# Proving Theorems using Incremental Learning and Hindsight Experience Replay

**Eser Aygün** [*1] **Ankit Anand** [*1] **Laurent Orseau** [*1] **Xavier Glorot** [1] **Stephen McAleer** [1 2] **Vlad Firoiu** [1]
**Lei Zhang** [1] **Doina Precup** [1 3 4] **Shibl Mourad** [*1]

## Abstract

Traditional automated theorem proving systems for first-order logic depend on speed-optimized search and many handcrafted heuristics designed to work over a wide range of domains. Machine learning approaches in the literature either depend on these traditional provers to bootstrap themselves, by leveraging these heuristics, or can struggle due to limited existing proof data. The latter issue can be explained by the lack of a smooth difficulty gradient in theorem proving datasets; large gaps in difficulty between different theorems can make training harder or even impossible. In this paper, we adapt the idea of hindsight experience replay from reinforcement learning to the automated theorem proving domain, so as to use the intermediate data generated during unsuccessful proof attempts. We build a first-order logic prover by disabling all the smart clause-scoring heuristics of the state-of-the-art E prover and replacing them with a clause-scoring neural network learned by using hindsight experience replay in an incremental learning setting. Clauses are represented as graphs and presented to transformer networks with spectral features. We show that provers trained in this way can outperform previous machine learning approaches and compete with the state of the art heuristic-based theorem prover E in its best configuration, on the popular benchmarks MPTP2078, M2k and Mizar40. The proofs generated by our algorithm are also almost always significantly shorter than E's proofs.

[*]Equal contribution [1]DeepMind, London, UK [2]Department of Information and Computer Science, University of California, Irvine, CA, USA [3]Mila - Quebec AI Institute, Montreal, QC, Canada [4]McGill University, Montreal, QC, Canada. Correspondence to: Eser Aygün <eser@deepmind.com>, Laurent Orseau <lorseau@deepmind.com>, Ankit Anand <anandank@deepmind.com>.

*Proceedings of the 39th International Conference on Machine Learning*, Baltimore, Maryland, USA, PMLR 162, 2022. Copyright 2022 by the author(s).

## 1. Introduction

*I believe that to achieve human-level performance on hard problems, theorem provers likewise must be equipped with soft knowledge, in particular soft knowledge automatically gained from previous proof experiences. I also suspect that this will be one of the most fruitful areas of research in automated theorem proving. And one of the hardest.* Schulz (2017, E's author)

Automated theorem proving (ATP) is an important tool both for assisting mathematicians in proving complex theorems as well as for areas such as integrated circuit design, and software and hardware verification (Leroy, 2009; Klein, 2009). Initial research on ATP dates back to the 1960s (*e.g.*, Robinson (1965); Knuth & Bendix (1970)) and was motivated partly by the fact that mathematics is a hallmark of human intelligence. However, despite significant research effort and progress, ATP systems are still far from human capabilities (Loos et al., 2017). The highest performing ATP systems (*e.g.*, Cruanes et al. (2019); Kovács & Voronkov (2013)) have been evolving for decades and have grown to use an increasing number of manually designed heuristics, mixed with some machine learning, to obtain a large number of search strategies that are tried sequentially or in parallel. Some recent works (Chvalovskỳ et al., 2019; Jakubův et al., 2020; Loos et al., 2017) build on top of these provers, using modern machine learning techniques to augment, select or prioritize their already existing heuristics, with some success. However, these machine-learning based provers usually require initial training data in the form of proofs, or positive and negative examples (provided by the high-performing existing provers) from which to bootstrap. Other recent works do not build on top of other provers, but still require existing proof examples as input (*e.g.*, Goertzel (2020); Polu & Sutskever (2020)). Such machine-learning-based ATP systems can struggle to solve difficult problems, partly due to the lack of problems of intermediate difficulty, which could provide training data of varying difficulty.

In this paper, we propose an approach which can build a strong theorem prover without relying on existing domainspecific heuristics or on prior input data (in the form of proofs) to prime the learning. We strive to design a learning methodology for ATP that allows a system to improve even when there are large gaps in the difficulty of given set of theorems. In particular, given a set of conjectures without proofs, our system trains itself, based on its own attempts and (dis)proves an increasing number of conjectures, an approach which can be viewed as a form of incremental learning.

Our approach is related in spirit to a particularly interesting recent system, rlCop (Kaliszyk et al., 2018; Zombori et al., 2020), which is based on the minimalistic leanCop 'tableau' theorem prover (Otten & Bibel, 2003) and uses reinforcement learning in proof attempts without relying on domain heuristics or proof data. It manages to surpass leanCop's performance—but falls short of better competitors such as E. This motivated the TRAIL algorithm Crouse et al. (2021); Abdelaziz et al. (2022), a prover trained using reinforcement learning that is built on top of a stripped version of E, and yields substantial improvements over previous results. Wu et al. (2021) proposed TacticZero, which applies reinforcement learning in an interactive theorem proving framework, focusing on the case when few human examples are available. The TacticZero agent not only identifies promising paths but also learns to discard bad proof states and restart from previously found good alternatives in the search tree.

However, all these previous approaches learn exclusively on *successful* proof attempts. When no new theorem can be proven, the learner may not be able to improve anymore and thus the system may not be able to obtain more training data. This could in principle happen even at the very start of training, if all the theorems available are too hard. In an attempt to tackle this issue, Aygün et al. (2020); Firoiu et al. (2021) proposed to create synthetic theorem generators based on the axioms used in actual conjectures, so as to provide a large initial training set of theorems with diverse difficulty. Unfortunately though, synthetic theorems can be very different from real theorems of interest to people, making transfer and generalization to target datasets difficult.

**Contributions:** In this paper, we propose a novel approach for building a theorem prover that can incrementally and continually improve itself, by applying the idea of hindsight experience replay (HER) (Andrychowicz et al., 2017) to ATP. Clauses reached during proof attempts (whether successful or not) are turned into goals in hindsight, producing a large amount of "auxiliary" theorems with proofs of varied difficulties for the learner, even when no theorem from the original set can be proven initially. This leads to a smoother learning regime and a constantly improving learner.

We evaluate our approach on two popular benchmarks: MPTP2078 (Alama et al., 2014) and M2k (Kaliszyk et al., 2018) and compare it both with TRAIL (Abdelaziz et al., 2022) as well as with E prover (Schulz, 2002; Cruanes et al., 2019), one of the leading heuristic provers. Our proposed approach substantially outperforms TRAIL (Abdelaziz et al., 2022) on both datasets, surpasses E in the *auto* configuration with a 100s time limit, and is competitive with E in the *autoschedule* configuration with a 7 days time limit. In addition, our approach finds shorter proofs than E in approximately 99.5% of cases. We perform an ablation experiment to highlight specifically the role of hindsight experience replay in the results. We also compare performance using multilayer perceptrons (MLP), graph neural networks (GNN) and transformers with spectral and sequential features, all combined with the same HER approach. Our best results are obtained using transformers with spectral features.

