Guiding an Automated Theorem Prover with Neural Rewriting

Jelle Piepenbrock[1,2] [0000−0002−8385−9157], Tom Heskes[2] [0000−0002−3398−5235], Mikoláš Janota[1] [0000−0003−3487−784X], and Josef Urban[1] [0000−0002−1384−1613]

[1] Czech Technical University in Prague, Czech Republic
[2] Radboud University, The Netherlands

Abstract. Automated theorem provers (ATPs) are today used to attack open problems in several areas of mathematics. An ongoing project by Kinyon and Veroff uses Prover9 to search for the proof of the Abelian Inner Mapping (AIM) Conjecture, one of the top open conjectures in quasigroup theory. In this work, we improve Prover9 on a benchmark of AIM problems by neural synthesis of useful alternative formulations of the goal. In particular, we design the 3SIL (stratified shortest solution imitation learning) method. 3SIL trains a neural predictor through a reinforcement learning (RL) loop to propose correct rewrites of the conjecture that guide the search.
3SIL is first developed on a simpler, Robinson arithmetic rewriting task for which the reward structure is similar to theorem proving. There we show that 3SIL outperforms other RL methods. Next we train 3SIL on the AIM benchmark and show that the final trained network, deciding what actions to take within the equational rewriting environment, proves 70.2% of problems, outperforming Waldmeister (65.5%). When we combine the rewrites suggested by the network with Prover9, we prove 8.3% more theorems than Prover9 in the same time, bringing the performance of the combined system to 90%.

Keywords: Automated theorem proving · machine learning

1 Introduction

Machine learning (ML) has recently proven its worth in a number of fields, ranging from computer vision [17], to speech recognition [15], to playing games [28,40] with *reinforcement learning* (RL) [45]. It is also increasingly applied in automated and interactive theorem proving. Learned predictors have been used for premise selection [1] in hammers [6], to improve clause selection in saturation-based theorem provers [9], to synthesize functions in higher-order logic [12], and to guide connection-tableau provers [21] and interactive theorem provers [14,2,5].

Future growth of the knowledge base of mathematics and the complexity of mathematical proofs will increase the need for proof checking and its better computer support and automation. Simultaneously, the growing complexity of software will increase the need for formal verification to prevent failure modes [10].

Automated theorem proving and mathematics will benefit from more advanced ML integration. One of the mathematical subfields that makes substantial use of automated theorem provers is the field of quasigroup and loop theory [32].

1.1 Contributions

In this paper, we propose to use a neural network to suggest lemmas to the Prover9 [25] ATP system by rewriting parts of the conjecture (Section 2). We test our method on a dataset of theorems collected in the work on the Abelian Inner Mapping (AIM) Conjecture [24] in loop theory. For this, we use the AIMLEAP proof system [7] as a reinforcement learning environment. This setup is described in Section 3. For development we used a simpler Robinson arithmetic rewriting task (Section 4). With the insights derived from this and a comparison with other methods, we describe our own 3SIL method in Section 5. We use a neural network to process the state of the proving attempt, for which the architecture is described in Section 6. The results on the Robinson arithmetic task are described in Section 7.1. We show our results on the AIMLEAP proving task, both using our predictor as a stand-alone prover and by suggesting lemmas to Prover9 in Section 7.2. Our contributions are:

1. We propose a training method for reinforcement learning in theorem proving settings: *stratified shortest solution imitation learning* (3SIL). This method is suited to the structure of theorem proving tasks. This method and the reasoning behind it is explained in Section 5.
2. We show that 3SIL outperforms other baseline RL methods on a simpler, Robinson arithmetic rewriting task for which the reward structure is similar to theorem proving (Section 7.1).
3. We show that a standalone neurally guided prover trained by the 3SIL method outperforms the hand-engineered Waldmeister prover on the AIMLEAP benchmark (Section 7.2).
4. We show that using a neural rewriting step that suggests rephrased versions of the conjecture to be added as lemmas improves the ATP performance on equational problems (Sections 2 and 7.2).

2 ATP and Suggestion of Lemmas by Neural Rewriting

Saturation-based ATPs make use of the *given clause* [30] algorithm, which we briefly explain as background. A problem is expressed as a conjunction of many initial clauses (i.e., the clausified axioms and the negated goal which is always an equation in the AIM dataset). The algorithm starts with all the initial clauses in the *unprocessed set*. We then pick a clause from this set to be the given clause and move it to the *processed set* and do all inferences with the clauses in the processed set. The newly inferred clauses are added to the unprocessed set. This concludes one iteration of the algorithm, after which we pick a new given

