into existing target systems, particularly as a single wrong prediction can sometimes have disastrous results (§1.2.1). Approaches are either ad-hoc or adapt existing techniques from other domains which are not necessarily well-suited to theorem proving.

7.1 Approach

We construct a system from scratch designed to avoid these issues. lazyCoP is an automatic theorem prover for first-order logic with equality in the connection-tableau family (§7.3). The system may use a policy learned from previous proofs (§7.5) to bias a special-purpose backtracking search (§7.4.1) toward areas the policy considers promising. Performance penalties are eliminated by asynchronously evaluating the policy network on a coprocessor, such as commodity GPU hardware (§7.4.2).

The result is a system in which learned guidance has no measurable impact on inference rate (§7.6.1) and learns in a feedback loop from previous proofs on a set of training problems (§7.6.2). No manual features are used for learning, and the only manual heuristic used is "tableaux with fewer subgoals are more likely to lead to a proof". The system augmented with the final learned policy improves from 64% to 70% in real time when provided with a coprocessor for policy evaluation.

7.2 Performance Penalties in Existing Solvers

The rlCoP system introduced in "Reinforcement Learning of Theorem Proving" [KUMO18] is the inspiration for this chapter and is most similar in spirit. A connection-tableau system is guided by Monte-Carlo Tree Search (*MCTS* henceforth, as in work on two-player games [SHM+16]), learning both policy and value guidance with gradient-boosted trees from hand-engineered features. Learning from previous proofs or failures is a common approach for many different applications of machine learning to theorem proving, avoiding the need to generate data manually. For instance, all learned premise-selection systems we are aware of are trained using premises used by automated systems in existing proofs [WTWD17, ISA+16]. rlCoP sets up a feedback loop in which new information automatically found by the system is added to the training set in order to guide future iterations, as we do here.

Connection tableaux and classical first-order logic are popular settings for other internal guidance experiments — notably monteCoP [FKU17], rlCoP, MaLeCoP [UVŠ11], FEMaLeCoP [KU15a], and FLoP [ZCM+19] — but internal guidance for other domains exist, including first-order saturation systems [JU17], SAT and QBF solvers [SLB+19, LRSL20], and systems for higher-order logics [BLR+19, FB16].

Performance is a recurring problem for systems with learned internal guidance. The authors of rlCoP exclude some kinds of learned models for performance reasons, and results are reported based on an inference, rather than time, limit. "Deep Network Guided Proof Search" [LISK17] reports that the main bottleneck in the guided saturation-style system E [Sch02] is the evaluation of inferences, and suggest a two-phase guided/unguided approach to theorem proving with learned guidance.

7.3 Unguided System

If an unguided system is completely hopeless, little progress can be made: very few positive training data can be generated from successful proofs, and the learned guidance must be better still in order to achieve reasonable performance. However, it is not as simple as selecting a state-of-the-art theorem prover, as some are more amenable to guidance than others (§1.2.4). Instead, there is a spectrum of different possible research directions, from attempting to guide weaker-yet-amenable systems up to meet stronger unguided systems, to integrating learning into already-strong systems which are not so easily improved by guidance.

The guidance scheme suggested here is designed for backtracking search, such as that found in systems based on connection calculi. It is not clear how this could be adapted to a modern saturation theorem prover such as Vampire or E, which employ proof-confluent search with a time-sensitive choice point at the selection of a given

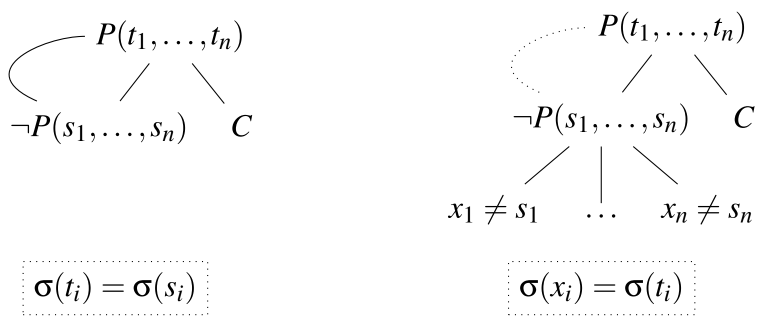


Figure 7.1: Adding $\neg P(s_1,\dots,s_n) \vee C$ to a tableau where $P(t_1,\dots,t_n)$ is the goal. The left tableau shows conventional "strict" extension, the right LPCT "lazy" extension.

clause. The basic system must therefore be as strong as possible while still allowing backtracking policy-guided search, and lazyCoP makes some attempt at this, notably with respect to equality (§7.3.2). A prototype version [RR20c] entered the most recent CASC competition [Sut16], and subsequent work has improved performance.

7.3.1 Connection Tableau Procedures

lazyCoP belongs to the connection-tableau/model-elimination family [LS01] of theorem provers, which includes systems such as leanCoP and SETHEO. *Connection* tableau methods reduce the search space by constraining general tableaux such that each addition to any given tableau must be *connected* in some way to the current leaf, as shown on the left-hand side of Figure 7.1 where $P(t_1,\dots,t_n)$ connects to $\neg P(s_1,\dots,s_n)$.

Since there is often more than one possible next step in building a tableau, not all of which lead to a proof, it is necessary to backtrack if a misstep is made. Typical connection systems often use some kind of iterative deepening to maintain completeness, but any reasonably-fair scheme works: rlCoP uses MCTS for this purpose.

7.3.2 Lazy Paramodulation

Reasoning with equality has traditionally been a weak point of connection systems. The most widespread method for efficiently reasoning with equality, *paramodulation* [NR01], is incomplete in the obvious formulation for connection tableau due to insufficient flexibility in the order of inferences. There have been various attempts to remedy this deficit, but as yet there is no conclusive solution.

lazyCoP uses the “lazy paramodulation” proof calculus LPCT [Pas08], which relaxes some of the classical connection-tableau rules in exchange for a paramodulation-like rule and some extra refinements. The basic idea is delaying unification to allow rewriting terms in the resulting disequations. For example, in the right-hand side of Figure 7.1, it is not required that $P(\bar{t})$ unify with $P(\bar{s})$ immediately as in the classical calculus, instead deducing that at least one of the terms must not be equal. Terms may still be unified with a reflexivity rule dispatching goals of the form $s \neq t$.

