

Deep Reinforcement Learning in HOL4^{*}

Thibault Gauthier

Czech Technical University in Prague, Prague, Czech Republic
email@thibaultgauthier.fr

Abstract. The paper describes an implementation of deep reinforcement learning through self-supervised learning within the proof assistant HOL4. A close interaction between the machine learning modules and the HOL4 library is achieved by the choice of tree neural networks (TNNs) as machine learning models and the internal use of HOL4 terms to represent tree structures of TNNs. Recursive improvement is possible when a given task is expressed as a search problem. In this case, a Monte Carlo Tree Search (MCTS) algorithm guided by a TNN can be used to explore the search space and produce better examples for training the next TNN. As an illustration, tasks over propositional and arithmetical terms, representative of fundamental theorem proving techniques, are specified and learned: truth estimation, end-to-end computation, term rewriting and term synthesis.

1 Introduction

Improvements in automated theorem provers (ATPs) have been so far predominantly done by inventing new search paradigms such as superposition [14] and SMT [2]. Over the years, developers of these provers have optimized their modules and fine-tuned their parameters. As time progresses, it is becoming evident that more intricate collaboration between search algorithms and intuitive guidance is necessary. ATP developers have frequently manually translated their intuition into guiding heuristics and tested many different parameter combinations. The first success demonstrating the possibility of replacing these heuristics by machine learning guidance has been demonstrated in ITP Hammers [3]. There, feature-based predictors on large interactive theorem prover (ITP) libraries learn to select relevant theorems for a conjecture. This step drastically reduces the search space. As a result, ATPs can prove many conjectures proposed by ITP users. The last landmark that is a major source of inspiration for this paper is the development and success of self-improving neurally guided algorithms in perfect information games [29]. In this work, we adapt such algorithms to theorem proving tasks.

The hope is that systems using the learning paradigm will eventually improve on and outperform the best heuristically-guided ATPs. We believe that the best

^{*} This work has been supported by the European Research Council (ERC) grant AI4REASON no. 649043 under the EU-H2020 programme. We would like thank Josef Urban for his contributions to the final version of this paper.

design for a solver is one that generalizes across many tasks. One way to achieve this is to minimize the amount of algorithmic bias based on human knowledge of the specific domain. Eventually, given enough examples, the neural architecture might be able to recognize and exploit patterns by itself. For large domains that require vast amount of knowledge and understanding, the number of required examples to capture all the patterns is too large. That is why our experiments are performed on domains with a small number of basic concepts.

To test the generality of our approach, we choose four different tasks related to theorem proving. The first task is to guess the truth of a formula. Acquiring this ability is important for discarding false conjectures and flawed derivations. The second task is to estimate the value of an expression. It is an example of model evaluation which can be in general useful for conjecturing and approximate reasoning. The aim of the third task is to rewrite a term to a desired form by applying rewrite steps. If successful, the search trace produced can be used as a justification for the computation. Term rewriting is a powerful tool for both ITPs and ATPs. In ITPs, rewrite systems are used extensively to simplify variety of expressions. In ATPs a generalization of term rewriting called paramodulation forms the basis for equality reasoning. The objective of the fourth task is to synthesize terms. In itself, term synthesis is probably the less explored technique as it is often not so efficient way of exploring a search space as deduction-based methods. This task is however crucial in inductive theorem proving [10] and in counterexample generators [6,4] as it can be used to provide an induction predicate or a witness. The current approaches for such tasks often rely on brute force enumeration, heuristics and manually deduced constraints.

Contributions This paper presents a general framework that lays out the foundations for neurally guided solving of multiple tasks related to theorem proving. We evaluate the suitability of tree neural networks to four different theorem proving tasks. On two, we focus on showing how deep reinforcement learning [31] algorithms can acquire the knowledge necessary to solve such problems through exploratory searches. The framework is integrated in the HOL4 [30] system (see Section 6). Hence, HOL4 terms and procedures are used to specify our tasks. The contributions of this paper are: (i) the implementation of TNNs and double-headed TNNs with the associated backpropagation algorithm, (ii) a comparison of their learning abilities with state-of-the-art predictors, (iii) the implementation of a guided MCTS algorithm [5] for arbitrary specified search problems, (iv) the creation of levels for curriculum learning, and (v) the demonstration of continuous self-improvement for a large number of generations.