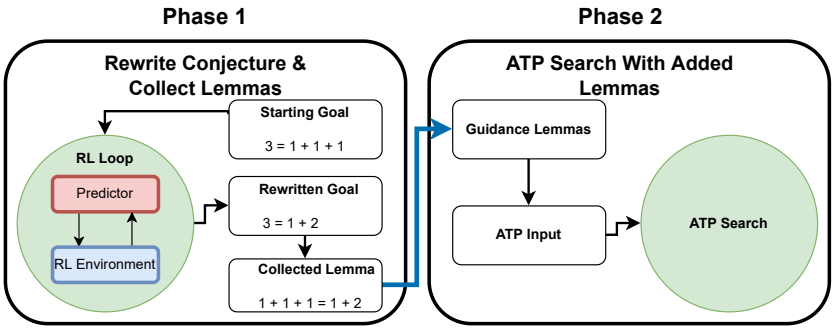

Fig. 1. Schematic representation of the proposed guidance method. In the first phase, we run a reinforcement learning loop to propose actions that rewrite a conjecture. This predictor is trained using the AIMLEAP proof environment. We collect the rewrites of the LHS and RHS of the conjecture. In the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

clause and repeat [23]. Typically, this approach is designed to be *refutationally complete*, i.e., the algorithm is guaranteed to eventually find a contradiction if the original goal follows from the axioms.

This process can produce a lot of new clauses and the search space can become quite large. In this work, we modify the standard loop by adding useful lemmas to the initial clause set. These lemmas are proposed by a neural network that was trained *from zero knowledge* to rewrite the left- and right-hand sides of the initial goal to make them equal by using the axioms as the available rewrite actions. Even though the neural rewriting might not fully succeed, the rewrites produced by this process are likely to be useful as additional lemmas when added to the problem. This idea is schematically represented in Figure 1.

3 AIM Conjecture and the AIMLEAP RL Environment

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

As a testbed for our machine learning and prover guidance methods we use a previously published dataset of problems generated by the AIM conjecture [7]. The dataset comes with a simple prover called AIMLEAP that can take machine learning advice.[3] We use this system as an RL environment. AIMLEAP keeps the state and carries out the cursor movements (the cursor determines the location of the rewrite) and rewrites that a neural predictor chooses.

[3] https://github.com/ai4reason/aimleap

The AIM conjecture concerns specific structures in *loop theory* [24]. A loop is a quasigroup with an identity element. A quasigroup is a generalization of a group that does not preserve associativity. This manifests in the presence of two different 'division' operators, one left-division ($\backslash$) and one right-division ($/$). We briefly explain the conjecture to show the nature of the data.

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

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

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

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

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

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

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

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

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

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

3.1 The AIMLEAP RL Environment

Currently, work in this area is done using automated theorem provers such as *Prover9* [25,24]. This has led to some promising results, but the search space is enormous. The main strategy for proving the AIM conjecture thus far has been to prove weaker versions of the conjecture (using additional assumptions) and then import crucial proof steps into the stronger version of the proof. The *Prover9* theorem prover is especially suited to this approach because of its well-established *hints* mechanism [48]. The AIMLEAP dataset is derived from this

Prover9 approach and contains around 3468 theorems that can be proven with the supplied definitions and lemmas [7].

There are 177 possible actions in the AIMLEAP environment [7]. We handle the proof state as a tree, with the root node being an equality node. Three actions are cursor movements, where the cursor can be moved to an argument of the current position. The other actions all rewrite the current term at the cursor position with various axioms, definitions and lemmas that hold in the AIM context. As an example, this is one of the theorems in the dataset (\ and = are part of the language):

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

The task of the machine learning predictor is to process the proof state and recognize which actions are most likely to lead to a proof, meaning that the two sides of the starting equation are equal according to the AIMLEAP system. The only feedback that the environment gives is whether a proof has been found or not: there is no intermediate reward (i.e. rewards are *sparse*). The ramifications of this are further discussed in Section 5.1.

4 Rewriting in Robinson Arithmetic as an RL Task

To develop a machine learning method that can help solve equational theorem proving problems, we considered a simpler arithmetic task, which also has a tree-structured input and *a sparse reward structure*: the normalization of Robinson arithmetic expressions. The task is to normalize a mathematical expression to one specific form. This task has been implemented as a Python RL environment, which we make available.[4] The learning environment incorporates an existing dataset, constructed by Gauthier for RL experiments in the interactive theorem prover HOL4 [11]. Our RL setup for the task is also modeled after [11].

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

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

[4] https://github.com/learningeqtp/rewriteRL

term which is out of scope for the current work. The place where the rewrite is applied is denoted by the location of the *cursor* in the expression tree.

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

5 Reinforcement Learning Methods

This section describes the reinforcement learning methods, while Section 6 then further explains the particular neural architectures that are trained in the RL loops. We first briefly explain here the approaches that we used as reinforcement learning (RL) baselines, then we go into detail about the proposed 3SIL method.