This implementation detail of lazyCoP is not the main focus of this chapter: the vital feature of the proof calculus is backtracking proof search.

7.3.3 Calculus Refinements

To improve performance against the pure calculus, lazyCoP implements a number of well-known refinements of the classical predicate calculus (which are lifted to equalities where appropriate), including tautology deletion, various regularity conditions, and *folding up*, a way of re-using proofs of literals. Additionally, it is frequently the case that a unification is “lazy” when it could have been “strict” — such as where no equality is present. lazyCoP implements “lazy” and “strict” versions of every relevant inference rule, which shortens some proofs considerably. The resulting duplication is eliminated by not permitting “lazy” rules to simulate their “strict” counterparts.

It is not clear whether some refinements help or hinder the learned-guidance scenario (§1.2.4). Some are definite improvements: folding up and strict rules decrease proof lengths and therefore increase the potential benefit of learned guidance. However, others, such as the regularity condition or the term ordering constraints in LPCT, are not as clear-cut. In some cases such refinements lengthen proofs significantly, outweighing the pruning effect, and previous work shows that guidance can partially replace these pruning mechanisms [Goe20]. We leave all refinements switched on for this approach, but allowing the learned policy a greater amount of freedom is a particularly interesting extension for this approach. Some techniques such as *restricted backtracking* [Ott10] sacrifice completeness for performance. lazyCoP does not implement any approach known to be incomplete: all problems attempted can be solved in principle.

7.4 Proof Search

Given a learned policy[1], we aim to use it to improve proof search outcomes (§1.2.1). The *policy* $\pi(a \mid n)$ is a function from a tableau n and possible inferences a to a probability distribution. We work with an explicit search tree, each node of the tree representing an open tableau, although tableaux are not actually kept in memory for efficiency reasons. From each open tableau, there is a positive non-zero number of possible inferences (or *actions* in the reinforcement learning literature) which may be applied to generate a new child tableau. Nodes with zero possible inferences cannot be closed and are pruned from the tree. The root of the tree is an empty tableau, from which possible inferences are the *start clauses*, in this case clauses from the conjecture.

7.4.1 Policy-Guided Search

There are many possible tree search algorithms which can include some kind of learned heuristic. We experimented with the classical A^* informed-search procedure, although we found that it was difficult to learn a good heuristic function that was neither too conservative nor too aggressive. Other approaches might include the aforementioned MCTS, single-player adaptations of MCTS [SWVDH$^+$08], single-agent approaches like that of LevinTS or LubyTS [OLLW18], or simply following a stochastic policy with restarts if no proof is found at some depth. While these approaches are no doubt interesting and provide theoretical guarantees, we did not find them to be necessary.

As a starting point, we can simply employ best-first search, expanding the leaf node that the policy considers most likely first. If a leaf node n was obtained by taking actions a_i from ancestor nodes n_i, select

$$\operatorname*{argmax}_{n} \prod_{i} \pi(a_i \mid n_i)$$

Unfortunately, this simple scheme is not likely to recover if π makes a confident misprediction, and is even incomplete if any node has an infinite chain of single children beneath, where $\pi\left(a_j \mid n_j\right) = 1$ by definition. To correct this issue we take inspiration from rlCoP's initial value heuristic, where tableaux are exponentially less likely to be closed the more open branches they have. We model this idea as an exponential

[1]no *value* function is employed: it is unclear how to adapt this to asynchronous evaluation, or how useful this would be in an asynchronous context

distribution

$$p(n) = \lambda e^{-\lambda g(n)}$$

where λ is a tunable parameter (set to 1 in our experiments here) and $g(n)$ is "number of open branches plus length of the active path". Including "length of the active path" in $g(n)$ makes little practical difference and makes the search procedure complete again. The two estimates are combined with a geometric mean so that nodes are selected by

$$\operatorname*{argmax}_n \sqrt{p(n) \prod_i \pi(a_i \mid n_i)}$$

In practice this expression is numerically-difficult to evaluate, but in logarithmic space it is better-behaved, producing the final expansion criterion

$$\operatorname*{argmax}_n \left[\left(\sum_i \ln \pi(a_i \mid n_i) \right) - \lambda g(n) \right]$$

7.4.2 Asynchronous Policy Evaluation

The proof search routine above assumes that the policy is evaluated synchronously for each expanded node. As discussed in the introductory sections, this has a significant impact on performance, particularly so for computationally-expensive policies. Instead, evaluation is deferred and a separate CPU thread continuously arranges for nodes to be processed on a GPU, selecting the first non-evaluated node on the path to the current best leaf node. $\pi(a \mid n)$ is set to 1 for nodes not yet evaluated: applying a uniform distribution does not work well in practice.

It does not appear to be particularly important that all nodes are evaluated for a learned policy to improve search, perhaps because guidance at the top of the search tree has a disproportionate effect. Asynchronous policy evaluation allows use of policies that are orders of magnitude slower than expansion steps without reduction in inference rate.

7.5 Learned Policy

7.4.1 describes biasing proof search with a learned policy, directing node expansions toward areas the policy considers useful. lazyCoP's policy is trained from its own proofs: at each non-trivial step in proofs the tableau, all available actions and the action that lead to a proof is recorded. This procedure produces a training set of tableaux and actions which we use to train a neural-network based policy to predict the correct action. Learning from existing system proofs in this way has advantages and disadvantages: each example's label is guaranteed to lead to a proof, but it is not necessarily the shortest proof, nor can the training data express preference amongst other actions.