2 Tree Neural Networks

In the machine learning field, various kinds of predictors are more suitable for learning various tasks. That is why with new problems come new kinds of predictors. It is particularly true for predictors such as neural networks. For maximum learning efficiency, the structure of the problem should be reflected in the structure of the neural network. For example, convolutional neural networks are best

for handling pictures as their structure have space invariant properties whereas recurrent networks can handle text better. For our purpose, we have chosen neural network that are in particular designed to take into account the tree structure of terms and formulas as in [22].

2.1 Architecture

A tree neural network (TNN) is a machine learning model designed to approximate functions from $\mathbb{T}_{\mathbb{O}} \mapsto \mathbb{R}^n$. We define first the structure of the tree neural network and then show how to compute with it.

Definition (tree neural network) Let \mathbb{O} be a set of operators and $\mathbb{T}_{\mathbb{O}}$ be the set of all terms than can be constructed from \mathbb{O} . We define a tree neural network to be a set of feed-forward neural networks with n *layers* and a *tanh* activation function for each layer. There is one network for each operator f noted $NN(f)$ and one for the head. The NN operator of a function with arity a is to learn a function from \mathbb{R}^{a*d} to \mathbb{R}^d . And the head network is to approximate a function from \mathbb{R} to \mathbb{R}^n . As an optimization for a operator f with arity 0, $NN(f)$ is defined to be a vector of weights in \mathbb{R} since multiple layers are not needed for learning a constant function.

Embedding Given a TNN, we can now define recursively an embedding function $E : \mathbb{T}_{\mathbb{O}} \mapsto \mathbb{R}^d$ by $E(f(t_1, \dots, t_a)) =_{def} NN(f)(E(t_1), \dots, E(t_a))$. This function produces an internal representation of the terms to be later processed by the head network. This internal representation is often called a *thought vector*.

Output The head network interprets the internal representation and makes the last computations towards the expected result. In particular, it reduces the embedding dimension d to the dimension of the output n . The application of a TNN on a term t gives the result $H(E(t))$. It is possible to learn a different objective by replacing only the head of the network. We use this to our advantage in the reinforcement learning experiments where we have the double objective of predicting a policy and a value (see Section3). For efficiency reasons, we have dedicated code for double-headed TNN in our implementation.

Higher-Order Terms A HOL4 term can be encoded into a first-order term (i.e. a labeled tree) in two steps. First, the function applications $f x$ are rewritten to $apply(f, x)$ introducing an explicit *apply* operator. Second, the lambda terms $\lambda x.t$ are substituted with $lam(x, t)$ using the additional operator *lam*. All our tasks 5 are however performed on HOL4 terms that are essentially first-order and thus do not require these encodings.

Training To train a TNN over a set of examples, we follow the batch gradient descent algorithm [23] and update the weights using backpropagation.

3 Deep Reinforcement Learning

When possible, the deep reinforcement learning approach [31] is preferable to a supervised learning approach for two main reasons. First, an oracle is not required. This means that the algorithm is more general as it does not require a specific oracle for each task and can even learn task for which nobody knows a good solution. Secondly by decomposing the problem in many steps, the trace of the computation becomes visible which is particularly important if one wants a justification for the final result.

We present here our methodology to achieve deep reinforcement learning through self-supervised learning. It consists of three phases. During the exploration phase, a search tree is build by the MCTS algorithm from a prior value and prior policy. These priors are updated by looking at the consequence of each action. During the training phase, a new TNN learns to replicate this improved policy and value. The competition phase verifies that the new TNN is indeed better than the prior TNN by comparing how efficient they are at guiding the MCTS algorithm on some problems. One iteration of the reinforcement learning loop is called a *generation*.

3.1 Specification of a Search Problem

Any task that can be solved in a series of steps with decision points at each step can be used to construct search problems. For example, theorem proving is considered a search task by construction. Other para-proving tasks are harder to view in such a light, such as: programming, conjecturing, making definitions, refactoring. Here, a search task is described as a single-player perfect information game.

