Proving theorems using Incremental Learning and Hindsight Experience Replay

1 Introduction

The highest performing ATP systems (e.g., [7, 18]) in first order logic 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 [5, 13, 19] build on top of these provers, using modern machine learning techniques to augment, select or prioritize their already existing heuristics, with some success. Other recent works do not build on top of other provers, but still require existing proof examples as input (e.g., [9, 23]). Such machine-learning-based ATP systems can struggle to solve difficult problems when the training dataset does not provide problems of sufficiently diverse difficulties.

In this paper, we propose an approach which can build a strong theorem prover without relying on existing domain-specific 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. Additionally, all the previous approaches [19, 1, 13] 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. To tackle this challenge, we adapt the idea of hindsight experience replay (HER) [3] 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 in principle 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 [2] and M2k [17] and compare it both with TRAIL [1], a recent machine learning prover as well as with E prover [24, 7], one of the leading heuristic provers. Our proposed approach substantially outperforms TRAIL [1] 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 almost always (99.5% of cases) finds shorter proofs than E.

2 Methodology

We describe the two key components of our approach: how we adapt hindsight experience replay in an incremental learning pipeline, and how clauses are represented for the learner.

Incremental Learning and Hindsight Experience Replay Similar to previous approaches [19, 13, 1], we use the given clause algorithm [21] where the clause scoring heuristics are replaced with a neural network. We start with no proof data to start, and train a simple binary classifier to determine if a particular clause appears in a proof of a conjecture or not. The classifier is

Table 1: Number of conjectures proven on MPTP2078 and M2k.

Domain	Conjectures	Heuristic Approaches				ML Approaches		
		E-basic (100s)	E-auto (100s)	E auto-schedule (100s)	E-best (7 days)	TRAIL	IL w/o HER	IL w/HER
MPTP2078	2078	555	1139	1289	**1369**	1213	1056	**1353**
M2k	2003	1451	1845	**1911**	**1923**	1808	1688	**1861**

trained in an incremental manner where the new proof data obtained by the proof attempts is used to feed the classifier in a continuous manner. The key issue in such an approach arises if the complete set of conjectures are either very difficult or there are big gaps in difficulty of given conjectures such that no training data can be generated by proof attempts to train the classifier. To counter this, we adapt the idea of hindsight experience replay in ATP where any proof attempt whether successful or failure would generate new data for classifier. 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. 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*.[1] Then, the set $C_s \cup \{\neg D\}$ can be refuted. Further, 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.

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 a heterogeneous directed acyclic graphs, called *clause graph* similar to [4]. We use a Transformer encoder architecture [25] 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 graph [8]; this is given by the eigenvectors of the Laplacian matrix of the graph. This replaces the traditional positional encoding in transformers.

3 Experiments and Results

We implement our approach on top of E prover but disable all clause scoring heuristics of E. We use a maximum time of 100s for each proof attempt. We evaluate on two popular benchmarks: MPTP2078[2] and M2k[17] which are widely used in literature. Further, we compare our results with E in different configurations as well as incremental learning without hindsight (IL w/o HER) and TRAIL[1], a recent ML based prover. Table 1 shows the number of proved conjectures by all provers. IL w/HER not only outperforms TRAIL, IL w/o HER and E (100s) but achieves a competing performance when E is run for the *whole* duration of training time. We refer the reader to the appendix for further details of methodology, experimental setup and additional results.

[1] Note that, while the original version of HER [3] only uses the last reached state as a single hindsight goal, we use all intermediate clauses, providing many more data points.

References

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

[2] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *Journal of Automated Reasoning*, 52(2):191–213, 2014.

[3] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[4] Eser Aygün, Zafarali Ahmed, Ankit Anand, Vlad Firoiu, Xavier Glorot, Laurent Orseau, Doina Precup, and Shibl Mourad. Learning to prove from synthetic theorems. *arXiv preprint arXiv:2006.11259*, 2020.

[5] Karel Chvalovskỳ, Jan Jakubův, Martin Suda, and Josef Urban. Enigma-ng: efficient neural and gradient-boosted inference guidance for e. In *International Conference on Automated Deduction*, pages 197–215. Springer, 2019.