## 2. Methodology

In this section, we describe the basic search algorithm used by most of the traditional first-order automated theorem provers, explain how we integrate our method into one of these provers and finally, provide a detailed description of our overall incremental learning system. For the reader unfamiliar with first-order logic, we give a succinct primer in Appendix A.

**Given-clause algorithm.** Almost all of the powerful automated theorem provers for first-order logic, including E, use some variation of a *given-clause* search algorithm (Kovács & Voronkov, 2013; Cruanes et al., 2019; McCune & Wos, 1997). This type of algorithm works by continuously choosing a new *given clause* to expand, with the help of one or more priority queues, until an empty clause (*i.e.* contradiction) is reached. The given clause is combined according to various logical operations (like resolution, factoring, etc.; see Appendix A for more details) with previously chosen *active clauses* to generate more clauses, which are consequently added to the priority queues. Each priority queue depends on a scoring function for sorting the clauses. At every step, a queue is selected based on a schedule, which usually consists of a simple cycle through all queues and each queue occurs for a fixed number of pre-determined steps within in each cycle. For example, the simplest schedule could be round robin sampling of all queues where each cycle consists of a single occurrence of each queue.

The two most basic types of queues are *the FIFO queue* and *the clause weight queue*. The former keeps the clauses sorted from oldest to youngest, guaranteeing that every clause will be visited after some finite amount of time. The latter uses a simple linear function that combines the numbers of various elements in the clauses (such as literals, atoms, variables) to obtain a "weight" and sorts the clauses from lightest to heaviest. The idea is to prioritize lighter or smaller clauses which, empirically, helps in reaching the empty clause faster.

**Using machine learning to improve provers that depend on the given-clause algorithm.** There are many ways to incorporate machine learning into a prover that is based on the given-clause algorithm. One option is to replace the queues with a policy over clauses that has full control over the search (Crouse et al., 2021; Abdelaziz et al., 2022). Another option is to train a clause scoring function which merely provides an additional queue that can be added to any existing set of queues (Loos et al., 2017; Chvalovskỳ et al., 2019).

**Integrating our method into E.** We take the latter approach in this work. We train a classifier that predicts the probability of a clause appearing in the proof given a set of initial clauses and use the predictions of this classifier to construct a "learned queue". We integrate this queue into the popular open-source first-order prover E using remote procedure calls (in a fashion similar to Enigma (Jakubův & Urban, 2017)). This allows us to take advantage of the sophisticated logic engine in E.

E, however, is more than its logic engine. It comes preloaded with hundreds of thousands of lines of code for heuristics (optimized for certain datasets) which help E pick the right set of queues with the right set of ratios for the given problem. As our goal is to replace these complicated heuristics with a single machine learning system, when we evaluate our method, we use a simple, fixed queue structure: a FIFO queue for completeness, a basic clause weight queue for greedy search and a 'learned' queue for guided search.

## 2.1. Clause-scoring and hindsight experience replay

In order to perform clause-scoring, we use deep neural networks, which can be trained in many ways so as to find proofs faster. A method utilized by Loos et al. (2017) and Jakubův & Urban (2019) turns the scoring task into a classification task: a network is trained to predict whether the clause to be scored will appear in the proof or not. In other words, the probability predicted by an 'in-proofness' classifier is used as the score. To train, once a proof is found, the clauses that participate in the proof (*i.e.*, the ancestors of the empty clause) are considered to be positive examples, while all other generated clauses are taken as negative examples.[1] Then, given as input one such generated clause $x$ along with the input clauses $C_s$, the network must learn to predict whether $x$ is part of the (found) proof.

There are two main drawbacks to this approach. First, if all conjectures are too hard for the initially unoptimized prover, no proof is found and no positive examples are available, making supervised learning impossible. Second, since proofs are often small (often a few dozen steps), only few positive examples are generated. As the number of available conjectures is often small too, there is far too little data to train a modern high-capacity neural network. Moreover, for supervised learning to be successful, the conjectures that can be proven must be sufficiently diverse, so the learner

[1]These examples are technically not necessarily negative, as they may be part of another proof. But avoiding these examples during the search still helps the system to attribute more significance to the positive examples.

**Algorithm 1** Distributed incremental learning. `launch` starts a new process in parallel. For each conjecture an instance of UBS decides the sequence of time limits for solving attempts.

```
def main(conjectures):
 # Launch and connect learners, actors and manager
     with example buffer &  task queue
  example_buffer = create_example_buffer()
  task_queue = create_task_queue()
  learners = [for i = 1..10:
      launch learner(example_buffer)]
  for i = 1..1000: launch actor(task_queue,
      learners, example_buffer)
  actor_manager = launch actor_manager(conjectures,
      task_queue)
  wait for actor_manager to finish

def learner(example_buffer):
  repeat forever:
    # Sample a batch of examples and train the
        network.
    batch = sample_batch_uniformly(example_buffer)
    minimize_classification_loss(batch)  # we use
        cross-entropy

def actor(task_queue, learners, example_buffer)
  repeat forever:
    # Fetch a task and attempt to prove the
        conjecture.
    conjecture, time_limit = get_task(task_queue)
    learner = sample_uniformly(learners)
    run E on conjecture
      for at most time_limit seconds;
      obtain generated_clauses
    examples = sample_examples(generated_clauses) #
        see Alg. 2
    put_examples(example_buffer, examples)

def actor_manager(conjectures, task_queue):
  schedulers = []
  for conjecture in conjectures:
    schedulers[conjecture] = initialize_UBS() # see
        Section 2.2
  repeat until all conjectures have been proven:
    # Choose a random conjecture and enqueue it.
    conjecture = sample_uniformly(conjectures)
    scheduler = schedulers[conjecture]
    time_limit = get_next_time_limit(scheduler)
    put_task(task_queue, (conjecture, time_limit))
```

can steadily improve. Unfortunately, there is no guarantee that such a curriculum is available. If the difficulty suddenly jumps, the learner may be unable to improve further. These shortcomings arise because the learner only uses successful proofs, and all the unsuccessful proof attempts are discarded. In particular, the overwhelming majority of the generated clauses become negative examples, and need to be discarded to maintain a good balance with the positive examples.

To leverage the data generated in unsuccessful proof attempts, we adapt the concept of hindsight experience replay (HER) (Andrychowicz et al., 2017) from goal-conditioned reinforcement learning to theorem proving. The core idea of HER is to take any "unsuccessful" trajectory in a goal-based task and convert it into a successful one by treating the final state that happened to be reached as if it were the goal state, in hindsight. A deep network is then trained with this trajectory, by contextualizing the policy with this state instead of the original goal. This way, even in the absence of positive feedback, the network is still able to adapt to the *dataset*, if not to the goal, thus having a better chance to reach the goal on future tries.