We train and evaluate using the same set of problems from the MPTP translation [Urb06] of the Mizar Mathematical Library [GKN10] into first-order logic with equality. There are 32,524 problems in total in the *M40k* set; we use the *M2k* subset of 2003 problems in order to iterate quickly. All problems have a labelled conjecture which lazyCoP is able to exploit so that search proceeds backward from the conjecture. Problems from the *M2k* set come from related articles in Mizar, suggesting a degree of similarity which may be exploited by learning.

7.5.1 Representing Tableaux with Actions

Construction of directed graphs from tableaux is mostly typical for first-order representations as in Chapter 5, with a few modifications specific to tableaux-with-actions. First, while occurrences of identical symbols and variables share nodes in the graph, identical compound terms do not: this is because they may be rewritten by equalities separately in LPCT. Additionally, variable binding is non-destructive in LPCT to implement a form of basic paramodulation[2]. Bound variables therefore remain in place but have an outgoing edge attached to their binding. Tableaux may then be represented in the conventional graphical way, linking literal nodes with directed edges.

Encoding actions is then straightforward. lazyCoP implements a small number of different types of inference, such as reductions, extensions, reflexivity and so on. Each inference is attached to some terms or literals in the tableau to form a concrete action: rewriting $s = t$ in $L[p]$, for example, is represented as a node connected to the graph with an incoming edge from the s in $s = t$ and an outgoing edge from p, uniquely identifying the inference. t is involved by being the "other side" of the purely-syntactic subgraph forming $s = t$.

7.5.2 Network Architecture

We use a residual version of the directed graph networks introduced in Chapter 5 which allow the network to distinguish incoming and outgoing edges. The core of the network

[2]concisely, in this context *basicness* forbids paramodulating into terms introduced by instantiation or previous paramodulations — see [Pas08] and [Mos93] for details and [NR01] for wider context

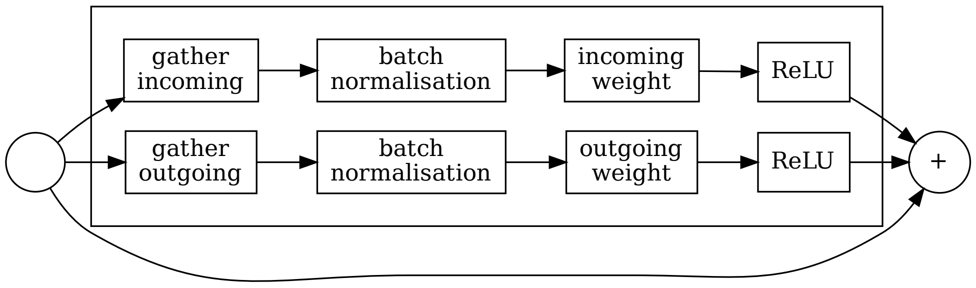


Figure 7.2: Residual block used in the network. Note disjoint parameters for incoming and outgoing edges, both linear and normalisation layers.

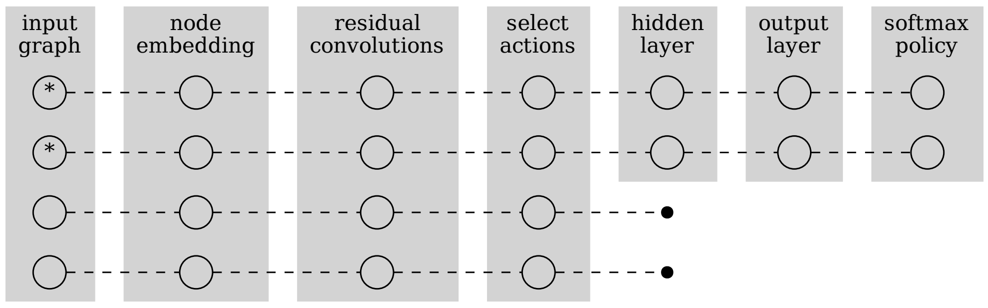


Figure 7.3: Network schematic. As there is no pooling of any kind, data is processed at the node level until action nodes marked (*) are projected out.

Table 7.1: Network and training hyper-parameters.

Parameter	Value	Parameter	Value
node dimension	64	initial learning rate	0.01
residual layers	24	cycle batches	2000
		batch size	64
		momentum	0.9
		weight decay	0.0001

is the residual block shown in Figure 7.2: this performs one round of message-passing from neighbouring nodes in the graph, treating incoming and outgoing edges separately before combining the results for the next layer. Batch normalisation [IS15] is inserted before the linear part of the convolution. The theoretical merits of this are unclear but it works well in practice. The complete network (Figure 7.3) is, in order:

Embedding. An embedding layer projects integer node labels into a real vector of the same size used in the convolutional layers. No symbol names are used here, as in e.g. ENIGMA Anonymous [JCO+20].

Convolution layers. Several residual blocks combine and transform feature maps from neighbouring nodes, producing in particular a real vector for each action node.

Action projection. The vector for each action node is projected out, all other nodes are discarded at this point.

Hidden layer. Each action vector is passed through a hidden layer.

Output layer. Computes a single output value for each action.

Rectified linear units are used as non-linearities throughout.

7.5.3 Training

Training such a network on limited training examples from early iterations is challenging due to its tendency to memorise the training set if sufficient parameters are available and underfit drastically if they are not. This is perhaps a good argument for feature-based learning rather than the approach we take here. However, the network can be made to train somewhat effectively by cosine annealing a high initial learning rate to 0 with “warm restarts” [LH17], repeating after a certain number of mini-batches.

This has two benefits: the regularising effect of high learning rates somewhat reduces overfitting, and the network also trains faster.

7.5.4 Integration and Optimisation

After the network is trained, network weight data are included as an admittedly-large compile-time constant into lazyCoP, which are then asynchronously uploaded onto the CUDA device at startup. The forward pass is re-implemented from scratch in CUDA [NBGS08], allowing a number of optimisations such as known array sizes, re-use of allocated buffers and the ability to profile for the specific workload. Additionally, batch normalisation layers' forward operation can be fused into the subsequent layer in this case, decreasing implementation complexity and increasing performance.

