Reinforced External Guidance for Theorem Provers

Michael Rawson, Ahmed Bhayat, and Giles Reger

University of Manchester, UK
michael@rawsons.uk,ahmed.bhayat@manchester.ac.uk,giles.reger@manchester.ac.uk

Abstract

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.

1 Introduction

We begin with the foibles of automated theorem proving (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. 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 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 huge 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.

2 Background

Many traditionally-distinct fields are combined for this work: conventional theorem-proving technology forms a setting for reinforcement learning, itself using a deep neural network as a sophisticated function approximator. Graph neural network methods are employed to control the highly-duplicated syntactic structure of logical data. We provide only brief background here, as these large areas of study are used merely as tools for the practical reasoning task.

2.1 Automated Theorem Proving

This work is deliberately agnostic to the particular theorem-proving technology used, but we use the archetypal superposition system Vampire [26] as a research vehicle. The system can prove many hard theorems in first-order logic, such as those found in the widely-used TPTP [31] problem set. Recently, the system has been extended to support problems expressed in a higher-order logic [2]. Vampire in its default mode follows a typical routine when trying to prove a conjecture from a set of axioms. First, the conjecture is negated and the problem converted to a set of formulae in clause normal form ("clausification"); if an empty clause can be derived from this set it amounts to proof by contradiction. The prover then explores a quickly-growing search space by applying generating inferences from these formulae, searching for the empty clause. Fair *clause selection* algorithms such as those used in Discount [4] are employed to search efficiently without losing completeness. Simplifying and/or deleting inferences may continuously eliminate redundant clauses in the search space and thus accelerate proof search.

2.2 ATP Folklore

Arcane knowledge has accumulated about the development of strong ATP systems [27]. One folklore maxim states that a proof is either found quickly, or not at all [23]. This, combined with the fact that ATP systems have many parameters which can be tuned individually, invites *portfolio modes* [24]: systems try many different combinations of settings ("strategies") to solve a problem, restarting for each attempt. Another states that presenting relevant lemmas to a system early is useful, as finding a key formula during search can produce a proof in short order.

2.3 Machine Learning and Theorem Proving

Despite the high degree of sophistication present in systems for automated reasoning, they still do not yet have the intuition of a human mathematician. Previous work has aimed to have systems learn from past experience proving theorems in a variety of ways. Premise selection [11] is the learning problem of selecting only those axioms relevant to a proof attempt, preventing extraneous formulae from slowing search. Selecting appropriate prover parameters for a given problem is another angle of attack, either statically [15] or mid-search [21].

A difficult but promising area of work has aimed to integrate some form of learned guidance inside a theorem prover [16], but here there is a trade-off between the performance of the theorem prover and the accuracy of guidance: the more expensive a learned heuristic is to evaluate, the slower the resulting search (even if it is better). Initial efforts have been made to apply reinforcement learning techniques to guiding theorem proving, notably by biasing Monte-Carlo tree search with learned approximations [12]. One problem observed by the reinforcement FLoP system [36] is the sparsity of reward available in theorem-proving environments.

2.4 Reinforcement Learning

Reinforcement learning (RL) deals with agents learning to take optimal actions in an environment that dispenses some reward [32]. 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 [18]. 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 [1] 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.

2.5 Graph Neural Networks

Directed graphs are a natural representation for logical formulae, with several desirable properties [34]. Graphs have proven difficult to process with available learning algorithms, but recent success on a number of benchmarks has popularised neural networks based upon learned graph convolutions [13]. Such networks pass (conceptual) messages between connected nodes in a graph, with information propagating at a maximum rate of one "hop" per layer. A typical graph convolution scenario assumes that each node has some real vector of size n attached, which we call *channels*. A simplified graph convolution function f is

$$H_{i+1} = f(H_i, A) = \sigma\left(AH_iW_i\right)$$

where H_i is the node feature matrix at layer i ($nodes \times channels$), A is the graph adjacency matrix, W_i is a learned weight matrix, and σ is an activation function such as ReLU [19]. Common modifications include normalising the adjacency matrix in some way and adding the identity matrix I to the adjacency matrix so that each node's own data can be included in the output of a convolution step [13]. In this work we use a directed adjacency matrix A and

$$f(H_i, A) = \sigma\left(\hat{D}^{-1}\hat{A}H_iW_i\right)$$

where $\hat{D}$ is the degree matrix of $\hat{A} = A + I$. This is a row-normalisation of the adjacency matrix: we did not find symmetric normalisation employed elsewhere to be useful in this case.

2.6 Residual Networks

The size and complexity of the graphs involved in our task suggest relatively deep networks are required to propagate sufficient information, and this is consistent with our experience. Deep networks are difficult to train in practice, motivating various network architectures with *skip connections* (such as highway networks [30], residual networks [7] or densely-connected networks [9]), in addition to general-purpose techniques such as batch normalisation [10]. We use a residual network for this work: the specific architecture was not found systematically.