Inspired by HER, we use the clauses generated during *any* proof attempt as additional conjectures, which we call *hindsight goals*, leading to a supply of positive and negative examples. Let $D$ be any non-input clause generated during the refutation attempt of $C_s$. We call $D$ a *hindsight goal*.[2] Then, the set $C_s \cup \{\neg D\}$ can be refuted. Furthermore, once the prover reaches $D$ starting from $C_s \cup \{\neg D\}$, only a few more resolution steps are necessary to reach the empty clause; that is, there exists a refutation proof of $C_s \cup \{\neg D\}$ where $D$ is an ancestor of the empty clause. Hence, we can use the ancestors of $D$ as positive examples for the negated conjecture and axioms $C_s \cup \{\neg D\}$. This generates a very large number of examples, allowing us to effectively train the neural network, even with only a few conjectures at hand.

Furthermore, to keep the network small, axioms are not provided as input to the scoring network Although the set of active clauses is an important factor in determining the usefulness of a clause, we ignore it in the network input to keep the network size smaller.

## 2.2. Incremental learning algorithm

Typical supervised learning ATP systems require a set of proofs (provided by other provers) to optimize their model (*e.g.*, Loos et al. (2017); Jakubův et al. (2020); Aygün et al. (2020)). Success is assessed by cross-validation. In contrast, we formulate ATP as an incremental learning problem—see in particular Orseau & Lelis (2021); Jabbari Arfaee et al. (2011). Given a pool of unproven conjectures, the objective is to prove as many as possible, even using multiple attempts, and ideally as quickly as possible. Hence, the learning system must bootstrap directly from initially-unproven conjectures, without any initial supervised training data. Success is assessed by the number of proven conjectures, and the time spent solving them. Hence, we do not need to split the set of conjectures into train/test/validate sets because, if the system overfits to the proofs of a subset of conjectures, it will not be able to prove more conjectures.

Our incremental learning system is described in Algorithm 1. Initially, all conjectures are unproven and the clause-scoring network is initialized randomly. At this stage, we have no information on how long it takes to prove a certain conjecture, or whether it can be proven at all. The prover attempts to prove all conjectures provided using a scheduler (described below), so as to vary time limits for each conjecture. This ensures that proofs for easy conjectures are obtained early, and the resulting positive and negative examples are then used to train the clause-scoring network. As the network learns, more conjectures can be proven, providing in turn more data, and so on. This incremental learning algorithm thus allows us to automatically build a capable prover for a given domain, starting from a basic prover that may not even be able to prove a single conjecture in the given set.

**Time scheduling.** All conjectures are attempted in parallel, each on a CPU. For each conjecture, we use the uniform budgeted scheduler (UBS) algorithm (Helmert et al., 2019, section 7) to further simulate running in (pseudo-)parallel the solver with varying time budgets, and restarting each time the budget is exhausted. In the terminology of UBS, we take $T(k, r) = 3r2^{k-1}$ in seconds, but we cap $k \leq k_{\max} = 10$. A UBS instance simulates on a single CPU running $k_{\max}$ restarting programs, by interleaving them: On a 'virtual' CPU of index $k \in \{1, \ldots, k_{\max}\}$, a program corresponds to running the prover for a budget of $3 \cdot 2^{k-1}$ seconds before restarting it for the same budget of time and so on; $r$ is the number of restarts. Hence, as the network learns, each conjecture is incrementally attempted with time budgets of varying sizes (3s, 6s, 12s, . . . , 3072s), using no more than one hour, while carefully balancing the cumulative time spent within each budget (Luby et al., 1993; Helmert et al., 2019). Once a proof has been found for a conjecture, the scheduler is not stopped, so as to continue searching for more (often shorter) proofs.

**Distributed implementation.** Our implementation consists of multiple actors running in parallel, a manager that distributes tasks to the actors using the time scheduling algorithm, and a task queue that handles manager-actors communication. We used ten learners training ten separate models to increase the diversity of the search without having to

[2]Note that, while the original version of HER (Andrychowicz et al., 2017) only uses the last reached state as a single hindsight goal, we use all intermediate clauses, providing many more data points.

**Algorithm 2** Example sampling algorithm.

```
def sample_examples(generated_clauses):
  # Estimate the number of examples that can be
      consumed by the learner
  target_num_examples =
    time_elapsed_since_last_attempt ×
        target_num_examples_per_second

  # Remove the input clauses
  hindsight_goals =
    generated_clauses \ input_clauses

  # Subsample the goals and the examples
  examples = []
  sizes = {tree_size(c) : c ∈ hindsight_goals}
  for size in sizes:
    size_goals = {c ∈ hindsight_goals :
        tree_size(c) == size}
    w_size = 1 / ln(size + e) - 1 / ln(size + e + 1)
          # See Appendix B
    num_examples = ceil(target_num_examples ×
        w_size)
    for _ in range(num_examples):
      goal = uniform_sample(size_goals) # pick
          hindsight goal of this size
      anc = ancestors(goal)
      examples += [positive_example(uniform_sample(
          anc), goal)]
      examples += [negative_example(uniform_sample(
          hindsight_goals \ anc), goal)]
  return examples
```

increase the number of actors. These learners are fed with training examples from the actors and use them to update their parameters of their clause-scoring networks. The actors share 10 inference servers with accelerators (Tensor Processing Units (Jouppi et al., 2017)) for doing fast inference (except in the ablation experiments where the actors do not use any accelerators).

**Subsampling hindsight goals and examples.** With HER, the number of available examples is actually far too large: if, after a proof attempt, $n$ clauses have been generated ($n$ may be in the thousands), not only can each clause be used as a hindsight goal, but there are about $n^2$ pairs of the form (positive example, hindsight goal), and far more negative examples. This suddenly puts us in a very data-rich regime, which contrasts with the data scarcity of learning only from complete proofs of the given conjecture. Hence, we need to *subsample* the examples in order to prevent overwhelming the learner (see Algorithm 2 in the appendix). To this end, we first estimate the number of examples the learner can consume per second before sampling. But there is an additional difficulty: the number of possible clauses is exponentially large in the `tree_size` (number of nodes in the clause tree) of the clause, while small clauses are likely more relevant since the empty clause (which is the true target) has size 0. Moreover, clauses can be rather large: a `tree_size` over 300 is quite common, and we observed some `tree_size` values over 6 000. To correct for this, we fix the proportion of positive and negative examples for each hindsight goal clause size, ensuring that small hindsight goal clauses are favoured, while allowing a diverse sample of large clauses, using a heavy-tail distribution $w_s$ described in Appendix B. Finally, all the positive and negative examples thus sampled are added to the training pool for the learners.

## 2.3. Representation

Our clause scoring network receives as input the clause to score, $x$, the hindsight goal clause, $g$, and a sequence of negated conjecture clauses $C_s$. Individual clauses are transformed into directed acyclic graphs (an example is depicted in Figure 1) with five different node types : clause, literal, atomic-term, variable-term or variable. First, there is a clause node, whose children are literal nodes, corresponding to all literals of the clause (each one is associated with a predicate). The children of literal nodes represent the arguments of the predicate; they are either variable-term nodes if the argument is a variable, or atomic-term nodes otherwise[3]. Children of atomic-term nodes follow the same description. Finally, each variable-term node is linked to a variable node, which has as many parents as there are instances of the corresponding variable in the clause.