[6] Maxwell Crouse, Ibrahim Abdelaziz, Bassem Makni, Spencer Whitehead, Cristina Cornelio, Pavan Kapanipathi, Kavitha Srinivas, Veronika Thost, Michael Witbrock, and Achille Fokoue. A deep reinforcement learning approach to first-order logic theorem proving. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(7):6279–6287, 2021.

[7] Simon Cruanes, Stephan Schulz, and Petar Vukmirović. Faster, Higher, Stronger: E 2.3. In *TACAS 2019*, volume 11716 of *LNAI*, pages 495–507, April 2019.

[8] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *CoRR*, abs/2012.09699, 2020.

[9] Zarathustra Amadeus Goertzel. Make E smart again (short paper). In *Automated Reasoning*, pages 408–415. Springer International Publishing, 2020.

[10] Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Four decades of mizar. *Journal of Automated Reasoning*, 55(3):191–198, 2015.

[11] Malte Helmert, Tor Lattimore, Levi H. S. Lelis, Laurent Orseau, and Nathan R. Sturtevant. Iterative budgeted exponential search. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, page 1249–1257. AAAI Press, 2019.

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

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

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

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

[16] Cezary Kaliszyk and Josef Urban. Learning-assisted theorem proving with millions of lemmas. *Journal of symbolic computation*, 69:109–128, 2015.

[17] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

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

[19] Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. 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*, pages 85–105, 2017.

[20] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180, September 1993.

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

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

[23] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

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

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Appendix

Methodology

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.

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 [18, 7, 21]. 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.;) 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 [6, 1]. 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 [19, 5].

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 [14]). 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.

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 [19] and [15] 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.[2] 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 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) [3] 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*.[3] 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] 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.

[3] Note that, while the original version of HER [3] only uses the last reached state as a single hindsight goal, we use all intermediate clauses, providing many more data points.

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. (*\ref{alg:sample_examples
        }*)
    put_examples(example_buffer, examples)

def actor_manager(conjectures, task_queue):
  schedulers = []
  for conjecture in conjectures:
    schedulers[conjecture] = initialize_UBS() # see Section (*\ref{sec:ubs}*)
  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))
```

Incremental learning algorithm

Typical supervised learning ATP systems require a set of proofs (provided by other provers) to optimize their model (e.g., [19, 13, 4]). Success is assessed by cross-validation. In contrast, we formulate ATP as an incremental learning problem—see in particular [22, 12]. 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

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 $\times$ 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 $\in$ hindsight_goals}
  for size in sizes:
    size_goals = {c $\in$ hindsight_goals :
        tree_size(c) == size}
    w_size = 1 / ln(size + e) - 1 / ln(size + e + 1)
    num_examples = ceil(target_num_examples $\times$ 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
```

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 [11, 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 [20, 11]. 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 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. Note that during the first 1 000 updates, the actors do not use the clause-scoring network as its outputs are mostly random.[4]

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. 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. Finally, all the positive and negative examples thus sampled are added to the training pool for the learners.

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[5]. 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

[4] We picked 1000 as it appeared to be approximately the number of steps required for the learner to reach the base prover performance on a few experiments.

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

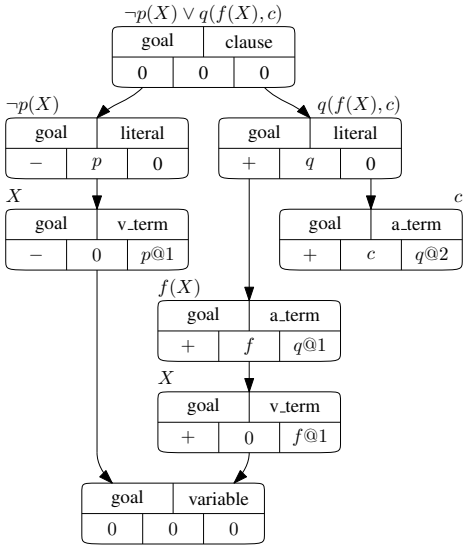

Figure 1: Clause graph of a goal clause. Each node has five features: clause type, node type, literal polarity, symbol hash and argument slot hash. The parts of formula corresponding to each node are shown outside of the nodes.

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 [25] 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 graph [8]; 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.