3 Motivation and Learning Task

Previous work on applying RL techniques to automated theorem proving has encountered obstacles to practicality, both in the sparsity of reward in theorem proving (either a proof is

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 π
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 π
    $axioms \leftarrow axioms \setminus \{a_t\}$
    $selected \leftarrow selected \cup \{a_t\}$
    $actions \leftarrow axioms \cup infer(selected)$
    $s_t \leftarrow \mathcal{A}(axioms \cup selected)$
    compute r_t from s_t, then report (a_t, r_t)
  end
end

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. 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. Below, we codify the new environment.

3.1 Necessity of Reinforcement Learning

At first this might appear to be a supervised learning problem: simply try a number of deductions, 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.

3.2 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:

- 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

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.

3.3 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 amount 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` [5]. 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 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 without deductions s_0 is used as a baseline, so that the final reward function is

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

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.

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

3.4 Example

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

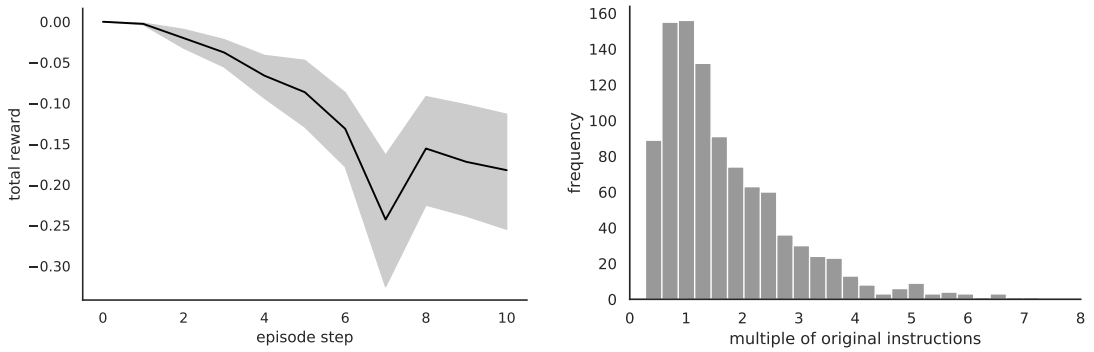
1. All men are mortal.
2. Socrates is a man.
3. Mortals eat olives.
4. Meletus is a man.
5. Mortals wear sandals.
6. Hemlock is unpleasant.

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

3.5 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. 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 during test-time: if we could always solve the problem, this work would not be necessary. Another difference is the ability to backtrack: while this is in principle possible with episode restarts in other environments, this is not typically exploited.

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

4 Approach

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.

4.1 Online Reinforcement Learning

We implemented a number of different conventional online RL algorithms in search of performance, but none were particularly successful. Significant engineering effort, manual tuning and

[1]The Vampire ATP and inference system described later is used for illustrative purposes here.

reducing the task to that of learning a policy for a single problem produced very few positive results. DQN [18] (value function estimation), REINFORCE [35] (policy gradients) and A3C [17] (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. 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 [33], presumably very different from this one.

4.2 Offline Reinforcement Learning

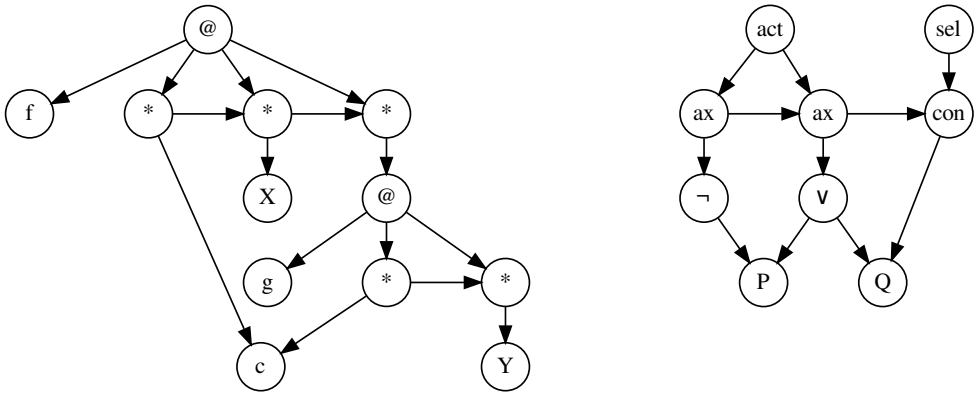
To reduce the amount of computational effort required, offline RL techniques are promising. Offline approaches are known to work well on heuristic search tasks such as two-player strategy games [28], but it is not clear how to adapt these techniques to our task, especially if expensive backtracking search 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 of this work 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: 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 4.6.