To each node, we associate a feature vector composed of the following five components: (i) A one-hot vector of length 3, encoding if the node belongs to $x$, $g$ or a member of $C_s$. (ii) A one-hot vector of length 5 encoding the node type: clause, literal, atomic-term, variable-term or variable. (iii) A one-hot vector of length 2 encoding if the node belongs to a positive or negative literal (null vector for clause and variable nodes). (iv) A hash vector representing the predicate name or the function/constant name respectively for predicate or atomic-term nodes (null vector for other nodes). (v) A hash vector representing the predicate/function argument slot in which the term is present (null vector for clause, literal and variable nodes). Hash vectors are randomly sampled uniformly on the 64 dimensional unit hyper-sphere, using the name of the predicate, function or constant (and the argument position for slots) as seed.

The node feature vectors are projected into a 64-dimensional node embedding space using a linear layer that trains during learning. We use a Transformer encoder architecture (Vaswani et al., 2017) for the clause-scoring network, whose input is composed of the set of node embeddings in the current clause $x$, goal clause $g$ and conjecture clauses $C_s$, up to 128 nodes. For each node, we compute a spectral encoding vector representing its position in the clause

[3]A constant argument is equivalent with a function of arity 0.

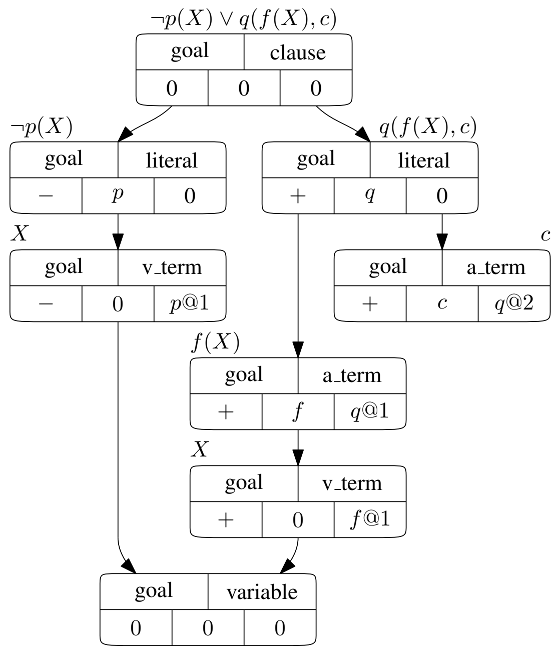

*Figure 1.* Clause graph of a goal clause. Each node has five features: clause type, node type, literal polarity, symbol hash and argument slot hash. The parts of formula corresponding to each node are shown outside of the nodes.

graph (Dwivedi & Bresson, 2020); this is given by the eigenvectors of the Laplacian matrix of the graph from which we keep only the 64 first dimensions, corresponding to the low frequency components. It replaces the traditional positional encoding of Transformers. Note that if there are more than 128 nodes in the set of clause graphs, we prioritize $x$, then $g$ and $C_s$. Within each graph, we order the nodes from top to bottom then left to right (e.g. the first nodes to be filtered out would be variable- or atomic-term nodes of the last conjecture clause). We only keep the transformer encoder output corresponding to the root node of the target clause and project it, using a linear layer, into a single logit, representing the probability that $x$ will be used to reach $g$ starting from $C_s$.

## 3. Experiments

To evaluate our approach, we use three popular datasets of the Mizar Mathematical Library (Grabowski et al., 2015) that has been used in previous works (Crouse et al., 2021; Kaliszyk et al., 2018; Kaliszyk & Urban, 2015a): (i) MPTP2078 (Alama et al., 2014) is a sample of larger Mizar datasets which is a good mixture of hard and easy theorems, (ii) M2k (Kaliszyk et al., 2018) is a relatively easier benchmark which contains theorems that have already been proven by at least one of the automated theorem provers in the past, and (iii) Mizar40 (Kaliszyk & Urban, 2015b) is a very large dataset of 57k theorems. The relative hardness of these datasets is also illustrated by the fact that the state-of-the-art E prover proves around 66% theorems in MPTP2078, around 40% theorems in Mizar40 while it achieves proof rate greater than 95% on M2k theorems.

We evaluate and compare our approach with both machine learning and heuristic based approaches on these two datasets. We compare our approach with E, considered a state-of-the-art heuristic based prover, in four configurations: (i) E in its default mode without any sophisticated heuristics and scheduling for 100s (referred to as E basic), (ii) E in *auto* mode for 100s (the mode that was used by Crouse et al. (2021) and Abdelaziz et al. (2022)[4]), (iii) E in *auto-schedule* mode (which outperforms *auto* mode consistently in our experience) for 100s, (iv) the best of different runs of E in *auto-schedule* mode with time limits of 100s, 1 hour, 1 day and 7 days (referred to as E best). It should be noted that we ran E best only with time limits up to 1 hour on Mizar40 due to resource constraints in proving 57k theorems.

We used E prover version 2.5 (Cruanes et al., 2019) in each of these configurations with a memory limit of 8192 GB.

We ran our incremental learning algorithm with hindsight experience replay (IL w/HER) for seven days on each dataset, using 1000 actors where each attempt was allowed a maximum duration of 100 seconds. The hyperparameters for all experiments are given in Appendix C. Every successful attempt that leads to a proof during training is logged, along with the time elapsed, the number of clauses generated, the length of the proof, and the proof itself. In order to show the importance of HER in achieving the results above, we ran the same experiments with incremental learning but without HER (IL w/o HER), by training the clause-scoring network using solely the data extracted from proofs found for the input problems. As another point of comparison, we include the results of TRAIL, which is a top performing learning method built on top of E prover, as reported in (Abdelaziz et al., 2022). Like our approach, TRAIL does not rely on E's heuristics and does not use additional input data from which to bootstrap, so it is directly comparable. Abdelaziz et al. (2022) also reported numbers for other learning provers that are similar in spirit, but since their performance is inferior to TRAIL, we do not include their reported numbers. We note that there are other machine-learning-based theorem provers, such as ENIGMA (Jakubův et al., 2020) and its variants, and the prover designed by Loos et al. (2017); but these provers rely heavily either on E's heuristics or on input proof data to bootstrap from, and thus fall in a different cat-

[4]The exact results reported by Abdelaziz et al. (2022) for E prover are significantly lower than what we obtained in our experiment. This could be attributed to a difference in the version of E prover, memory allocated or processor speed—the exact configuration details are not reported in their paper.

*Table 1.* Number of conjectures proven on MPTP2078, M2k and Mizar40.

| Domain | Conjectures | Heuristic Approaches | | | | ML Approaches | | |
|---|---|---|---|---|---|---|---|---|
| | | E basic (100s) | E auto (100s) | E auto-schedule (100s) | E best (100s–7d) | TRAIL | IL w/o HER | IL w/HER |
| MPTP2078 | 2 078 | 555 | 1 139 | 1 289 | **1 369** | 1 213 | 1 278 | **1 424** |
| M2k | 2 003 | 1 451 | 1 845 | 1 911 | **1 934** | 1 808 | 1 814 | **1 895** |
| Mizar40 | 57 880 | - | 17 346 | 21 693 | **23 173*** | - | 23 070 | **24 363** |

* On Mizar40, the highest time limit used for E best was 1 hour instead of 7 days due to resource constraints.