5.1 Reinforcement Learning Baselines

General RL setup For comparison, we used implementations of four established reinforcement learning baseline methods. In reinforcement learning, we consider an *agent* that is acting within an *environment*. The agent can take actions a from the action-space $\mathcal{A}$ to change the state $s \in \mathcal{S}$ of the environment. The agent can be rewarded for certain actions taken in a certain states, with reward given by the *reward function* $\mathcal{R} : (\mathcal{S} \times \mathcal{A}) \to \mathbb{R}$. The behavior of the environment is given by the *state transition function* $\mathcal{P} : (\mathcal{S} \times \mathcal{A}) \to \mathcal{S}$. The history of the agent's actions and the environments states and rewards at each timestep t are collected in tuples (s_t, a_t, r_t). For a given history of a certain agent within an environment, we call the list of tuples (s_t, a_t, r_t) describing this history an *episode*. The *policy function* $\pi : \mathcal{S} \to \mathcal{A}$ allows the agent to decide which action to take. The agent's goal is to maximize the return R: the sum of discounted rewards $\sum_{t \geq 0} \gamma^t r_t$, where γ is a *discount factor* that allows control over how heavily rewards further in the future should be weighted. We will use R_t when we mean R, but calculated only from rewards from timestep t on. In the end, we are thus looking for a policy function π that maximizes the sum R of (discounted) expected rewards [45].

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

A2C The first method, *Advantage Actor-Critic*, or *A2C* [27] contains ideas on which the other three RL baseline methods build, so we will go into more detail for this method, while keeping the explanation for the other methods brief. For details we refer to the corresponding papers.

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

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

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

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

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

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

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

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

which minimizes the advantage function. The logic of this is that the value estimate at timestep t, $\mathcal{V}_\mu(s_t)$, will learn to incorporate the later rewards R_t, ensuring that when later seeing the same state, the possible future reward will be considered. Note that the sets of parameters θ and μ are not necessarily disjoint (see Section 6).

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

Compared to two-player games, such as chess and go, for which many approaches have been tailored and successfully used [41], theorem-proving has the property that it is hard to collect useful examples to learn from, as only successful proofs are likely to contain useful knowledge. In chess or go, however, one

player almost always wins and the other loses, which means that we can at least learn from the difference between the two strategies used by those players. As an example, we executed 2 million random proof attempts on the AIMLEAP environment, which led to 300 proofs to learn from, whereas in a two-player setting like chess, we would get 2 million games in which one player would likely win.

ACER The second RL baseline method we tested in our experiments is ACER, *Actor-Critic with Experience Replay* [49]. This approach can make use of data from older episodes to train the current predictor. ACER applies corrections to the value estimates so that data from old episodes may be used to train the current policy. It also uses trust region policy optimization [35] to limit the size of the policy updates. This method is included as a baseline to check if using a larger replay buffer to update the parameters would be advantageous.

PPO Our third RL baseline is the widely used *proximal policy optimization* (PPO) algorithm [36]. It restricts the size of the parameter update to avoid causing a large difference between the original predictor's behavior and the updated version's behavior. The method is related to the above trust region policy optimization method. In this way, PPO addresses the training instability of many reinforcement learning approaches. It has been used in various settings, for example complex video games [4]. With its versatility, the PPO algorithm is well-positioned. We use the PPO algorithm with clipped objective, as in [36].

SIL-PAAC Our final RL baseline uses only the transitions with positive advantage to train on for a portion of the training procedure, to learn more from good episodes. This was proposed as *self-imitation learning* (SIL) [29]. To avoid confusion with the method that we are proposing, we extend the acronym to SIL-PAAC, for positive advantage actor-critic. This algorithm outperformed A2C on the sparse-reward task Montezuma's Revenge (a puzzle game). As theorem proving has a sparse reward structure, we included SIL-PAAC as a baseline. More information about the implementations for the baselines can be found in the Implementation Details section at the end of this work.

5.2 Stratified Shortest Solution Imitation Learning

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

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

Algorithm 1 CollectEpisode

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

Algorithm 2 3SIL

Input: set of problems P, randomly initialized policy π_θ, batch size B, number of batches NB, problem history H, number of warmup episodes m, number of episodes f, max epochs ME
Output: trained policy π_θ, problem history H
for $e = 0$ **to** ME $- 1$ **do**
  if $e = 0$ **then** num $= m$ **else** num $= f$
  for $i = 0$ **to** num $- 1$ **do**
    CollectEpisode(sample(P), π_θ, H) (Algorithm 1)
  end for
  for $i = 0$ **to** NB $- 1$ **do**
    Sample B tuples (s, a) with uniform probability for each problem from H
    Update θ to lower $-\sum_{b=0}^{B} \log \pi_\theta(a_b|s_b)$ by gradient descent
  end for
end for

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

The 3SIL algorithm is described in more detail in Algorithm 2. Sampling from a noisy version of policy π_θ means that actions are sampled from the predictor-defined distribution and in 5% of cases a random valid action is selected. This is also known as the ϵ-greedy policy (with ϵ at 0.05).