4.3 Data Generation

The 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 [14] similar to that employed in AlphaGo and rlCoP, but without random playouts. Playouts (and associated leaf value estimates) are replaced by an ATP invocation



(a) Fragment encoding $f\,(c, X, g\,(c, Y))$. (b) Complete state for a propositional problem.

Figure 3: Encoding states with directed graphs. Note auxiliary nodes and edges encoding function argument order, and "marker" nodes differentiating formula roles in the overall state.

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.

4.4 State Encoding

In order to represent the syntactic, duplicative structure of states and actions, we use a graphical representation. Formulae are first parsed into a syntax tree, and common sub-trees (such as shared symbols and compound terms) are merged to form a directed acyclic graph, preserving logically-distinct variables. Symbol and variable names are then discarded and replaced with generic "symbol" or "variable" labels, as distinct nodes encode distinct symbols after merging.

As noted in the motivation of FormulaNet [34], a limitation of directed graphs for this purpose is the lack of ordering between immediate children of a node: $f(a, b)$ encodes to the same graph as $f(b, a)$: this is remedied by adding auxiliary nodes and edges to represent each argument position. With this modification, the encoding becomes lossless, as a logically-equivalent state can be recovered from the encoding of that state. 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 3 shows both a state graph fragment encoding a particular first-order term, and a complete graph encoding initial state of a propositional problem: if $\neg P$, and $P \vee Q$, then Q.

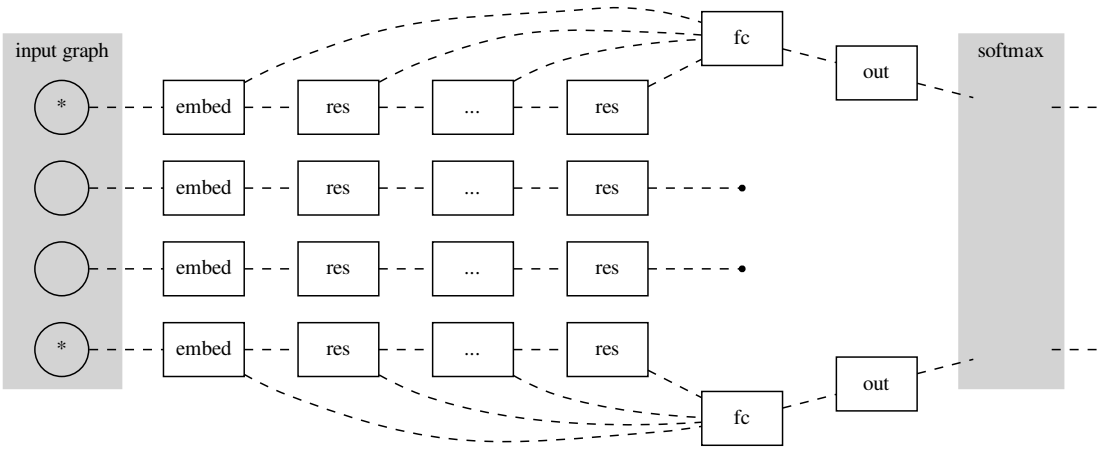


Figure 4: Overall network architecture. Nodes marked (*) are those that require an output value: that is, actions in the state encoding. Residual blocks use data from neighbouring nodes.

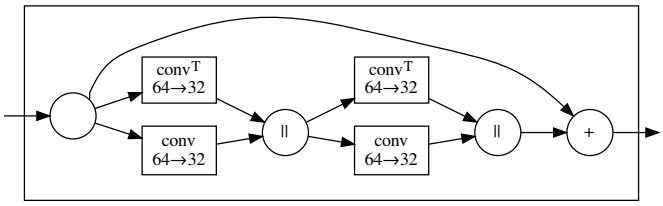


Figure 5: Residual module. || denotes feature concatenation and + element-wise addition.

4.5 Network Architecture

The neural network architecture (Figure 4) employed to process and learn from data of this kind is quite involved, and we present it only briefly here as a practical instrument. For motivation and details, refer to work on benchmark logical tasks [22] or more general work [7]. In order:

Embedding An embedding layer produces a real vector $x \in \mathbb{R}^n$ from each nodes' integer label.

Residual Modules A number of residual modules in sequence follow. Each module transforms the input twice and then adds the unmodified input. Each transformation is two convolutions which each halve the number of channels: one with the adjacency matrix, and one with its transpose, which are then immediately recombined — see Figure 5.

Action Projection After the convolution steps, only action nodes are retained.

Output Each nodes' final vector and all intermediate vectors are input into a typical output stage with a hidden layer. Using intermediate vectors (as in dense networks) improved network performance, but full *DenseNet*-style connectivity between all layers did not.