7.6 Experimental Results

We investigate two areas of practical interest: the effect of learned policy evaluations on inference rate, and whether this learning translates into improved performance on a training set of problems. Systems are only allowed 10 seconds of real time: this is relatively short, but a good approximation to real-world settings in which users of automatic "hammers" included in interactive theorem proving systems are unwilling to wait much longer than 30 seconds [MP08].

7.6.1 Inference Rates

There is no measurable decrease in inference rate when learned guidance is switched on. Occasionally the rate of inference even *improves*, perhaps due to guidance producing areas which are less productive or otherwise easier to explore. Running on `TOP001-1`, a non-theorem mid-sized topology problem from TPTP [SSY94], unguided lazyCoP achieves around 62,000 expansions per second for 10 seconds at the time of writing on desktop hardware. Guided, the system evaluates around 200 policies per second and reaches inference speeds in excess of 70,000 expansions per second.

7.6.2 Effect of Guidance

We train lazyCoP iteratively on *M2k* as described in §7.5, training each iteration on the proofs produced by all previous iterations. Iteration 0 does not have access to a learned

Table 7.2: Results from iterative training of lazyCoP's policy on *M2k*.

#	Proved	Cumulative	Steps
0	1,289	1,289	16,880
1	1,390	1,406	19,394
2	1,402	1,419	19,700
3	1,403	1,426	19,881

policy, iteration 1's policy is trained on iteration 0's proofs, iteration 2 on proofs from both iteration 0 and 1, etc. If there are two proofs for the same problem, the shorter proof is retained. The system is given 10 seconds of real time per problem, measured from program startup to the point of discovering a proof (but before output begins), and 16GB memory on a desktop machine[3]. Table 7.2 shows the number of problems solved by that iteration, the number of problems proved by all previous iterations, and the total number of proof steps for training available after the iteration finishes.

7.7 Summary

We have shown that even heavyweight neural guidance can be integrated without performance penalty, provided we have ability to backtrack and tolerate a temporary lack of learned guidance. Therefore, the challenge of performance penalties laid out in §1.2.3 is solved, albeit under some conditions. There are several directions that might improve results further:

Improving basic solver performance. Basic lazyCoP is clearly not yet competitive with state-of-the-art systems. There are many possible directions to achieve improved system performance, but we are particular interested in the integration of SAT/SMT solvers into backtracking systems for fast ground reasoning.

Scaling network and problem sets. It is very possible that a larger/deeper policy network would allow learning better policies. This requires either more careful tuning or a larger set of problems such as *M40k* to avoid overfitting excessively.

Parallelism. Implementing both parallel search and parallel evaluation on today's multicore machines would have a beneficial impact on performance. Parallel search allows exploiting remaining cores to search faster and is a clear win, the

[3]Intel® Core™ i7-6700 CPU @ 3.40GHz, NVIDIA® GeForce® GT 730

explicit search tree of lazyCoP allowing for several easy schemes to inject parallelism. Parallel evaluation does not inherently improve performance, but does ensure that the coprocessor is always kept busy: at present there are short pauses while the evaluation thread propagates the previous evaluation and prepares another input. Using multiple host threads also allows hiding latency from e.g. coprocessor cache misses, increasing overall throughput at the expense of the speed of single evaluation.

Incomplete modes. A system does not necessarily have to be complete to be useful. leanCoP includes a powerful but incomplete restricted-backtracking mode, for example. As well as e.g. restricted backtracking, lazyCoP could implement a strategy in which parts of the search tree are progressively discarded as resource limits draw nearer, in a similar way to Vampire's *limited resource strategy* [RV03]. We expect this to help with finding extremely long proofs.

Chapter 8

Concluding Remarks

There are challenges to practical learned guidance for automated reasoning on several fronts (§1.2), but this work has made some progress. Attempts are made to improve the relationship between training objective and resulting performance (§1.2.1), particularly in Chapter 6. Chapter 5 describes simple-but-effective methods for learning from syntactic data (§1.2.2), such as those used in automated reasoning systems. Such methods can improve the accuracy of learning directly from problem structure and therefore increase the possible impact of learned guidance in a variety of settings. Chapter 7 gives one solution to the practical problem of performance penalties from learned guidance (§1.2.3), although Chapters 3, 4, and 6 find their own ways around this issue. The tradeoff between unguided performance and the suitability for integration of learned guidance (§1.2.4) is explored throughout.

Several entirely-new ways to combine machine learning to automated reasoning are discussed. Chapter 3 introduces a method for scheduling running proof attempts in portfolio systems. Chapter 4 is a new take on the internal-guidance theme explored more conventionally in Chapter 7. Chapter 6 provides a new type of reinforcement-learning setting for ATP systems.

8.1 Retrospective

I had a lot of fun coming up with new ways to use old ideas, hopefully with practical applications. “Dynamic Strategy Priority” (Chapter 3) was in some ways technically easier than later work, and provided a good starting point. Continuing similar projects would have been much safer and perhaps more practical, but my dissatisfaction with the feature-based approach and the lack of decision-making power led me into dangerous, neural, internally-guided waters. LERNA (Chapter 4) is probably my most innovative work, although the sacrifices that were made in order to achieve its design meant that it was never particularly practical, despite my efforts to improve matters.

Looking back, the representation of proof state was a strong aspect of the LERNA work, although the learning itself was not systematic and the generation/distribution of learning data left something to be desired. Graphs were just becoming popular as a representation for logical data: previous approaches had used some variation on manual features, term walks, bag-of-words, or "it's just text/tokens". Syntax trees and tree networks were also used for logical data: these had a major advantage over other methods in that the representation was pre-parsed and did not lose any information: in principle an equivalent datum could be reconstructed from its representation.

Graphs were still a big step forward in my view: they had many of the benefits of trees but with more flexible information flow, the ability to represent shared substructures, and some types of invariance, such as commutativity and associativity of binary operators and binding up to α-equivalence. The machine learning community progressed fortuitously at about the same time: *graph convolutional networks* first became well-known around 2017. I had been experimenting with such networks and logical data since spring 2018, although without any real direction. In spring 2019 I had the good fortune to talk to Karel Chvalovský of *TopDownNet* fame, who let me take a look at his preprint[1]. The paper used a propositional dataset which seemed easy to get started with, and those investigations became Chapter 5.