Related Methods Our approach is similar to the imitation learning algorithm DAGGER (Dataset Aggregation), which was used for several games [34] and modified for branch-and-bound algorithms in [16]. The behavioral cloning (BC) technique used in robotics [47] also shares some elements. 3SIL significantly differs from DAGGER and BC because it does not use an outside expert to obtain useful data, because of the stratified sampling procedure, and because of the selection of the shortest solutions for each problem in the training dataset. We include as an additional baseline an implementation of behavioral cloning (BC), where we regard proofs already encountered as coming from an expert. We minimize cross-entropy between the actions in proofs we have found and

the predictions to train the predictor. For BC, there is no stratified sampling or shortest solution selection, only the minimization of cross-entropy between actions taken from recent successful solutions and the predictor's output.

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

6 Neural Architectures

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

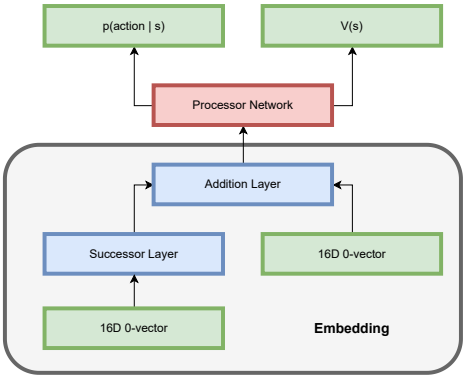

Fig. 2. Schematic representation of the creation of a representation of an expression (*an embedding*) using different neural network layers to represent different operations. The figure depicts the creation of a numerical representation for the Robinson arithmetic expression $(S(0) + 0)$. Note that the successor layer and the addition layer consist of trainable parameters, for which the values are set through gradient descent.

There are two main components to the neural network we use: an *embedding* tree neural network that outputs a numerical vector representing the tree-structured proof state and a second *processor* network that takes this vector representation of the state and outputs a distribution of the actions possible in the environment.[5]

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

Tree neural networks have been used in various settings, such as natural language processing [20] and also in Robinson arithmetic expression embedding [13]. These networks consist of smaller neural networks, each representing one of the possible functions that occur in the expressions. For example, there will be separate networks representing addition and multiplication. The cursor is a special unary operation node with its own network that we insert into the tree at the current location. For each unique constant, such as the constant 0 in RA or the identity element 1 for the AIM task, we generate a random vector (from a standard normal distribution) that will represent this leaf. In the case of the AIM task, these vectors are parameters that can be optimized during training.

At prediction time, the numerical representation of a tree is constructed by starting at the leaves of the tree, for which we can look up the generated vectors. These vectors act as input to the neural networks that represent the parent node's operation, yielding a new vector, which now represents the subtree of the parent node. The process repeats until there is a single vector for the entire tree after the root node is processed (see also Figure 2).

The neural networks representing each operation consist of a linear transformation, a non-linearity in the form of a rectified linear unit (ReLU) and another linear transformation. In the case of binary operations, the first linear transformation will have an input dimension of $2n$ and an output dimension of n, where n is the dimension of the vectors representing leaves of the tree (the *internal representation size*). The weights representing these transformations are randomly initialized at the beginning of training.

When we have obtained a single vector embedding representing the entire tree data structure, this vector serves as the input to the *predictor* neural network, which consists of three linear layers, with non-linearities (Sigmoid/ReLU) in between these layers. The last layer has an output dimension equal to the number of possible actions in the environment. We obtain a probability distribution over the actions, e.g. by applying the softmax function to the output of this last layer. In the cases where we also need a value prediction, there is a parallel last layer that predicts the state's value (usually referred to as a *two-headed* network [41]). The internal representation size n for the Robinson arithmetic experiments is set to 16, for the AIM task this is 32. The number of neurons in each layer (except for the last one) of the predictor networks is 64.

In the AIM dataset task, an arbitrary number of variables can be introduced during the proof. These are represented by untrainable random vectors. We add a special neural network (with the same architecture as the networks representing unary operations, so from size n to n) that processes these vectors before they are processed by the rest of the tree neural network embedding. The idea is that this neural network learns to project these new variable vectors into a subspace and that an arbitrary number of variables can be handled. The vectors are resampled at the start of each episode, so the agent cannot learn to recognize specific variables. This approach was partly inspired by the *prime* mechanism in [13], but we use separate vectors for all variables instead of building vectors sequentially. All our neural networks are implemented using the PyTorch library [31].