Definition 1. (*Search problem*)

A search problem is an oriented graph.

The nodes are a set of states (\mathbb{S}) with particular labels for: a starting state $s_0 \in \mathbb{S}$, a subset of winning states $\mathbb{W} \subset \mathbb{S}$, and a subset of losing states $\mathbb{L} \subset \mathbb{S}$.

The oriented edges are labeled by a finite set of moves \mathbb{M} . The transition function $T : \mathbb{S} \times \mathbb{M} \mapsto \mathbb{S}$ returns the state reached by making a move from the input state. To solve the problem, an algorithm needs to find a path p from the starting state to a winning state avoiding the losing states.

An end state is a state that is either winning or losing. A policy P is a function from \mathbb{S} to $[0, 1]^{\text{cardinal}(\mathbb{M})}$ that assigns to each state s a real number for each move. It is intended to be a probability which indicates the percentage of times each move should be explored at each state. Policy score for impossible moves are set to 0 thus never selected during MCTS.

A value V is a function from \mathbb{S} to the interval $[0, 1]$. $V(s)$ is used as an estimate of how likely the search algorithm is to complete the task. Therefore, the function needs to respect this additional constraints: a value of 1 for winning states, and a value of 0 for losing states.

In our implementation, P and V are approximated simultaneously by a double-headed TNN.

3.2 Monte Carlo Tree Search

An in-depth explanation of the Monte Carlo Tree Search algorithm is given in [5]. This search algorithm strikes a good balance between exploration of uncertain paths and exploitation of path leading to states with good value V . The algorithm was recently improved in [29]. The estimation of the value V , which used to be approximated by the proportion of random walks to a winning state, is now returned by a deep neural network.

A search problem is explored by the MCTS algorithm with the help of a prior policy P and a prior value V . The algorithm starts from an initial tree (that we call a *root tree*) containing a initial state and proceeds to gradually build a search tree. Each iteration of the MCTS loop can be decomposed into three main components: node selection, node extension and backup. One step of this loop usually creates a new node in the search tree unless the node selection reached an end state (winning or losing). The search is stopped after a fixed number of iterations of the loop (typically 1600 in our experiments), also called the *number of simulations*. The node selection process is guided by the PUCT formula [1]. We set its *exploration coefficient* to 2.0 in our experiments. The backup step is slightly modified. The rewards are calculated as usual with winning state given a reward of 1 and losing states a reward of 0 and the rewards for other states is given by the value V . The difference to the standard algorithm is that rewards are multiplied by a *discount factor* when propagating a reward of a node to its parent. The maximum search depth in our experiments is around 200, thus we use a discount factor of 0.99 as $0.99^{200} \simeq 0.13$.

This factor was mainly introduced as an attempt to limit looping behavior in our search without explicitly forbidding it when defining the search problem. However, the introduction of a discount factor has the side effects that now the search has two potentially conflicting objectives. The first one is to find a winning state and the second one is to find the shortest one. For example, a value of 0.13 might either indicate that there is 100% chance that a solution can be found in 200 steps or a 13% chance of being found in one step or anything in between. In the future, it might be useful to train for these two objectives separately.

3.3 Big Steps

A full attempt at a solution will rely on multiple calls to the MCTS algorithm. The first call is performed starting from a root tree containing the starting state. After an application of the MCTS algorithm, the constructed tree is used to decide which move to choose. The application of this move to the starting state is a *big step*. The MCTS algorithm is then restarted on a root tree containing the resulting state. This procedure is repeated until a big step results in an end state or after the number of big steps exceeds a fixed bound. This bound is fixed to be twice the difficulty of the starting state. The difficulty measure used varies depending on the task (see Section 5). An attempt is successful if it ends in a winning state.

The decision which big steps to make is taken from the number of visits for each child of the root. During exploration, a big step is chosen randomly with probability proportional to its number of visits. During competition, the move with the highest number of visits is chosen. To encourage exploration even more during the exploration phase, a noise [29] given by the Dirichlet distribution with parameter $\alpha = 0.2$ is added to the prior policy of the root node.