Discussions with Martin Suda during the same trip produced the idea for Chapter 6, using reinforcement learning and an existing strong system to smooth out the delayed, discrete reward in theorem proving: either you've found a proof, or you haven't. Despite the theoretical niceties and minimal assumptions of the approach, results were mixed and I was running out of time for research in my program by spring 2020. I plan to return to this promising idea when time permits.

Finally, I'd wanted to return to the ideas of Chapter 4 for some time, but in a more practical context and in a more conventional way. Some concessions were made to allow for connection tableaux, and the result with some refinements is Chapter 7. Asynchronous evaluation, present in Chapter 4 and developed in Chapter 7, is a powerful mechanism allowing for guided proof search at full speed.

[1]subsequently published [Chv18]

8.2 Epilogue

The future for this intersection of machine learning and automated reasoning seems bright: methods for learning over syntactic data improve all the time, and new environments and new techniques to apply learning to theorem proving arrive almost constantly. Practical implementations also continue to surprise: it is no longer inconceivable that future systems must integrate some sort of learning to be competitive, and even the most recent edition of CASC, traditionally a failed hurdle for learning-assisted reasoning, seems to show that the grip of traditional systems is slowly loosening. The system that learns to write my Isabelle proofs for me is just around the corner...

Bibliography

[ABC$^+$16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[ACBF02] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[Agr95] Rajeev Agrawal. Sample mean based index policies with $O(\log n)$ regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27(4):1054–1078, 1995.

[AS97] Christopher G Atkeson and Juan Carlos Santamaria. A comparison of direct and model-based reinforcement learning. In *Proceedings of the International Conference on Robotics and Automation*, volume 4, pages 3557–3564. IEEE, 1997.

[ASN19] Rishabh Agarwal, Dale Schuurmans, and Mohammad Norouzi. An optimistic perspective on offline reinforcement learning. In *NeurIPS Deep Reinforcement Learning Workshop*, 2019.

[ATCF20] Ibrahim Abdelaziz, Veronika Thost, Maxwell Crouse, and Achille Fokoue. An experimental study of formula embeddings for automated theorem proving in first-order logic. *arXiv preprint arXiv:2002.00423*, 2020.

[Bar84] Hendrik P Barendregt. *The Lambda Calculus*. North-Holland Amsterdam, 1984.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[BEL01] Matthias Baaz, Uwe Egly, and Alexander Leitsch. Normal form transformations. In *Handbook of Automated Reasoning*, pages 273–334. Elsevier, 2001.

[BG01a] Leo Bachmair and Harold Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning*, pages 19–100. Elsevier, 2001.

[BG01b] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[BHP14] James P Bridge, Sean B Holden, and Lawrence C Paulson. Machine learning for first-order theorem proving. *Journal of Automated Reasoning*, 53(2):141–172, 2014.

[Bis06] Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[BKM17] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

[BLR$^+$19] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. HOList: An environment for machine learning of higher order logic theorem proving. In *International Conference on Machine Learning*, pages 454–463, 2019.

[BPM15] Samuel Bowman, Christopher Potts, and Christopher D Manning. Recursive neural networks can learn logical semantics. In *Proceedings of the 3rd workshop on Continuous Vector Space Models and their Compositionality*, pages 12–21, 2015.

[BR20] Ahmed Bhayat and Giles Reger. A combinator-based superposition calculus for higher-order logic. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *IJCAR 2020*, LNCS. Springer, 2020.

[BS20] Filip Bártek and Martin Suda. Learning precedences from simple symbol features. In *7th Workshop on Practical Aspects of Automated Reasoning*, 2020.

[BST$^+$10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, volume 13, page 14, 2010.

[CAC$^+$19] Maxwell Crouse, Ibrahim Abdelaziz, Cristina Cornelio, Veronika Thost, Lingfei Wu, Kenneth Forbus, and Achille Fokoue. Improving graph neural network representations of logical formulae with subgraph pooling. *arXiv preprint arXiv:1911.06904*, 2019.

[CBSS08] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. *AIIDE*, 8:216–217, 2008.

[CGCB15] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Gated feedback recurrent neural networks. In *International Conference on Machine Learning*, pages 2067–2075, 2015.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[Chv18] Karel Chvalovský. Top-down neural model for formulae. In *International Conference on Learning Representations*, 2018.

[CJSU19] Karel Chvalovský, Jan Jakubův, Martin Suda, and Josef Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In *International Conference on Automated Deduction*, pages 197–215. Springer, 2019.

[Dav01] Martin Davis. The early history of automated deduction: Dedicated to the memory of Hao Wang. In *Handbook of Automated Reasoning*, pages 3–15. Elsevier, 2001.

[DBV16] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.

[DEKB16] Maria Dimakopoulou, Stéphane Eranian, Nectarios Koziris, and Nicholas Bambos. Reliable and efficient performance monitoring in Linux. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 396–408. IEEE, 2016.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[Die19] Frederik Diehl. Edge contraction pooling for graph neural networks. *arXiv preprint arXiv:1905.10990*, 2019.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DRK$^+$14] Morgan Deters, Andrew Reynolds, Tim King, Clark Barrett, and Cesare Tinelli. A tour of CVC4: how it works, and how to use it. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 7–7. IEEE, 2014.

[DV96] Anatoli Degtyarev and Andrei Voronkov. The undecidability of simultaneous rigid E-unification. *Theoretical Computer Science*, 166(1-2):291–300, 1996.

[DV98] Anatoli Degtyarev and Andrei Voronkov. What you always wanted to know about rigid E-unification. *Journal of Automated Reasoning*, 20(1-2):47–80, 1998.

[EMH18] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

[ES11] Stefan Edelkamp and Stefan Schroedl. *Heuristic search: theory and applications*. Elsevier, 2011.