*Table 2.* Problems uniquely solved by one method but not the other (E best or IL w/HER) on both datasets.

| Domain | Only E best | Only IL w/HER |
|---|---|---|
| MPTP2078 | 58 | 113 |
| M2k | 59 | 20 |
| Mizar40 | 2 678 | 3 868 |

egory from ours, where the machine learning system based on a basic prover should bootstrap on its own.

**Conjectures proven.** Table 1 shows the number of conjectures proven by each of these approaches as well as the actual number of conjectures in each dataset. According to these results, IL w/HER outperforms all other provers that also use a 100 second time limit on all datasets except M2k, where E auto-schedule solves 16 more problems. It also manages to outperform E best, which is the best of eight different runs of E with various configurations some of which use time limits up to 7 days, on MPTP2078 as well as Mizar40, and comes as close as 1% on M2k. IL w/HER proves 2.6 times (1.3 times) as many problems as the E basic, which it is based on, on MPTP2078 (on M2k), improving its performance substantially purely via learning. Finally, IL w/HER significantly outperforms TRAIL on both MPTP2078 and M2k. Interestingly, since using HER is orthogonal to the methods used by TRAIL, one could hope that combining both approaches will lead to even better results—but we leave this as future work. As can be seen in these results, we do not observe important gains from learning on the M2k dataset. We believe that this is due to M2k being a set of theorems that can already be proven by ATPs. In other words, by construction, it consists of a subset of Mizar40 on which E already performs well. Additionally, TRAIL (Abdelaziz et al., 2022) reported their results only on MPTP2078 and M2k which are taken directly from the paper itself.

**Unique theorems proved by our approach.** Table 2 shows the number of theorems proven only by our approach and not by E best, and vice versa. IL w/HER proves 113, 20 and 3868 additional theorems on MPTP2078, M2k and Mizar40, respectively, which cannot proven by E in any of the configurations that we have tried. This suggests that IL w/HER can find strategies that are absent in E.

**Without hindsight.** In order to evaluate specifically the impact of using HER, we also report the performance of incremental learning alone which does not use any data from unsuccessful proof attempts. As seen in Table 1, IL w/o HER performed significantly worse, failing to prove 146 of the conjectures on MPTP2078, 81 of the conjectures on M2k and 1293 of the conjectures on Mizar40 that can be proven by IL w/HER. Without enough proofs of hard theorems from which to learn, IL w/o HER underperformed significantly on these domains compared to IL w/HER.

**Training vs. searching.** Figure 2 presents a comparison of the progress of E and the improvement of our systems as a survival plot over seven days of run time (wall-clock). Unlike E, which performed up to seven days of proof search per conjecture but has been under constant development for almost two decades, IL w/HER spent the same time to train provers based on a simple proof search algorithm from scratch, and ended up finding as many proofs as E.

**Quality of proofs.** We also looked at the individual proofs discovered by both systems. Incremental learning combined with the revisiting of previously proven conjectures allowed our system to discover shorter proofs continually. Figure 3 shows a scatter plot of the lengths of the shortest proofs found by E vs. found by IL w/HER for each theorem on MPTP2078 and M2k. The shortest proofs found by our system were consistently shorter than those found by E. Out of the 3140 conjectures proven by both systems, our proofs were shorter for 3131 conjectures (99.7%) whereas E's proofs were shorter for only 4 conjectures, with 5 proofs being of the same length.

**Speed of search.** E generated 5.72 times more clauses per seconds than our provers. We believe that the only way for our system to compete with E under these conditions is to find scoring functions that are much stronger than the numerous heuristics that have been built into E over time.

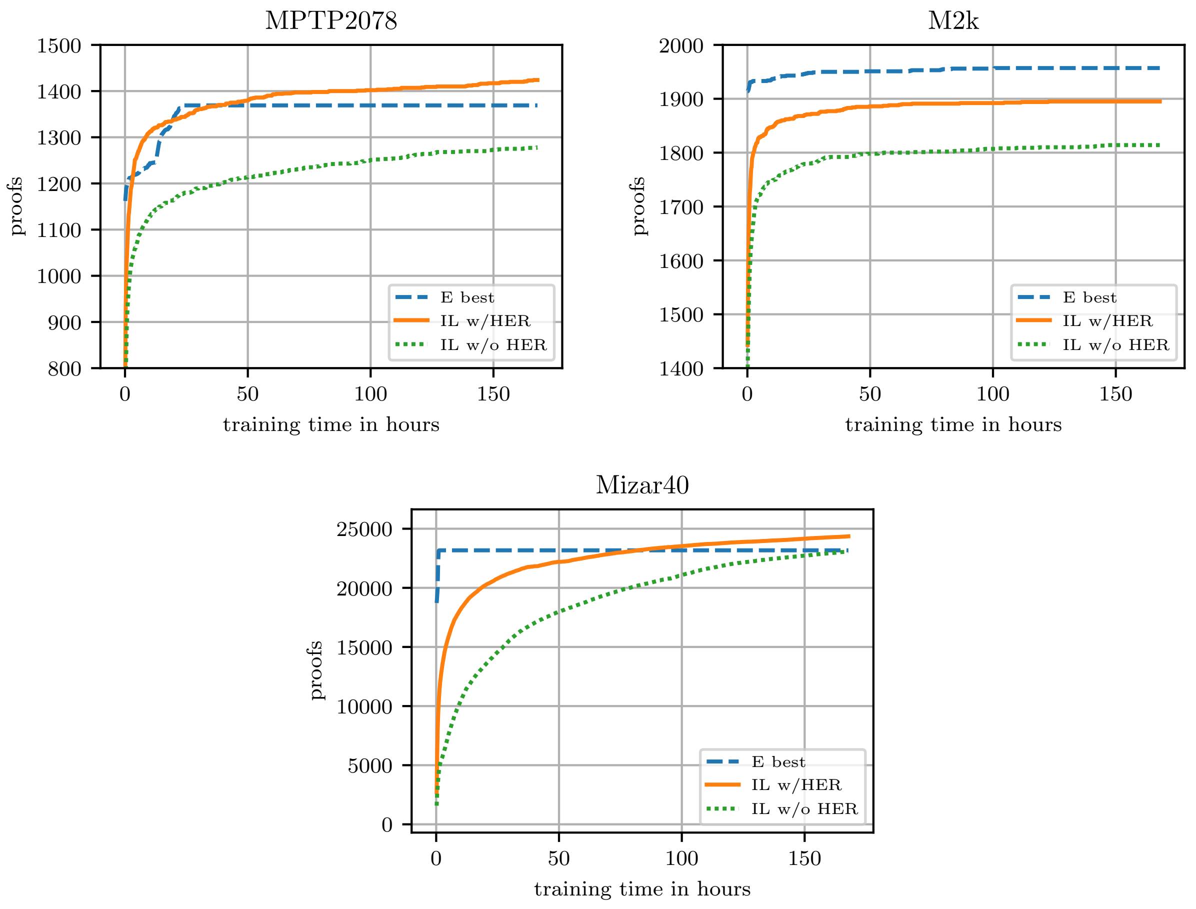

*Figure 2.* Survival plot showing the progress of E and incremental learning with and without hindsight experience replay over the course of seven days of training.