3.4 Collecting Examples

During the exploration phase, we attempt to solve problems by using multiple big steps as described just above. After each big step, we collect an example for an improved policy and an improved value. The input of the example is the state of the root. The improved policy for this example is computed by dividing the number of visits for each child by the total number of visits for all children. The improved value is the average of the values of all the nodes in the tree counted with multiplicity. The new example is added to a dataset of training examples. This dataset is then used to train the TNNs in future generations. When the number of examples in the dataset reaches x , older examples are discarded whenever newer examples are added so that the number of examples never exceeds x . This number is called the *size of the window* and is set to 40000 in our experiments.

3.5 Levels for Curriculum Learning

During the search phases (competition or exploration), it is computationally expensive to make attempts on all 11700 problems constructed from the training set (see Section 4.2). A single generation would take multiple days. Moreover, the first generation starts with a random double-headed TNN and therefore would only find a small percentage of solutions resulting in few positive examples.

That is why we select a restricted number of problems for each search phase. To guarantee that these problems are adapted to the current understanding of the TNN, we construct levels of increasing difficulty by relying on a difficulty measure for the task (see Section 5). The level k consists of the $k * 400$ easiest problems according to the difficulty measure. A search phase on level k is performed on 400 of these problems chosen at random. The first generation of a reinforcement learning run starts at level 1. Leveling up occurs when more than 95 percent of a batch of 400 problems of a certain level is solved. In order to go faster on easy levels, if leveling up occurs during exploration, the phase is restarted one level higher essentially skipping the competition and training phase. We do not count this short loop as one generation. The idea for this levels is to follow a curriculum learning approach and start by training our networks on easy problems and then gradually move to harder problems. Yet, the harder levels still includes easy problems as we do not want the TNN forgetting how to solve them.

4 Datasets

In all our tasks, our algorithms require a training set in order to learn the task at hand, a validation test for tuning the hyper-parameters and one or more testing sets for a final evaluation of the generalization abilities of our algorithm to new examples. The ability of TNNs to learn a task is heavily influenced by the quality of the training examples. Therefore, we think that the following objectives should guide the algorithm generating the training set: a large and diverse enough set of input terms, a uniform distribution of output classes and a gradual increase in difficulty. The number of examples generated should be proportional to the set of operators in our training examples as they influence the number of parameters of the TNN. That is why we try use a minimal number of operators to represent the terms.

4.1 Propositional Formulas

For the propositional task, we re-use the benchmark created by the authors of [12]. Their task was trying to learn if a propositional formula A entails a formula B ($A \models B$). In our experiment, we decide instead to solve the equivalent task of determining if the formula $A \Rightarrow B$ is a tautology or not. From their datasets using this simple transformation, we get 100000 formulas for our training set, 100 formulas in the exam set which consists of examples taken from textbooks, and 5000 formulas for each test set. The "a priori" difficulty of a propositional formula can be estimated as a function of the size of the formula and the number of variables. Using these measures, the five testing sets can be ordered from easiest to hardest: exam, easy, hard, big, massive. More detailed statistics about the benchmark can be found in [12].

4.2 Arithmetical Expressions

The arithmetical experiments relies on three sets of expressions: a training test (*train*), a validation set (*valid*) and a test set (*test*). These datasets with some associated statistics can be downloaded from our github repository¹.

Generation algorithm We describe first the generation of the training set. We call $\mathbb{A} = \{0, s, +, \times\}$ the set of arithmetical operators where s stands for the successor function. $\mathbb{T}_{\mathbb{A}}$ will denote the set of all terms built from those operators. The training set is a subset of $\mathbb{T}_{\mathbb{A}}$. To make the generation of such a subset feasible, we will first try to generate terms for a fixed size. Because the number of such terms grows exponentially as the size increases, it is not feasible to enumerate exhaustively all terms for each size (even for relatively small size value) and take a random subset afterwards. Instead, we use top-down generation by randomly selecting from the root each operator proportionally to the number

¹ https://github.com/barakeel/arithmetic_datasets

of subterms starting with this operator that respects the total size constraint. This algorithm gives a distribution of random terms equivalent to uniformly selecting in $\mathbb{T}_{\mathbb{A}}^k$ avoiding the cost of exhaustive generation.