[ESA$^+$18] Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. Can neural networks understand logical entailment? *arXiv preprint arXiv:1802.08535*, 2018.

[ESS89] Wolfgang Ertel, Johann M Ph Schumann, and Christian B Suttner. Learning heuristics for a theorem prover using back propagation. In *5. Österreichische Artificial-Intelligence-Tagung*, pages 87–95. Springer, 1989.

[FB16] Michael Färber and Chad Brown. Internal guidance for Satallax. In *International Joint Conference on Automated Reasoning*, pages 349–361. Springer, 2016.

[FKU17] Michael Färber, Cezary Kaliszyk, and Josef Urban. Monte-Carlo tableau proof search. In *International Conference on Automated Deduction*, pages 563–579. Springer, 2017.

[FL19] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[GAA$^+$19] Xavier Glorot, Ankit Anand, Eser Aygün, Shibl Mourad, Pushmeet Kohli, and Doina Precup. Learning representations of logical formulae using graph neural networks. In *Neural Information Processing Systems*, Workshop on Graph Representation Learning, 2019.

[Gau20] Thibault Gauthier. Tree neural networks in HOL4. In *International Conference on Intelligent Computer Mathematics*, pages 278–283. Springer, 2020.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[GDM09] Yeting Ge and Leonardo De Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *International Conference on Computer Aided Verification*, pages 306–320. Springer, 2009.

[Gie06] Martin Giese. Saturation up to redundancy for tableau and sequent calculi. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 182–196. Springer, 2006.

[GJ19] Hongyang Gao and Shuiwang Ji. Graph U-nets. *arXiv preprint arXiv:1905.05178*, 2019.

[GJU19] Zarathustra Goertzel, Jan Jakubův, and Josef Urban. ENIGMAWatch: ProofWatch meets ENIGMA. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 374–388. Springer, 2019.

[GK03] Harald Ganzinger and Konstantin Korovin. New directions in instantiation-based theorem proving. In *18th Annual IEEE Symposium of Logic in Computer Science*, pages 55–64. IEEE, 2003.

[GKN10] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a nutshell. *Journal of Formalized Reasoning*, 3(2):153–245, 2010.

[GMR$^+$18] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. A survey of methods for explaining black box models. *ACM computing surveys (CSUR)*, 51(5):1–42, 2018.

[Goe20] Zarathustra Amadeus Goertzel. Make E smart again. In *International Joint Conference on Automated Reasoning*, pages 408–415. Springer, 2020.

[GS20] Bernhard Gleiss and Martin Suda. Layered clause selection for theory reasoning. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 402–409, Cham, 2020. Springer International Publishing.

[Häh01] Reiner Hähnle. Tableaux and related methods. In *Handbook of Automated Reasoning*, pages 101–178. Elsevier, 2001.

[Hal06] Thomas C Hales. Introduction to the Flyspeck project. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[HK19] Edvard K Holden and Konstantin Korovin. SMAC and XGBoost your theorem prover. In *Proc. 4th Conference on Artificial Intelligence and Theorem Proving (AITP 2019)*, pages 93–95, 2019.

[HLMW17] Gao Huang, Zhuang Liu, Laurens van der Maarten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[Hor51] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.

[HV11] Kryštof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In *International Conference on Automated Deduction*, pages 299–314. Springer, 2011.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[Irp18] Alex Irpan. Deep reinforcement learning doesn't work yet. `https://www.alexirpan.com/2018/02/14/rl-hard.html`, 2018.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[ISA$^+$16] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. DeepMath — deep sequence models for premise selection. In *Advances in Neural Information Processing Systems*, pages 2235–2243, 2016.

[JCO$^+$20] Jan Jakubův, Karel Chvalovský, Miroslav Olšák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In *International Joint Conference on Automated Reasoning*, pages 448–463. Springer, 2020.

[JU17] Jan Jakubův and Josef Urban. ENIGMA: efficient learning-based inference guiding machine. In *International Conference on Intelligent Computer Mathematics*, pages 292–302. Springer, 2017.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KBKU13] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. MaSh: machine learning for Sledgehammer. In *International Conference on Interactive Theorem Proving*, pages 35–50. Springer, 2013.

[KH17] Ekaterina Komendantskaya and Jónathan Heras. Proof mining with dependent types. In *International Conference on Intelligent Computer Mathematics*, pages 303–318. Springer, 2017.

[KK18] Andrzej Stanisław Kucik and Konstantin Korovin. Premise selection with neural networks and distributed representation of features. *arXiv preprint arXiv:1807.10268*, 2018.

[KLT$^+$12] Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. Overview and evaluation of premise crate techniques for large theory mathematics. In *International Joint Conference on Automated Reasoning*, pages 378–392. Springer, 2012.

[Kor08] Konstantin Korovin. iProver — an instantiation-based theorem prover for first-order logic. In *International Joint Conference on Automated Reasoning*, pages 292–298. Springer, 2008.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning*, pages 282–293. Springer, 2006.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[KSU13] Daniel Kühlwein, Stephan Schulz, and Josef Urban. E-MaLeS 1.1. In *International Conference on Automated Deduction*, pages 407–413. Springer, 2013.

[KSUV15] Cezary Kaliszyk, Stephan Schulz, Josef Urban, and Jiří Vyskočil. System description: ET 0.1. In *International Conference on Automated Deduction*, pages 389–398. Springer, 2015.

[KU14] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *Journal of Automated Reasoning*, 53(2):173–213, 2014.

[KU15a] Cezary Kaliszyk and Josef Urban. FEMaLeCoP: Fairly efficient machine learning connection prover. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 88–96. Springer, 2015.

[KU15b] Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *Journal of Automated Reasoning*, 55(3):245–256, 2015.

[KU15c] Daniel Kühlwein and Josef Urban. MaLeS: A framework for automatic tuning of automated theorem provers. *Journal of Automated Reasoning*, 55(2):91–116, 2015.

[KUMO18] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems*, pages 8822–8833, 2018.