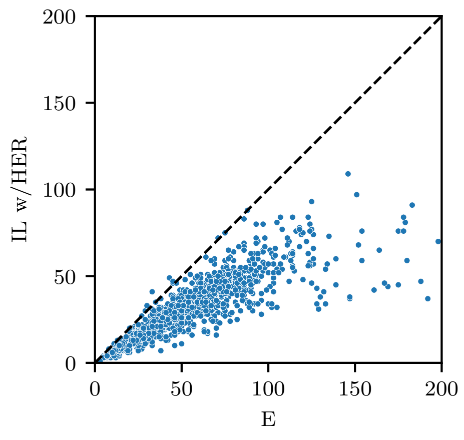

*Figure 3.* Scatter plot of the shortest proof lengths achieved by E vs. incremental learning with hindsight experience replay on the conjectures that can be proven by both in MPTP2078 and M2k.

**Comparison between different representations.** In order to understand the impact of the choice of network architecture on the results, we compared different neural networks trained with the proposed approach. We compared the spectral transformer representation described in Section 2.3 with MLP (based on manually defined features), Graph Neural Networks (GNNs) and a sequential text-based representation of the logical formulae which is used in a standard sequential transformer. On these experiments, we did not use accelerated actors as GNNs were not benefitting from acceleration as much as the other architectures, so the results for spectral transformers are different than results in Table 1. We used the same graph structure as the spectral transformer described in Sec. 2.3 for GNNs but added an additional root node at the top that connects the target clause with the negated conjecture clauses in order to allow message passing between different clauses. The details of each representation along with the hyperparameters used are described in Appendix C. Table 3 shows the conjectures solved by using different representations trained with IL w/HER using 1000 actors. We observe that GNNs outperform MLPs

*Table 3.* Comparison of different neural network architectures in IL w/HER on MPTP2078 and M2k. All the experiments in this table are performed without accelerators for a fair comparison which lead to a difference in performance of spectral transformers here compared to Table 1.

| Domain | Conjectures | MLP | GNN | Sequential transformer | Spectral transformer |
|---|---|---|---|---|---|
| MPTP2078 | 2 078 | 1 049 | 1 221 | 1 076 | **1 353** |
| M2k | 2 003 | 1 772 | 1 756 | 1 704 | **1 861** |

but fall short of the spectral transformer in our implementation on the MPTP2078 dataset. It should be noted that there are multiple ways to represent logical formulae as graphs, but we confine ourselves within the representation which is closest to spectral transformers. A detailed investigation of other graph representations proposed in the literature in combination with IL w/HER is left for future work. Also, we observe that spectral transformers outperform sequential transformers significantly in all our experiments. This can be attributed to the fact that spectral transformers capture graphical structure, and hence exploit logical invariances in formulae, in contrast to sequential transformers which treat these formulae as text.

## 4. Discussion

In this work, we proposed a method for training a first order logic theorem prover given a set of conjectures without proofs. Our proposed method starts from a very simple given-clause algorithm and uses hindsight experience replay (HER) to learn how to prove an increasing number of conjectures in an incremental fashion. We train a transformer network using spectral features in order to provide a useful scoring function for the prover. Our approach significantly outperforms TRAIL (Abdelaziz et al., 2022) on the MPTP2078 and M2k datasets, and surpasses the state-of-the-art heuristic-based prover E in its best setting on the MPTP2078 and Mizar40 datasets. Furthermore, our proofs are almost always shorter than those generated by E.

An obvious area of improvement is to provide more side information to the transformer network, so as to make decisions more context-aware. Also, since HER is a generic data-augmentation scheme, it appears plausible that a similar approach could be used for other kinds of logic and proof searchers, and it would be interesting to adapt our methodology to higher-order logic in particular.

## Acknowlegements

We would like to thank Tara Thomas and Natalie Lambert for helping us in planning our research goals and keeping our research focused during the project. We are also grateful to Koray Kavukcuoglu, Oriol Vinyals, Alex Davies and Pushmeet Kohli for the helpful discussions during the course of this work. We would also like to thank Dimitrios Vytiniotis for providing suggestions which helped in improving the manuscript substantially.

## References

Abdelaziz, I., Crouse, M., Makni, B., Austel, V., Cornelio, C., Ikbal, S., Kapanipathi, P., Makondo, N., Srinivas, K., Witbrock, M., and Fokoue, A. Learning to guide a saturation-based theorem prover. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 2022. ISSN 1939-3539.

Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., and Urban, J. Premise selection for mathematics by corpus analysis and kernel methods. *Journal of Automated Reasoning*, 52(2):191–213, 2014.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Pieter Abbeel, O., and Zaremba, W. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Aygün, E., Ahmed, Z., Anand, A., Firoiu, V., Glorot, X., Orseau, L., Precup, D., and Mourad, S. Learning to prove from synthetic theorems. *arXiv preprint arXiv:2006.11259*, 2020.

Chvalovskỳ, K., Jakubův, J., Suda, M., and Urban, J. Enigma-ng: efficient neural and gradient-boosted inference guidance for e. In *International Conference on Automated Deduction*, pp. 197–215. Springer, 2019.

Crouse, M., Abdelaziz, I., Makni, B., Whitehead, S., Cornelio, C., Kapanipathi, P., Srinivas, K., Thost, V., Witbrock, M., and Fokoue, A. A deep reinforcement learning approach to first-order logic theorem proving. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(7): 6279–6287, 2021.

Cruanes, S., Schulz, S., and Vukmirović, P. Faster, Higher, Stronger: E 2.3. In *TACAS 2019*, volume 11716 of *LNAI*, pp. 495–507, April 2019.

Dwivedi, V. P. and Bresson, X. A generalization of transformer networks to graphs. *CoRR*, abs/2012.09699, 2020.

Elias, P. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21 (2):194–203, 1975.

Firoiu, V., Aygün, E., Anand, A., Ahmed, Z., Glorot, X., Orseau, L., Zhang, L. M., Precup, D., and Mourad, S. Training a first-order theorem prover from synthetic data. *CoRR*, abs/2103.03798, 2021.

Fitting, M. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

Goertzel, Z. A. Make E smart again (short paper). In *Automated Reasoning*, pp. 408–415. Springer International Publishing, 2020.

Grabowski, A., Korniłowicz, A., and Naumowicz, A. Four decades of mizar. *Journal of Automated Reasoning*, 55 (3):191–198, 2015.

Helmert, M., Lattimore, T., Lelis, L. H. S., Orseau, L., and Sturtevant, N. R. Iterative budgeted exponential search. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pp. 1249–1257. AAAI Press, 2019.