Using this algorithm to generate a subset of $\mathbb{T}_{\mathbb{A}}^{12}$, we noticed that 39.0% of the expression created evaluate to zero. Moreover, the probability of generating the number 10 would be very low. This bias (or lack of bias) in our dataset is detrimental for learning tasks such as end-to-end computation (Section 6.1) and term synthesis (Section 5.4) that have an objective linked to the value of the expression. That is why we decided to try to decrease this number by applying a final modification to our algorithm. We temporarily increase the number of operators during the generation phase to $\mathbb{A} \cup \{1, \dots, n\}$, where $1, \dots, n$ are constants (operators with zero arity). These constants are replaced by their representation in unary $s(0), \dots, s^n(0)$ after the term building process is finished. Our algorithm now has two parameters: the size of the term k and the maximal operator n . To generate a range of different terms, we vary those parameters creating 100 classes $(k, n) \in [1, 10] \times [1, 10]$ and take 200 distinct terms per classes. The upper bound on the size of our training set is then $200 \times 100 = 20000$ but many of these classes have fewer than 200 distinct terms. After the final replacement, *train* contains 11990 distinct terms of $\mathbb{T}_{\mathbb{A}}$. This procedure reduces the number of expressions in *train* evaluating to zero to 9.8%.

The two other sets (*valid*, *test*) are created by the same algorithm with an additional equality check to make the three sets disjoint. The validation set and testing set include respectively 10781 and 10180 arithmetical expressions.

Distributions In order to give a better understanding of the formulas contained in the sets described above, we analyze distributions for the training set with respect to the following measures: term size, value and left-outermost proof length. The validation and testing sets have very similar distributions.

Definition The left-outermost proof length, noted $lopl(x)$ adds the costs of the rewrite steps needed to reach a unary normal form using the left outermost strategy for the following term rewriting system: $x + 0 \rightarrow x$, $x + s(y) \rightarrow s(x + y)$, $x \times 0 \rightarrow 0$, $x \times s(y) \rightarrow x \times y + x$. The cost of a rewrite step at position p is $1 + d$ where d is the depth of p where the rewriting steps occurs.

We form classes by regrouping expressions that are equal with respect to the measures. The size of these classes are displayed in Figure 1. The biggest size for an expression is 54. The graphs of the two other measures are truncated. The largest value is 9072 but 91% of the expressions evaluates to a number less or equal to 100. Among the three measures, the *lopl* function gives the widest range of outputs. There are only 5477 expressions (45.7%) for which $lopl(t) \leq 100$. The proofs (sequences of rewrite steps) become so long that we were unable to compute an upper bound for this measure. We have only determined that there are 1% of the terms for which $lopl(t) \geq 1000$. These measures are important for our four tasks as they provide upper bounds for the difficulty of the tasks.

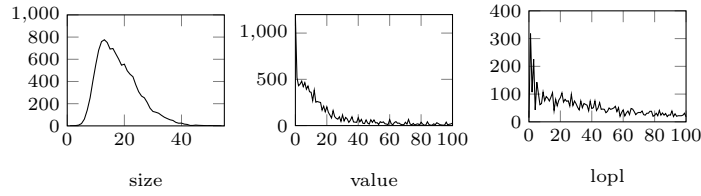


Fig. 1: Number of expressions per class

5 Specification of the Tasks

In this section, we specify the four tasks that are used for experiments with our framework. We describe how to express them in our framework and provide some optimizations that facilitate the learning. The first two tasks use supervised learning while the other two tasks use deep reinforcement learning.

5.1 Truth Estimation

The aim of this task is to teach a TNN to estimate if a propositional formula is true or not. The TNN is trained on a set of 100000 propositional formulas presented in Section 4.1. These propositional formulas contain boolean variables and a direct representation in our TNN would create one neural network operator for each named variables (up to 25 in the dataset). Such a high number of operators is detrimental to the generalization ability of our network as there would be fewer examples in which each variable appears. Therefore, we first number the variables according to their order of appearance in the formula (from left to right). Then the numbered (indexed) variables x_0, x_1, \dots, x_n are encoded as $x, \text{prime}(x), \dots, \text{prime}^n(x)$. In the end, all variables are encoded using two operators x and prime . Thus, propositional formulas are represented by terms built from the set of operators $\{x, \text{prime}, \Rightarrow, \neg, \vee, \wedge\}$. This approach becomes computationally expensive when the number of distinct variables per formula grows large. In this particular situation, architectures such as the graph neural networks presented in [33] are preferable, because their graph structure encodes positions of variables.