7 Experiments

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

7.1 Robinson Arithmetic Dataset

Dataset details The Robinson arithmetic dataset [11] is split into three distinct sets, based on the number of steps that it takes a fixed rewriting strategy to normalize the expression. This fixed strategy, LOPL, which stands for *left outermost proof length*, always rewrites the leftmost possible element. If it takes this strategy less than 90 steps to solve the problem, it is in the *low* difficulty category. Problems with a difficulty between 90 and 130 are in the *medium* category and a greater difficulty than 130 leads to the *high* category. The *high* dataset also contains problems the LOPL strategy could not solve within the time limit. The *low* dataset is split into a training and testing set. We train on the *low* difficulty problems, but after training we also test on problems with a higher difficulty. Because we have a difficulty measure for this dataset, we use a curriculum setup. We start by learning to normalize the expressions that a fixed strategy can normalize in a small amount of steps. This setup is similar to [11].

Training setup The 400 problems with the lowest difficulty are the starting point. Every time an agent reaches 95 percent success rate when evaluated on a sample of size 400 from these problems, we add 400 more difficult problems to set of training problems P. One iteration of the *collection* and *training* phase is called an *epoch*. Agents are evaluated after every epoch. The blocks of size 400 are called *levels*. The number of episodes m and f are set to 1000. For 3SIL and BC, the batch size BS is 32 and the number of batches NB is 250. The baselines are configured so that the number of episodes and training transitions is at least as many as the 3SIL/BC approaches. Episodes that take over 100 steps are stopped. ADAM [22] is used as an optimizer.

Results on RA curriculum In Figure 3, we show the progression through the training curriculum for behavioral cloning (BC), the RL methods (PPO, ACER) and two configurations of 3SIL. Behavioral cloning simply imitates actions from successful episodes. Of the RL baselines, PPO reaches the second level in one run, while ACER steadily solves the first level and in the best run solves around 80% of the second level. Both methods do not learn enough solutions for the second level to advance to the third. A2C and SIL-PAAC do not reach the second level, so these are left out of the plot. However, they do learn to solve about 70-80% of the first 400 problems. From these results we can conclude that the RL baselines do not perform well on this task in our experiment. We attribute this to the difficulty of learning a good value function due to the sparse rewards (Section 5.1). Our hypothesis is that because this value estimate influences the policy updates, the RL methods do not learn well on this task. Note

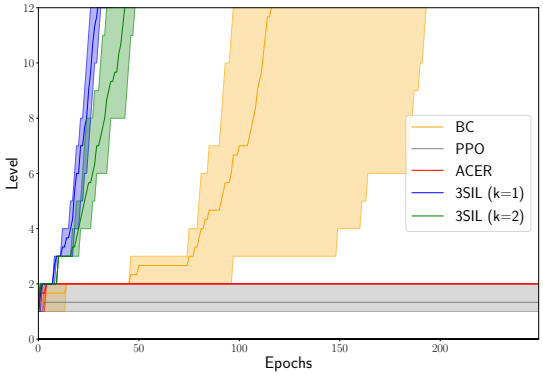

Fig. 3. The level in the curriculum reached by each method. Each method was run three times. The bold line shows the mean performance and the shaded region shows the minimum and maximum performance. K is the number of proofs stored per problem.

that the two methods with a trust region update mechanism, ACER and PPO, perform better than the methods without this mechanism. From these results, it is clear that 3SIL with 1 shortest proof stored, $k = 1$, is the best-performing configuration. It reaches the end of the training curriculum of about 5000 problems in 40 epochs. We experimented with $k = 3$ and $k = 4$, but these were both worse than $k = 2$.

Generalization While our approach works well on the training set, we must check if the predictors generalize to unseen examples. Only the methods that reached the end of the curriculum are tested. In Table 1, we show the results of evaluating the performance of our predictors on the three different test sets: the unseen examples from the *low* dataset and the unseen examples from the *medium* and *high* datasets. Because we expect longer solutions, the episode limits are expanded from 100 steps to 200 and 250 for the *medium* and *high* datasets respectively. For the *low* and *medium* datasets, the second of which contains problems with more difficult solutions than the training data, the predictors solve almost all test problems. For the *high* difficulty dataset, the performance drops by at least 20 percentage points. Our method outperforms the Monte Carlo Tree Search approach used in [11] on the same datasets, which got to 0.954 on the *low* dataset with 1600 iterations and 0.786 on the *medium* dataset (no results on the *high* dataset were reported). These results indicate that this training method might be strong enough to perform well on the AIM rewriting RL task.

Table 1. Generalization with greedy evaluation on the test set for the Robinson arithmetic normalization tasks, shown as average success rate and standard deviation from 3 training runs. Generalization is high on the low and medium difficulty (training data is similar to the low difficulty dataset). With high difficulty data, performance drops.