Jabbari Arfaee, S., Zilles, S., and Holte, R. C. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

Jakubův, J. and Urban, J. Enigma: efficient learning-based inference guiding machine. In *International Conference on Intelligent Computer Mathematics*, pp. 292–302. Springer, 2017.

Jakubův, J. and Urban, J. Hammering Mizar by Learning Clause Guidance (Short Paper). In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141, pp. 34:1–34:8. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

Jakubův, J., Chvalovskỳ, K., Olšák, M., Piotrowski, B., Suda, M., and Urban, J. Enigma anonymous: Symbol-independent inference guiding machine (system description). In *International Joint Conference on Automated Reasoning*, pp. 448–463. Springer, 2020.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

Kaliszyk, C. and Urban, J. Learning-assisted theorem proving with millions of lemmas. *Journal of symbolic computation*, 69:109–128, 2015a.

Kaliszyk, C. and Urban, J. Mizar 40 for mizar 40. *Journal of Automated Reasoning*, 55(3):245–256, 2015b.

Kaliszyk, C., Urban, J., Michalewski, H., and Olšák, M. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

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

Klein, G. Operating system verification—an overview. *Sadhana*, 34(1):27–69, 2009.

Knuth, D. E. and Bendix, P. B. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pp. 263–297. Pergamon, 1970.

Kovács, L. and Voronkov, A. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pp. 1–35. Springer, 2013.

Leroy, X. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

Loos, S., Irving, G., Szegedy, C., and Kaliszyk, C. Deep network guided proof search. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pp. 85–105, 2017.

Luby, M., Sinclair, A., and Zuckerman, D. Optimal speedup of Las Vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180, September 1993.

McCune, W. and Wos, L. Otter - the CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2): 211–220, 1997.

Orseau, L. and Lelis, L. H. S. Policy-guided heuristic search with guarantees. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(14):12382–12390, May 2021.

Otten, J. and Bibel, W. leancop: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36 (1):139–161, 2003.

Polu, S. and Sutskever, I. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

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

Schulz, S. E–a brainiac theorem prover. *AI Communications*, 15(2, 3):111–126, 2002.

Schulz, S. We know (nearly) nothing! But can we learn? In *ARCADE 2017. 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements*, volume 51 of *EPiC Series in Computing*, pp. 29–32. EasyChair, 2017.

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

Wu, M., Norrish, M., Walder, C., and Dezfouli, A. Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning. *arXiv preprint arXiv:2102.09756*, 2021.

Zombori, Z., Urban, J., and Brown, C. E. Prolog technology reinforcement learning prover. In *International Joint Conference on Automated Reasoning*, pp. 489–507. Springer, 2020.

## A. A quick primer on first-order logic and resolution calculus

First-order logic (FOL) is a formal language used to express mathematical or logical statements. We give a brief introduction to first-order logic here. For more information, see (Fitting, 2012). Any statement expressed in first-order logic is called first-order logic formula. For example the statement: "for all people $X, Y$ and $Z$, if $X$ is a parent of $Y$ and $Y$ is a parent of $Z$, then $X$ is grandparent of $Z$" can be expressed as the FOL formula: $\forall X, \forall Y, \forall Z, \mathrm{parent}(X, Y) \wedge \mathrm{parent}(Y, Z) \Rightarrow \mathrm{grandparent}(X, Z)$. Here, $X, Y, Z$ are *variables*, and parent and grandparent are *predicates*. We will only consider FOL formulas expressed in Conjunctive Normal Form (CNF), as a conjunction ($\wedge$) of *clauses*, in which all variables are implicitly universally quantified ($\forall$). Consider the following CNF formula:

$$\underbrace{(\neg\mathrm{parent}(X, Y) \vee \neg\mathrm{parent}(Y, Z) \vee \mathrm{grandparent}(X, Z))}_{C_1}$$
$$\wedge\, \underbrace{\mathrm{parent}(\mathrm{alice}, \mathrm{bob})}_{C_2} \wedge \underbrace{\mathrm{parent}(\mathrm{bob}, \mathrm{charlie})}_{C_3}$$

A *clause* is a disjunction ($\vee$) of a number of *literals*; we will also consider clauses to be sets of literals to simplify the notation. $C_1, C_2$ and $C_3$ are clauses. Note that $C_1$ is equivalent to the first FOL formula example above, expressed as a clause. A *literal* is an *atom*, possibly preceded by the negation $\neg$, in which case it is a negative literal (positive otherwise). For example, $\mathrm{parent}(X, Y)$ and $\neg\mathrm{parent}(Y, Z)$ are literals. An *atom* is a *predicate* name of arity $n \in \mathbb{N}$ followed by a list of $n$ *terms*. A *term* is either a *function* name of associated *arity* $n \in \mathbb{N}$ followed by a list of $n$ terms, or a *constant* (such as alice, bob), or a *variable*. We assume that two different clauses of the same formula cannot share variables. A clause $C$ is a *tautology* if $C$ contains both a literal $\ell$ and its negation $\neg\ell$. Tautologies can be safely removed from a CNF formula without changing its truth value. The `tree_size` of a clause is the number is the number of nodes when the clause is represented as a tree of terms. For example, `tree_size`$(\mathrm{parent}(X, \mathrm{bob}))$ is 3.

A *substitution* is a set $\{V_1 \rightsquigarrow t_1, V_2 \rightsquigarrow t_2, \dots\}$ where $V_1, V_2, \dots$ are variables and $t_1, t_2, \dots$ are terms. The set of variables $\{V_1, V_2, \dots\}$ is the *domain* of the substitution. The *application* $\sigma(C)$ of a substitution $\sigma$ to a clause $C$ (or also to a single literal) results in a new clause $C' = \sigma(C)$ where all occurrences of the variables (of the domain of $\sigma$) in $C$ are replaced with their corresponding terms according to the substitution $\sigma$. For example, if $C = \mathrm{parent}(X, Y)$ and $\sigma = \{X \rightsquigarrow \mathrm{alice}, Y \rightsquigarrow Z\}$ then $\sigma(C) = \mathrm{parent}(\mathrm{alice}, Z)$.

Two literals $\ell_1$ and $\ell_2$ can be *unified* if there exists a substitution $\sigma$ such that applying it to both literals result in the same literal, that is, $\sigma(\ell_1) = \sigma(\ell_2)$. In such a case, the most general unifier $\mathrm{mgu}(\ell_1, \ell_2)$ of $\ell_1$ and $\ell_2$ is the smallest substitution that unifies the two literals, and it is unique (up to a renaming of the variables). For example, the most general unifier of $C = \mathrm{parent}(X, Y)$ and $C' = \mathrm{parent}(\mathrm{alice}, Z)$ is $\{X \rightsquigarrow \mathrm{alice}, Y \rightsquigarrow Z\}$, such that $\sigma(C) = \sigma(C') = \mathrm{parent}(\mathrm{alice}, Z)$.