5.2 End-to-end Computation

This task as well as the following two are performed on the arithmetical datasets presented in Section 4.2. The aim of the task is to compute the first four bits of the value x of an arithmetical expression. In other words, the TNN should learn to output the binary representation of $x \bmod 16$. Ignoring the possible decoding step performed by the head network, to generate the first bit (reasoning modulo 2) the arithmetical operators $0, s, +, \times$ have to behave respectively like the boolean operators *false*, *not*, *xor*, *and*. The bottom-up architecture of the TNN is perfectly suited for this task as it is a natural way to evaluate an expression. And since the knowledge of the structure of the formula is hard-coded in the tree structure, we expect the TNN to generalize well.

5.3 Term Rewriting

The goal of this task is to produce a sequence of rewrite steps from an arithmetical expression t_0 to a unary number. The task relies on the following equations which are a subset of Robinson axioms: $\forall x. x + 0 = x$, $\forall x y. x + s(y) = s(x + y)$, $\forall x. x \times 0 = 0$, $\forall x y. x \times s(y) = x \times y + x$. From these four axioms, we construct seven rewrite rules using both directions of the equalities. The rewrite rule $0 \rightarrow x \times 0$, which is not necessary to solve our problem, is omitted. The reason is that it would require us to create a ground term for x and we prefer to experiment with term synthesis in a separate task (see Section 5.4). This term rewrite system however does not allow us to define directly a finite set of moves for our search problem. Indeed, a rewrite rule can often be applied at many different positions of a term. And our TNN architecture for approximating the policy does not support an arbitrary number of outputs. To fix the issue, we introduce a tagging operator tag that indicates the only position where rewriting is possible in the current state and add rules to move the tagging operator in the term structure. After this modification, the starting state is $tag(t_0)$. There is only one winning state $tag(s^{value(t_0)}(0))$ and no losing states. The application of a rewrite rule restores the tag to the root of the expression. The additional argument move a_1 (respectively a_2) shifts the tag down to the left (respectively right) argument.

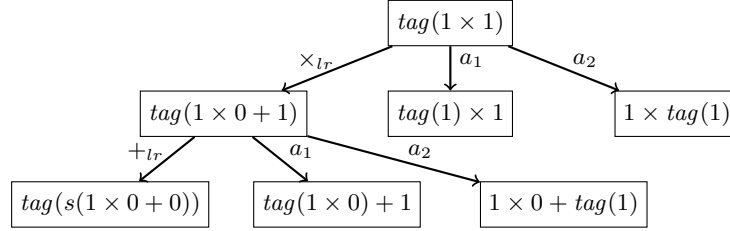


Fig. 2: Search tree for the rewriting problem with starting state $tag(1 \times 1)$.

Given an arithmetical expression t_0 , the difficulty measure associated with this task is $lopl(t_0)$ (defined in Section 4.2). This is an upper bound for the derivation length.

5.4 Term Synthesis

The aim of this task is to find a term whose value is equal to a starting term. This defines a simple problem that requires term synthesis. It can be expressed as finding a witness for the following existential theorem $\exists t. t_0 = t$ where t_0 is a given arithmetical expression taken from our dataset and $=$ is the predicate checking if the value of t_0 and t are equal. The term t to synthesize is built starting from the root. At each step, the predictor chooses the operator it wants to imitate. In the general case, if an operator has two arguments (or more), the left argument is constructed first. For the TNN to know for which initial term

t_0 it has to construct an equal term, the state will always include t_0 as the left-hand side and the term to be synthesized as the right-hand side of an equality. The operator X is a placeholder for an uncompleted sub-tree. Altogether, the language of the TNNs is enriched by two extra operators X and $=$. The starting state is $t_0 = X$. The search ends when the state $t_0 = t$ does not contain any X . The end state is winning if $value(t_0) = value(t)$ and losing otherwise. For each move in $\{0, s, +, \times\}$, the replacement rule applied is respectively $X \rightarrow 0$, $X \rightarrow s(X)$, $X \rightarrow X + X$ and $X \rightarrow X \times X$. A search tree build for this problem is shown in Figure 3.