Additional Results

To evaluate our approach, we use two popular benchmarks created out of the larger Mizar Mathematical Library [10] and used in previous works [6, 17, 16]: MPTP2078 [2] is a sample of larger Mizar datasets which is a good mixture of hard and easy theorems; M2k [17] 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. The relative hardness of these datasets is also illustrated by the fact that the state-of-the-art E prover proves less than 70% theorems in MPTP2078 while it achieves proof rate greater than 95% on M2k theorems. We ignore five problems in MPTP2078 and 13 problems in M2k due to E failing to generate a conjunctive

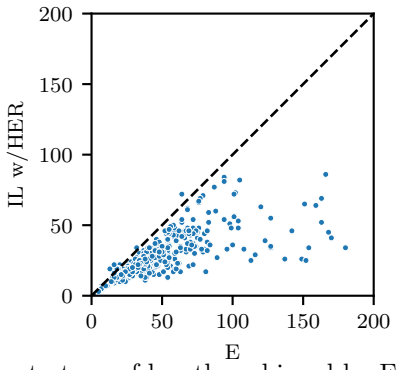

Figure 2: Scatter plot of the shortest proof lengths achieved by E vs. incremental learning with hindsight experience replay on the conjectures that can be proven by both.

normal form (first step in proving) of the problem.

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 [6] and [1] [6]), (iii) E in *auto-schedule* mode for 100s (we observed that the auto-schedule mode significantly outperforms the auto mode), (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). We used E prover version 2.5 [7] 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 100s. 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 [1]. 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. [1] 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 [13] and its variants, and [19]; but these provers rely heavily either on E's heuristics on or input proof data to bootstrap from, and thus fall in a different category 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 significantly outperforms TRAIL on both datasets. Interestingly, since using HER is orthogonal to the methods used by TRAIL, one could hope that combining both

[6] The exact results reported by [1] 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 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	94	78
M2k	79	17

approaches could lead to even better results—but we leave this as future work. IL w/HER proved 2.5 times as many problems as the E-basic on MPTP2078 and 1.28 times as many as E basic on M2k, improving its performance substantially through the use of a learned clause-scoring network. IL w/HER also outperforms E-auto as well as E auto-schedule on the MPTP2078 dataset. We do not see a similar improvement on the M2k dataset. This can be due to the fact that M2k is a subset of theorems already proven by ATPs and hence, by construction, it consists of the sub-sample of theorems on which E already performs well. Lastly, as IL w/HER ran for seven days, attempting each conjecture multiple times (though each attempt was allowed a maximum of 100s) , in order to give E a fair chance, we also ran E for multiple time durations (100s, 1h, 1d, 7d) and we report the maximum number of conjectures proved in of all these runs as E-best. Our approach comes very close (less than 1% difference) to the performance of E-best on the MPTP2078 dataset.

Unique theorems proved by our approach. Additionally, Table 2 shows the number of theorems proven only by our approach and not E-best, and the other way around.. IL w/HER manages to prove 78 theorems on MPTP-2078 and 17 theorems on M2k which are not proven by E-best. This suggests that IL w/HER can find strategies that are absent from E.

Table 3: Comparison of different neural network architectures in IL w/HER on MPTP-2078 and M2k.

Domain	Conjectures	MLP	GNN	Sequential transformer	Spectral transformer
MPTP2078	2078	1049	1221	1076	**1353**
M2k	2003	1772	1756	1704	**1861**

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 297 (14.3%) of the conjectures on MPTP2078 and 173(8.6%) conjectures on M2k 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.

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 2 shows a scatter plot of the lengths of the shortest proofs found by E vs. found by IL w/HER for each theorem. The shortest proofs found by our system were consistently shorter than those found by E. Out of the 3119 conjectures

proven by both systems, our proofs were shorter for 3106 conjectures (99.5%) whereas E's proofs were shorter for only 8 conjectures, with 5 proofs being of the same length.

Speed of search. E was able to search 13.6 times faster than our provers, in terms of clauses generated per second. 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.

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 Sec. 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. For GNNs, we used the same graph structure as the spectral transformer described in Sec. 3. An additional root node is added at the top to connect the target clause with the negated conjecture clauses, in order to allow message passing between different clauses. Table 3 shows the conjectures solved by using different representations trained with IL w/HER using 1000 actors. We observe that GNNs outperform MLPs 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.