A clause $C_1$ *subsumes* a clause $C_2$ if there exists a substitution $\sigma$ such that $\sigma(C_1) \subseteq C_2$ where the clauses are considered to be sets of literals.[5] For example the clause $p(X, a)$ subsumes the clause $p(b, a) \vee p(c, a)$, with $\sigma = \{X \rightsquigarrow b\}$ (or also with $\sigma = \{X \rightsquigarrow c\}$) as $\sigma(\{p(X, a)\}) \subseteq \{p(b, a), p(c, a)\}$.

The clause $C'$ is a *factor* of a clause $C$ if there exist a substitution $\sigma$ and two literals $\ell$ and $\ell'$ in $C$ such that $\sigma = \mathrm{mgu}(\ell, \ell')$ and $C' = \sigma(C \setminus \{\ell\})$. The operation $\mathrm{factoring}(C)$ returns the set of all factors (unique up to renaming of the variables) of $C$, with 'fresh' (never used) variables. For example, we can factor $C_1$ on its first two literals to obtain the clause $\neg\mathrm{parent}(Y', Y') \vee \mathrm{grandparent}(Y', Y')$ with the substitution $\{X \rightsquigarrow Y, Z \rightsquigarrow Y\}$. A clause is the sole *parent* of its factors.

The clause $C''$ is a *resolvent* of two clauses $C$ and $C'$ if there exist a substitution $\sigma$, a positive literal $\ell$ in $C$ and a negative literal $\ell'$ in $C'$ such that $\sigma = \mathrm{mgu}(\ell, \ell')$ and $C'' = \sigma(C \setminus \{\ell\} \cup C' \setminus \{\ell'\})$. The operation $\mathrm{resolution}(C, C')$ produces the set of all possible resolvents of $C$ and $C'$ (Robinson, 1965), with 'fresh' variables. For example, the resolvents of $C_1$ and $C_2$ are $\{\neg\mathrm{parent}(\mathrm{bob}, Z') \vee \mathrm{grandparent}(\mathrm{alice}, Z'), \neg\mathrm{parent}(X', \mathrm{alice}) \vee \mathrm{grandparent}(X', \mathrm{bob})\}$. The clauses $C$ and $C'$ are called the *parents* of $C''$. The *ancestors* of a clause are its parents, the parents of its parents and so on.

Together, resolution and factoring are sound and also sufficient for *refutation completeness*, that is, they can only produce clauses that are logically implied by the initial clauses, and if the empty clause is logically implied by the initial clauses, then the empty clause can be also be produced by a sequence of resolution and factoring operations starting from the initial clauses. For example, suppose that we want to prove that alice is the grandparent of someone, that is, that $\mathrm{grandparent}(\mathrm{alice}, A)$ can be satisfied for some value of $A$. Then we negate this conjecture to obtain the clause (implicitly universally quantified over $A$) with a single literal: $C_4 = \neg\mathrm{grandparent}(\mathrm{alice}, A)$ and we attempt to refute the CNF formula $C_1 \wedge \dots \wedge C_4$, that is, to reach the empty clause using resolution and factoring. First we can resolve $C_4$ with $C_1$ to obtain $C_5 = \neg\mathrm{parent}(\mathrm{alice}, Y') \vee \neg\mathrm{parent}(Y', A')$. Then we can resolve $C_5$ with $C_2$ to obtain $C_6 = \neg\mathrm{parent}(\mathrm{bob}, A'')$ and fi-

[5] We assume that syntactic duplicate literals are removed automatically.

nally we can resolve $C_6$ with $C_3$ to obtain the empty clause, which means that indeed alice is the grandparent of someone.

## B. A heavy-tail distribution over the integers

To ensure a preference for smaller clauses, while ensuring some diversity of the clause sizes, we use the following heavy-tail distribution for $s \in \{0, 1, 2 \ldots\}$:

$$w_s = 1/\ln(s+e) - 1/\ln(s+e+1)\,.$$

These weights constitute a telescoping series and ensure that $\sum_{s=0}^{\infty} w_s = 1$ while, using $\ln(1+1/x) \geq 1/(x+1)$,

$$\begin{aligned} w_s &= \frac{\ln\left(1+\frac{1}{s+e}\right)}{\ln(s+e)\ln(s+e+1)} \\ &\geq \frac{1}{(s+e+1)(\ln(s+e+1))^2}\,. \end{aligned}$$

Thus, $w$ is a heavy-tailed universal distribution over the nonnegative integers in the sense that $-\ln w_s \in O(\ln s)$, similarly to Elias' delta coding (Elias, 1975).

Tangentially, sampling according to $w_s$ is simple since its cumulative distribution telescopes: Sample $u$ uniformly in $[0, 1]$, then select the integer $\min\{s \geq 0 : 1 - 1/\ln(s+e+1) \geq u\}$, that is, select $s = \lceil \exp(1/(1-u)) - e - 1 \rceil$.

## C. Hyperparameters

For the transformer encoder, hyperparameter notations from (Vaswani et al., 2017) are given in parenthesis for reference. The model is trained using stochastic gradient descent with the Adam optimizer (Kingma & Ba, 2015) with $\beta_1$ = 0.9, $\beta_2$ = 0.999 and $\epsilon = 10^{-8}$. A subset of hyperparameters have been selected by an initial investigation for smaller time duration, selected values are underlined: number of layers ($N$) in {<u>3</u>, 4, 5}, embedding size ($d_k$, $d_v$) in {<u>64</u>, 128, 256}, hash vector size in {16, <u>64</u>, 256}, learning rate in {0.003, <u>0.001</u>, 0.0003}, probability of dropout ($P_{drop}$) in {0., <u>0.1</u>, 0.2, 0.3, 0.5, 0.7}. The other hyperparameters were fixed: the batch size is 2560, the number of attention heads ($h$) is 8, which leads to a dimensionality ($d_{model}$) of 512, the inner-layers have a dimensionality ($d_{FF}$) of 1024. To ensure diversity in the training examples, learners wait for the experience replay buffer to contain at least 65536 examples and then sample uniformly from it. The learners used Nvidia V100s GPUs with 16GB of memory.

For the MLP architecture, we use the same architecture as described in (Firoiu et al., 2021) with the same hyperparameters.

The sequential transformer which represents every formula as text we need to use a smaller batch size to fit on GPU memory. We train our method with the maximum sequence length of 400 and batch size of 512. The rest of the parameters like number of layers, embedding size, hash vector size, learning rate and dropout probability are same as spectral transformer described above.

For GNNs, we use a simple architecture with node-update as well as edge-update at each message passing step. The globals are also updated at each step and used for final prediction. After a preliminary hyperparameter search on number of layers, number of message passing steps, embedding size, aggregation function, we observe that number of layers=1, steps=8, embedding size=64, aggregation function=max performs best and is used for all experiments. We ue Adam optimizer with learning rate of 3e-5.

For the actors, the age, weight, and learned-cost queues in our given-clause algorithm are selected on average 1/13th, 3/13th and 9/13th of the steps, respectively. All actors and E are limited at 8GB of memory on modern AMD 64-bit platforms.