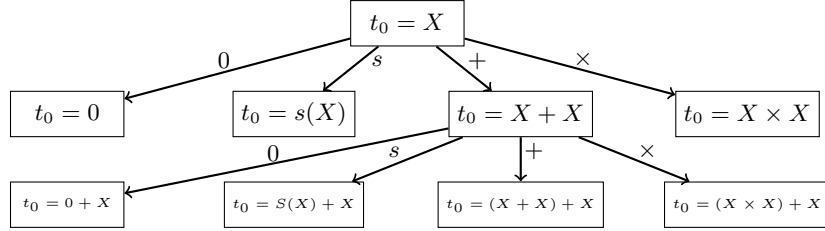


Fig. 3: Search tree for synthesis task from the starting state $t_0 = X$

An upper bound for the minimal length of a solution is $size(t_0)$ since copying the term t_0 is a possible solution to the task. Another upper bound is the $value(t_0) + 1$ achieved by constructing the term $s^{value(t_0)}(0)$. By combining the two results, we obtain an upper bound of $\min(value(t_0) + 1, size(t_0))$. For this task, we set the difficulty measure to $mvs(t_0) = \min(value(t_0), size(t_0))$. The minor difference does not affect search attempts as the limit in the number of big steps is fixed to twice the difficulty.

6 Results

The search phases were executed on 16 CPUs, with one core per search attempt. Training a TNN was performed on 1 CPU per parameters in the tuning phase and 4 CPUs in the other experiments. The code for the framework and the experiments is available in the github repository for HOL4.² The most recent updates can be seen on the development branch under the `src/AI` directory.

In the supervised learning experiments, we compare the efficiency of our TNN architecture and implementation with other approaches. The end-to-end computation tasks is also where parameters of the TNN and the training schedule are tuned. The two reinforcement learning experiments then demonstrate how the reinforcement learning framework is able to gradually learn a task by recursive self-improvement using only minimal human guidance in the form of the difficulty measures.

² <https://github.com/HOL-Theorem-Prover/HOL>

6.1 Supervised Learning Tasks

The training parameters are tested by performing an exhaustive grid search for the values of the following parameters: the learning rate, the batch size, the number of layers (3 stands for one hidden layer) and the dimension of the embeddings. The range of these values is indicated in Table 1. The 24 training runs are executed in parallel for 400 epochs. Table 1 shows the 5 sets of parameters that resulted in an accuracy above 90% on the validation set.

Table 1: Parameter tuning of the TreeHOL4 predictor (left) and prediction accuracy (right) on the end-to-end computation task

valid	train	learn. rate	batch	layers	dim.	Predictors	train	test
		(0.1,0.05,0.02)	(16,64)	(3,5)	(12,16)			
96.9	99.8	0.02	16	3	12	TreeHOL4	98.5	94.3
94.8	99.8	0.1	64	3	16	NMT [32]	100.0	77.2
94.6	99.8	0.05	64	3	16	NearestNeighbor [11]	100.0	11.7
94.3	99.8	0.02	16	3	16	LibLinear [13]	84.4	18.3
92.1	97.9	0.05	64	3	12	XGBoost [7]	99.5	16.8

The logs of the run for the best parameters show that almost all the training is achieved during the first 100 epochs. To save time, this number will be used along with the best parameters for all subsequent training.