[KUV15] Cezary Kaliszyk, Josef Urban, and Jiří Vyskočil. Efficient semantic features for automated reasoning over large theories. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[KW16a] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[KW16b] Thomas N Kipf and Max Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.

[LaV06] Steven M LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[LH17] Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with warm restarts. In *5th International Conference on Learning Representations*, 2017.

[Lig73] James Lighthill. Artificial intelligence: A general survey. In *Artificial Intelligence: a paper symposium*, pages 1–21. Science Research Council London, 1973.

[Lin69] Per Lindström. On extensions of elementary logic. *Theoria*, 35:1–11, 1969.

[LISK17] Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. *arXiv preprint arXiv:1701.06972*, 2017.

[LRSL20] Gil Lederman, Markus Rabe, Sanjit Seshia, and Edward A. Lee. Learning heuristics for quantified boolean formulas through reinforcement learning. In *International Conference on Learning Representations*, 2020.

[LS01] Reinhold Letz and Gernot Stenz. Model elimination and connection tableau procedures. In *Handbook of Automated Reasoning*, pages 2015–2114. Elsevier, 2001.

[LSBB92] Reinhold Letz, Johann Schumann, Stefan Bayerl, and Wolfgang Bibel. SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[Mar18] Gary Marcus. Deep learning: A critical appraisal. *arXiv preprint arXiv:1801.00631*, 2018.

[MBM$^+$16] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[MKS$^+$15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[Mos93] Max Moser. Improving transformation systems for general E-unification. In *International Conference on Rewriting Techniques and Applications*, pages 92–105. Springer, 1993.

[MP08] Jia Meng and Lawrence C Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.

[Mur12] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[MW97] William McCune and Larry Wos. Otter — the CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[NH10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted Boltzmann machines. In *International Conference on Machine Learning*, 2010.

[NR01] Robert Nieuwenheis and Albert Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning*, pages 371–444. Elsevier, 2001.

[OB03] Jens Otten and Wolfgang Bibel. leanCoP: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36(1-2):139–161, 2003.

[OKU19] Miroslav Olšák, Cezary Kaliszyk, and Josef Urban. Property invariant embedding for automated reasoning. *arXiv preprint arXiv:1911.12073*, 2019.

[OLLW18] Laurent Orseau, Levi Lelis, Tor Lattimore, and Théophane Weber. Single-agent policy tree search with guarantees. In *Advances in Neural Information Processing Systems*, pages 3201–3211, 2018.

[Ott10] Jens Otten. Restricting backtracking in connection calculi. *AI Communications*, 23(2-3):159–182, 2010.

[Ott11] Jens Otten. A non-clausal connection calculus. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 226–241. Springer, 2011.

[Ott16] Jens Otten. nanoCoP: A non-clausal connection prover. In *International Joint Conference on Automated Reasoning*, pages 302–312. Springer, 2016.

[Pas08] Andrei Paskevich. Connection tableaux with lazy paramodulation. *Journal of Automated Reasoning*, 40(2-3):179–194, 2008.

[PdMB08] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. Deciding effectively propositional logic with equality. Technical Report MSR-TR-2008-181, Microsoft Research, December 2008.

[PGM$^+$19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8026–8037, 2019.

[Pit01] Andrew M Pitts. Nominal logic: A first order theory of names and binding. In *International Symposium on Theoretical Aspects of Computer Software*, pages 219–242. Springer, 2001.

[PLR$^+$19] Aditya Paliwal, Sarah Loos, Markus Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. *arXiv preprint arXiv:1905.10006*, 2019.

[PLR$^+$20] Aditya Paliwal, Sarah M Loos, Markus N Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. In *AAAI*, pages 2967–2974, 2020.

[PU18] Bartosz Piotrowski and Josef Urban. ATPboost: Learning premise selection in binary setting with ATP feedback. In *International Joint Conference on Automated Reasoning*, pages 566–574. Springer, 2018.

[PU20] Bartosz Piotrowski and Josef Urban. Guiding inferences in connection tableau by recurrent neural networks. In *International Conference on Intelligent Computer Mathematics*, pages 309–314. Springer, 2020.

[PUBK19] Bartosz Piotrowski, Josef Urban, Chad E Brown, and Cezary Kaliszyk. Can neural networks learn symbolic rewriting? *arXiv preprint arXiv:1911.04873*, 2019.

[RBR20] Michael Rawson, Ahmed Bhayat, and Giles Reger. Reinforced external guidance for theorem provers. Presented at *7th Workshop on Practical Aspects of Automated Reasoning*, 2020.

[Reg19] Giles Reger. Boldly going where no prover has gone before. *arXiv preprint arXiv:1912.12958*, 2019.

[RN02] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 2002.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[RR18] Michael Rawson and Giles Reger. Dynamic strategy priority: Empower the strong and abandon the weak. In *6th Workshop on Practical Aspects of Automated Reasoning*, pages 58–71, 2018.

[RR19a] Michael Rawson and Giles Reger. A neurally-guided, parallel theorem prover. In *International Symposium on Frontiers of Combining Systems*, pages 40–56. Springer, 2019.

[RR19b] Michael Rawson and Giles Reger. Old or heavy? Decaying gracefully with age/weight shapes. In *International Conference on Automated Deduction*, pages 462–476. Springer, 2019.

[RR19c] Michael Rawson and Giles Reger. Towards an efficient architecture for intelligent theorem provers. *Conference on Artificial Intelligence and Theorem Proving*, 2019.

[RR20a] Michael Rawson and Giles Reger. Autoencoding TPTP. In *Conference on Artificial Intelligence and Theorem Proving*, 2020.

[RR20b] Michael Rawson and Giles Reger. Directed graph networks for logical reasoning. In *7th Workshop on Practical Aspects of Automated Reasoning*, pages 109–119, 2020.

[RR20c] Michael Rawson and Giles Reger. lazyCoP 0.1. EasyChair Preprint no. 3926, EasyChair, 2020.

[RR21] Michael Rawson and Giles Reger. lazyCoP: Lazy paramodulation meets neurally guided search. In *30th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, 2021.