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

7.2 AIM Conjecture Dataset

Training setup Finally, we train and evaluate 3SIL on the AIM Conjecture dataset. We apply 3SIL ($k = 1$) to train predictors in the AIMLEAP environment. Ten percent of the AIM dataset is used as a hold-out test set, not seen during training. As there is no estimate for the difficulty of the problems in terms of the actions available to the predictor, we do not use a curriculum ordering for these experiments. The number m of episodes collected before training is set to 2,000,000. These random proof attempts result in about 300 proofs. The predictor learns from these proofs and afterwards the search for new proofs is also guided by its predictions. For the AIM experiments, episodes are stopped after 30 steps in the AIMLEAP environment. The predictors are trained for 100 epochs. The number of collected episodes per epoch f is 10,000. The successful proofs are stored, and the shortest proof for each theorem is kept. NB is 500 and BS is set to 32. The number of problems with a solution in the history after each epoch of the training run is shown in Figure 4.

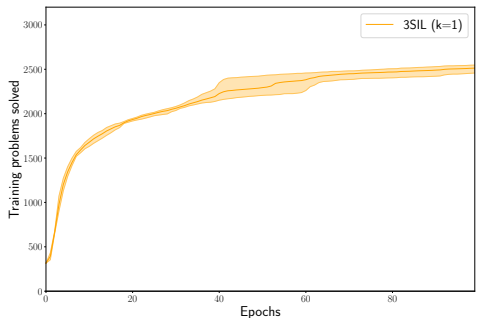

Fig. 4. The number of training problems for which a solution was encountered and stored (cumulative). At the start of the training, the models rapidly collect more solutions, but after 100 epochs, the process slows down and settles at about 2500 problems with known solutions. The minimum, maximum and mean of three runs are shown.

Results as a standalone prover After 100 epochs, about 2500 of 3114 problems in the training dataset have a solution in their history. To test the generalization capability of the predictors, we inspect their performance on the holdout test set problems. In Table 2 we compare the success rate of the trained predictors on the holdout test set with three different automated theorem provers: E [37,38], Waldmeister [19] and Prover9. E is currently one of the best overall automated theorem provers [44], Waldmeister is a prover specialized in memory-efficient equational theorem proving [18] and Prover9 is the theorem prover that is used for AIM conjecture research and the prover that the dataset was generated by. Waldmeister and E are the best performing solvers in competitions for the relevant unit equality (UEQ) category [44].

The results show that a single greedy evaluation of the predictor trying to solve the problem in the AIMLEAP environment is not as strong as the theorem proving software. However, the theorem provers got 60 seconds of execution

Table 2. Theorem proving performance on the hold-out test set in fraction of problems solved. Means and standard deviations are the results of evaluations of 3 different predictors from 3 different training runs on the 354 unseen test set problems.

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

time, and the execution of the predictor, including interaction with AIMLEAP, takes on average less than 1 second. We allowed the predictor setup to use 60 seconds, by running attempts in AIMLEAP until the time was up, sampling actions from the predictor's distribution with 5% noise, instead of using greedy execution. With this approach, the predictor setup outperforms Waldmeister.[6] Figure 5 shows the overlap between the problems solved by each prover. The

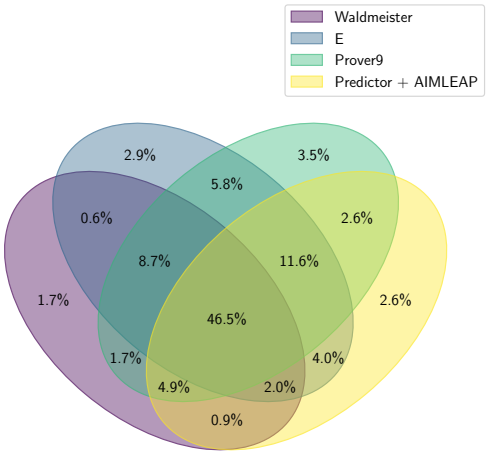

Fig. 5. Venn diagram of the test set problems solved by each solver with 60s time limit.

diagram shows that each theorem prover found a few solutions that no other prover could find within the time limit. Almost half of all problems from the test set that are solved are solved by all four systems.

Results of neural rewriting combined with Prover9 We also combine the predictor with *Prover9*. In this setup, the predictor modifies the starting form of the goal, for a maximum of 1 second in the AIMLEAP environment. This produces new expressions on one or both sides of the equality. We then add, as lemmas, equalities between the left-hand side of the goal before the predictor's

[6] After the initial experiments, we also evaluated Twee [42], which won the most recent UEQ track: it can prove most of the test problems in 60s, only failing for 1 problem.

Table 3. Prover9 theorem proving performance on the hold-out test set when injecting lemmas suggested by the learned predictor. *Prover9*'s performance increases when using the suggested lemmas.

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

rewriting and after each rewriting (see Figure 1). The same is done for the right-hand side. For each problem, this procedure yields new lemmas that are added to the problem specification file that is given to *Prover9*.