ReLU activation is used throughout the network except after embedding and output layers. Batch normalisation is used before both convolutional stages in residual modules in "full pre-activation" [8] configuration. Training used mini-batch stochastic gradient descent with Nesterov momentum, L_2 regularisation and a cyclic learning rate with fixed triangular policy [29].

Table 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

4.6 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 simplistic 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 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 Monte-Carlo methods as future work.

5 Experiments

We perform three experiments of increasing difficulty to exercise the environment, test our solution method and investigate practical uses for this work. The first is a pilot study both 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 show the practical use case for this work. An online repository[2] contains scripts and data that (i) were used to produce these experiments, and (ii) can be adapted for similar work.

5.1 Experimental Setup

We set up our experiments in the context of the Vampire system. A deterministic configuration 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 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

[2] https://github.com/MichaelRawson/paar20

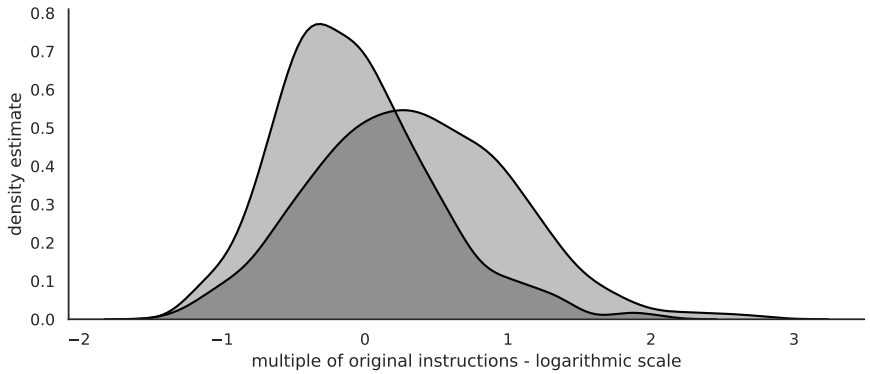


Figure 6: Kernel density estimates showing distribution of relative instructions required for Vampire to solve problems derived from `GRP001-1` with a learned (left) and random (right) policy. 1000 samples per distribution, Gaussian kernel, natural logarithmic scale for clarity.

the neural network. The neural network implementation and training uses the PyTorch [20] library for GPU acceleration. Tunable parameters for the system are listed in Table 1.

5.2 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. The best episode with learned policy produced a derived problem requiring only 28% of the instructions required to solve the unmodified problem.

5.3 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 [2]: 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 [3]: 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 furthermore 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. While this is not particularly encouraging, we report the result anyway to demonstrate the strengths and weaknesses of the approach. We expect that future work, especially tuning system options to produce a more graduated reward, will produce more concrete results: we have at least shown that modifying the input problem can significantly help the ATP system.

5.4 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 [23].

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 [25]. 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 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.

Table 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

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.

Results are shown in Table 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: compare the uniform random policy to the 10-second run.

6 Research Directions

This early work in a new area allows many possible developments. As well as the typical optimisation, further exploration and application to a wider range of settings (e.g. validating the architecture with a broader selection of ATP systems), we feel there are two fundamental areas that may produce disproportionate improvement in our approach.

1. Neural network architectures specific to theorem proving or to structured reasoning tasks in general are presently under-developed. In particular, the network we use here is a modestly-tweaked adaptation of a design originally intended to classify images, a task that almost certainly places very different demands on function approximators.

2. More fundamentally, the offline RL 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 similar performance.

In this work we focussed on specialising to a particular domain, leaving the implicit bias to a particular ATP system in the background. 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 [6] may even allow for automatic discovery of good ATP heuristics for manual implementation.

7 Conclusions

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 initial work, 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 a bespoke 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.

References

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

[2] 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.

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

[4] Jörg Denzinger, Martin Kronenburg, and Stephan Schulz. DISCOUNT—a distributed and learning equational prover. *Journal of Automated Reasoning*, 18(2):189–198, 1997.

[5] 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.

[6] 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.

[7] 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.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

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

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

[11] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Een, Francois Chollet, and Josef Urban. DeepMath — deep sequence models for premise selection. In *Advances in Neural Information Processing Systems*, pages 2235–2243, 2016.

[12] 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.

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

[14] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

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

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

[17] 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.

[18] 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.

[19] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[20] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. *NIPS Workshop*, 2017.

[21] 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.

[22] Michael Rawson and Giles Reger. Directed graph networks for logical entailment. Technical report, EasyChair, 2020.

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

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

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

[26] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2, 3):91–110, 2002.

[27] 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.

[28] 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, 2016.

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

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

[31] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated*

Reasoning, 43(4):337, 2009.

[32] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*. MIT press Cambridge, 1998.

[33] 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.

[34] 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.

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

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