[RRWN11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[RS17] Giles Reger and Martin Suda. Measuring progress to predict success: Can a good proof strategy be evolved? *Conference on Artificial Intelligence and Theorem Proving*, 2017.

[RS19] Giles Reger and Martin Suda. Can a failed strategy be useful? In *Conference on Artificial Intelligence and Theorem Proving*, 2019.

[RSV14] Giles Reger, Martin Suda, and Andrei Voronkov. The challenges of evaluating a new feature in Vampire. In *Vampire Workshop*, pages 70–74, 2014.

[RSV15] Giles Reger, Martin Suda, and Andrei Voronkov. Playing with AVATAR. In *International Conference on Automated Deduction*, pages 399–415. Springer, 2015.

[RSV16] Giles Reger, Martin Suda, and Andrei Voronkov. Finding finite models in multi-sorted first-order logic. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 323–341. Springer, 2016.

[RV01] Alan JA Robinson and Andrei Voronkov. *Handbook of Automated Reasoning*. Elsevier, 2001.

[RV03] Alexandre Riazanov and Andrei Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computation*, 36(1-2):101–115, 2003.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[Sch02] Stephan Schulz. E — a brainiac theorem prover. *AI Communications*, 15(2, 3):111–126, 2002.

[SE90] Christian Suttner and Wolfgang Ertel. Automatic acquisition of search guiding heuristics. In *International Conference on Automated Deduction*, pages 470–484. Springer, 1990.

[SGS15] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[SHM$^+$16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[SKB$^+$18] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018.

[SLB$^+$19] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *International Conference on Learning Representations*, 2019.

[SLJ$^+$15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[SM96] Geoff Sutcliffe and Stuart Melville. The practice of clausification in automatic theorem proving. *Computer Science and Information Systems*, 1996.

[SM16] Stephan Schulz and Martin Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In *International Joint Conference on Automated Reasoning*, pages 330–345. Springer, 2016.

[Smi17] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision*, pages 464–472. IEEE, 2017.

[Smu95] Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.

[SP07] Geoff Sutcliffe and Yury Puzis. SRASS — a semantic relevance axiom selection system. In *International Conference on Automated Deduction*, pages 295–310. Springer, 2007.

[SRV01] R Sekar, IV Ramakrishnan, and Andrei Voronkov. Term indexing. In *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier, 2001.

[SS94] Christian B Suttner and Johann Schumann. Parallel automated theorem proving. In *Machine Intelligence and Pattern Recognition*, volume 14, pages 209–257. Elsevier, 1994.

[SST14] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In *International Joint Conference on Automated Reasoning*, pages 367–373. Springer, 2014.

[SSY94] Geoff Sutcliffe, Christian Suttner, and Theodor Yemenis. The TPTP problem library. In *International Conference on Automated Deduction*, pages 252–266. Springer, 1994.

[ST19] Leslie N Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, volume 11006, page 1100612. International Society for Optics and Photonics, 2019.

[Sut08] Geoff Sutcliffe. CASC-J4 the 4th IJCAR ATP system competition. In *International Joint Conference on Automated Reasoning*, pages 457–458. Springer, 2008.

[Sut16] Geoff Sutcliffe. The CADE ATP system competition — CASC. *AI Magazine*, 37(2):99–101, 2016.

[SWK09] Yanmin Sun, Andrew KC Wong, and Mohamed S Kamel. Classification of imbalanced data: A review. *International Journal of Pattern Recognition and Artificial Intelligence*, 23(04):687–719, 2009.

[SWVDH$^+$08] Maarten PD Schadd, Mark HM Winands, H Jaap Van Den Herik, Guillaume MJ-B Chaslot, and Jos WHM Uiterwijk. Single-player Monte-Carlo tree search. In *International Conference on Computers and Games*, pages 1–12. Springer, 2008.

[TET12] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[TSM15] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *Association for Computational Linguists*, 2015.

[TUGH11] Evgeni Tsivtsivadze, Josef Urban, Herman Geuvers, and Tom Heskes. Semantic graph kernels for automated reasoning. In *Proceedings of the 2011 SIAM International Conference on Data Mining*, pages 795–803. SIAM, 2011.

[UJ20] Josef Urban and Jan Jakubův. First neural conjecturing datasets and experiments. In *International Conference on Intelligent Computer Mathematics*, pages 315–323. Springer, 2020.

[Urb06] Josef Urban. MPTP 0.2: Design, implementation, and initial experiments. *Journal of Automated Reasoning*, 37(1-2):21–43, 2006.

[Urb07] Josef Urban. MaLARea: a metasystem for automated reasoning in large theories. *ESARLT*, 257, 2007.

[USPV08] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jiří Vyskočil. MaLARea SG1 — machine learner for automated reasoning with semantic guidance. In *International Joint Conference on Automated Reasoning*, pages 441–456. Springer, 2008.

[UVŠ11] Josef Urban, Jiří Vyskočil, and Petr Štěpánek. MaLeCoP machine learning connection prover. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 263–277. Springer, 2011.

[Vor14] Andrei Voronkov. AVATAR: The architecture for first-order theorem provers. In *International Conference on Computer Aided Verification*, pages 696–710. Springer, 2014.

[VSP$^+$17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[Waa01] Arild Waaler. Connections in nonclassical logics. In *Handbook of Automated Reasoning*, pages 1487–1580. Elsevier, 2001.

[Wil92] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.

[WL99] Andreas Wolf and Reinhold Letz. Strategy parallelism in automated theorem proving. *International journal of pattern recognition and artificial intelligence*, 13(02):219–245, 1999.

[WTWD17] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems*, pages 2786–2796, 2017.

[XHHLB08] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research*, 32:565–606, 2008.

[XHLJ18] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[ZCM$^+$19] Zsolt Zombori, Adrián Csiszárik, Henryk Michalewski, Cezary Kaliszyk, and Josef Urban. Towards finding longer proofs. *arXiv preprint arXiv:1905.13100*, 2019.

[ZTXM19] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6(1):11, 2019.