In Table 3, it is shown that adding lemmas suggested by the rewriting actions of the trained predictor improves the performance of *Prover9*. Running *Prover9* for 2 seconds results in better performance than running it for 1 second, as expected. The combined (1s + 1s) system improved on *Prover9's* 2-second performance by 12.7% (= 0.841/0.746), indicating that the predictor suggests useful lemmas. Additionally, 1 second of neural rewriting combined with 59 seconds of Prover9 search proves almost 8.3% (= 0.902/0.833) more theorems than Prover9 with a 60 second time limit (Table 2).

7.3 Implementation Details

All experiments for the Robinson task were run on a 16 core Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz. The AIM experiments were run on a 72 core Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz. All calculations were done on CPU. The PPO implementation was adapted from an existing implementation [3]. The model was updated every 2000 timesteps, the PPO clip coefficient was set to 0.2. The learning rate was 0.002 and the discount factor γ was set to 0.99. The ACER implementation was adapted from an available implementation [8]. The replay buffer size was 20,000. The truncation parameter was 10 and the model was updated every 100 steps. The replay ratio was set to 4. Trust region decay was set to 0.99 and the constraint was set to 1. The discount factor was set to 0.99 and the learning rate to 0.001. Off-policy minibatch size was set to 1.The A2C and SIL implementations were based on Pytorch actor-critic example code available at the PyTorch repository [33]. For the A2C algorithm, we experimented with two formulations of the advantage function: the 1-step lookahead estimate $(r_t + \gamma\mathcal{V}_\mu(s_{t+1})) - \mathcal{V}_\mu(s_t)$ and the $R_t - \mathcal{V}_\mu(s_t)$ formulation. However, we did not observe different performance, so we opted in the end for the 1-step estimate favored in the original A2C publication. For SIL-PAAC, we implemented the SIL loss on top of the A2C implementation. There is also a prioritized replay buffer with an exponent of 0.6, as in the original paper. Each epoch, 8000 (250 batches of size 32) transitions were taken from the prioritized replay buffer in the SIL step of the algorithm. The size of the prioritized replay buffer was 40,000. The critic loss weight was set to 0.01 as in the original paper. For the 3SIL and behavioral cloning implementations, we sample 8000 transitions (250 batches of

size 32) from the replay buffer or history. For the behavioral cloning, we used a buffer of size 40,000. An example implementation of 3SIL can be found in the RewriteRL repository. On the Robinson arithmetic task, for 3SIL and BC, the evaluation is done greedily (always take the highest probability actions). For the other methods, we performed experiments with both greedy and non-greedy (sample from the predictor distribution and add 5% noise) evaluation and show the results the best-performing setting (which in most cases was the non-greedy evaluation, except for PPO). On the AIM task, we evaluate greedily with 3SIL.

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

Versions of the automated theorem provers used: Version 2.5 of E [39], the Nov 2017 version of Prover9 [26] and the Feb 2018 version of Waldmeister [46] and version 2.4.1 of Twee [43].

8 Conclusion and Future Work

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

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

Acknowledgements

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

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

References

1. Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., Urban, J.: Premise selection for mathematics by corpus analysis and kernel methods. J. Autom. Reasoning **52**(2), 191–213 (2014). https://doi.org/10.1007/s10817-013-9286-5
2. Bansal, K., Loos, S., Rabe, M., Szegedy, C., Wilcox, S.: HOList: An environment for machine learning of higher order logic theorem proving. In: International Conference on Machine Learning. pp. 454–463 (2019)
3. Barhate, N: Implementation of PPO algorithm, https://github.com/nikhilbarhate99
4. Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al.: DOTA 2 with large scale deep reinforcement learning. arXiv preprint arXiv:1912.06680 (2019)
5. Blaauwbroek, L., Urban, J., Geuvers, H.: The Tactician - A seamless, interactive tactic learner and prover for Coq. In: CICM. Lecture Notes in Computer Science, vol. 12236, pp. 271–277. Springer (2020)
6. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Formalized Reasoning **9**(1), 101–148 (2016). https://doi.org/10.6092/issn.1972-5787/4593, http://dx.doi.org/10.6092/issn.1972-5787/4593
7. Brown, C.E., Piotrowski, B., Urban, J.: Learning to advise an equational prover. Artificial Intelligence and Theorem Proving (2020)
8. Chételat, D: Implementation of ACER algorithm, https://github.com/dchetelat/acer
9. Chvalovský, K., Jakubuv, J., Suda, M., Urban, J.: ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In: International Conference on Automated Deduction. pp. 197–215. Springer (2019)
10. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
11. Gauthier, T.: Deep reinforcement learning in HOL4. arXiv preprint arXiv:1910.11797v1 (2019), https://arxiv.org/abs/1910.11797v1
12. Gauthier, T.: Deep reinforcement learning for synthesizing functions in higher-order logic. In: International Conference on Logic for Programming, Artificial Intelligence and Reasoning (2020)
13. Gauthier, T.: Tree neural networks in HOL4. In: International Conference on Intelligent Computer Mathematics. pp. 278–283. Springer (2020)
14. Gauthier, T., Kaliszyk, C., Urban, J., Kumar, R., Norrish, M.: TacticToe: Learning to prove with tactics. Journal of Automated Reasoning pp. 1–30 (2020)
15. Graves, A., Fernández, S., Gomez, F., Schmidhuber, J.: Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In: Proceedings of the 23rd International Conference on Machine Learning. pp. 369–376 (2006)
16. He, H., Daume III, H., Eisner, J.M.: Learning to search in branch and bound algorithms. Advances in Neural Information Processing Systems **27**, 3293–3301 (2014)
17. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 770–778 (2016)
18. Hillenbrand, T., Buch, A., Vogt, R., Löchner, B.: WALDMEISTER - High-Performance Equational Deduction. Journal of Automated Reasoning **18**, 265–270 (2004)

19. Hillenbrand, T.: Citius altius fortius: Lessons learned from the theorem prover WALDMEISTER. ENTCS **86**(1), 9–21 (2003)
20. Irsoy, O., Cardie, C.: Deep recursive neural networks for compositionality in language. Advances in Neural Information Processing Systems **27**, 2096–2104 (2014)
21. Kaliszyk, C., Urban, J., Michalewski, H., Olšák, M.: Reinforcement learning of theorem proving. Advances in Neural Information Processing Systems **31**, 8822–8833 (2018)
22. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
23. Kinyon, M.: Proof simplification and automated theorem proving. CoRR **abs/1808.04251** (2018), http://arxiv.org/abs/1808.04251
24. Kinyon, M., Veroff, R., Vojtěchovský, P.: Loops with abelian inner mapping groups: An application of automated deduction. In: Automated Reasoning and Mathematics, pp. 151–164. Springer (2013)
25. McCune, W.: Prover9 and Mace (2010), http://www.cs.unm.edu/~mccune/prover9/
26. McCune, W: Prover9, https://github.com/ai4reason/Prover9
27. Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: International conference on machine learning. pp. 1928–1937 (2016)
28. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
29. Oh, J., Guo, Y., Singh, S., Lee, H.: Self-imitation learning. In: International Conference on Machine Learning. pp. 3878–3887 (2018)
30. Overbeek, R.A.: A new class of automated theorem-proving algorithms. J. ACM **21**(2), 191–200 (Apr 1974). https://doi.org/10.1145/321812.321814, https://doi.org/10.1145/321812.321814
31. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019), http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
32. Phillips, J., Stanovský, D.: Automated theorem proving in quasigroup and loop theory. AI Communications **23**(2-3), 267–283 (2010)
33. PyTorch: RL Examples, https://github.com/pytorch/examples/tree/main/reinforcement_learning
34. Ross, S., Gordon, G., Bagnell, D.: A reduction of imitation learning and structured prediction to no-regret online learning. In: Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. pp. 627–635 (2011)
35. Schulman, J., Levine, S., Abbeel, P., Jordan, M., Moritz, P.: Trust region policy optimization. In: Bach, F., Blei, D. (eds.) Proceedings of the 32nd International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 37, pp. 1889–1897. PMLR, Lille, France (07–09 Jul 2015), https://proceedings.mlr.press/v37/schulman15.html
36. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)

37. Schulz, S.: E - A Brainiac Theorem Prover. AI Commun. **15**(2-3), 111–126 (2002)
38. Schulz, S., Cruanes, S., Vukmirovic, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11716, pp. 495–507. Springer (2019). https://doi.org/10.1007/978-3-030-29436-6_29, https://doi.org/10.1007/978-3-030-29436-6_29
39. Schulz, S: Eprover, https://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html
40. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
41. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of go without human knowledge. Nature **550**(7676), 354–359 (2017)
42. Smallbone, N.: Twee: An equational theorem prover. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12699, pp. 602–613. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_35, https://doi.org/10.1007/978-3-030-79876-5_35
43. Smallbone, N: Twee 2.4.1, https://github.com/nick8325/twee/releases/download/2.4.1/twee-2.4.1-linux-amd64
44. Sutcliffe, G.: The CADE-27 automated theorem proving system competition - CASC-27. AI Communications **32**(5-6), 373–389 (2020)
45. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction (2018)
46. T. Hillenbrand and A. Buch and R. Vogt and Bernd Löchner: Waldmeister, https://www.mpi-inf.mpg.de/departments/automation-of-logic/software/waldmeister/download
47. Torabi, F., Warnell, G., Stone, P.: Behavioral cloning from observation. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence. p. 4950–4957. IJCAI'18, AAAI Press (2018)
48. Veroff, R.: Using hints to increase the effectiveness of an automated reasoning program: Case studies. J. Autom. Reason. **16**(3), 223–239 (1996). https://doi.org/10.1007/BF00252178, https://doi.org/10.1007/BF00252178
49. Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., de Freitas, N.: Sample efficient actor-critic with experience replay. International Conference on Learning Representations (2016)