In the right part of Table 1, we compare our TNN predictor with feature based predictors. These predictors are quite successful in the premise selection task in ITP Hammers [3]. However, experiments with these predictors are performed with a standard set of syntactical features consisting of all the subterms of the arithmetical expressions. This requires almost no engineering. The accuracy of these predictors on the test set is only slightly better than random (6.25%). The obvious reason is that it is very difficult to compute the value of an expression simply by comparing its subterms with features of terms in the training set. This highlights the need of some feature engineering for these predictors. The slight decrease in performance observed for the TNN architecture during the test phase compared to the tuning phase is due to the reduction of the number of epochs to 100. As a final comparison, we test the deep learning recurrent neural model NMT with parameters taken from those shown as best in the informal-to-formal task [34]. This is a sequence-to-sequence model with attention, typically used for machine translation. This is a more general approach compared to ours, and it may be able to learn the structure of the tree even if represented as a sequence.³. However, despite the perfect training accuracy, the testing accuracy of NMT is below the TNN.

Table 2 compares the results of our TNNs on the truth estimation task with the best neural network architectures for this task.

The first three architectures are the best extracted from the table of results in [12]. The first one is a tree neural network similar to ours which also indexes

³ We use prefix notation for representing the terms as sequences for NMT

the variables. A major difference is that we use the *prime* operator to encode variables while they instead rely on data augmentation by permuting the variable indices. The second one replaces feedforward network by LSTMs. The third architecture bases its decision on simultaneously using multiple embeddings for boolean variables. That is why this architecture is named PossibleWorld. In contrast, the TopDown architecture [8] inverts the structure of the TNNs, and combines the embedding of boolean variables (that are now outputs) using recurrent networks. Because the names of the variables are implicit in the structure of the graph, this architecture is naturally invariant under variable renaming. The results on the test set demonstrates that our implementation of TNN is at least as good the one in [12] as it beats it on every test set. Overall, the more carefully designed architectures for this task (PossibleWorld and TopDown) outperform it. One thing to note is that these architectures typically rely on a much larger embedding dimension – up to $d = 1024$ for the TopDown architecture – expected to require much longer training schedule. We are planning to train networks multiple times during a reinforcement learning loop and this is the main reason why we preferred the faster and smaller networks. However, our TNN implementation rivals with the best architectures on the exam dataset which consists of 100 small examples with few variables extracted from textbooks.

Table 2: Accuracy on the test sets of the truth estimation tasks

Architecture	easy	hard	big	mass.	exam
Tree	72.2	69.7	67.9	56.6	85.0
TreeLSTM	77.8	74.2	74.2	59.3	75.0
PossibleWorld	98.6	96.7	93.9	73.4	96.0
TopDown	95.9	83.2	81.6	83.6	96.0
TreeHOL4	86.5	77.5	79.6	59.7	98.0

6.2 Reinforcement Learning Tasks

The results for both reinforcement learning tasks are presented together. Figure 4 shows the level achieved after x generations for each task.

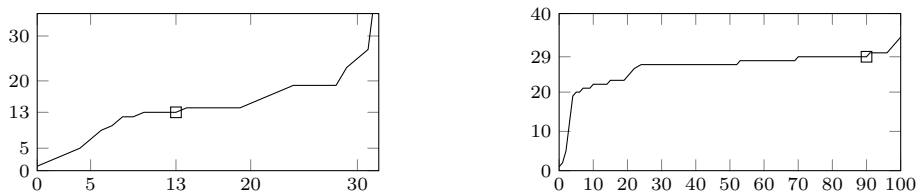


Fig. 4: Level achieved (y-left) at each generation (x). The term rewriting run is shown on the left and the term synthesis run on the right.

The analysis of the left-outermost proof length shows that some expressions might require very long derivations. In order for our algorithm to finish in a

reasonable time, we decided to restrict the training set for the rewriting tasks to terms with $lopl$ less than 89. This forms 13 levels, hence the total number of problems for our training run is $13 \times 400 = 5200$ which amounts to 43.4% of the training set. For the synthesis task, the 11600 training problems are regrouped to form 29 levels. The hardest 300 problems are not enough to form a 30th level and thus are not considered. In Figure 4, the square indicates the generation at which the maximum level is achieved first. After this point, all levels are equal to the maximum level and the upward slope only shows how frequently the maximum levels passed. Since leveling up during the exploration phase restarts this phase, multiple levels may be passed during one generation. In the rewriting task at generation 31, the exploration phase consistently solves more than 95 percent of the randomly selected problems, therefore a generation 32 network is never trained.

During the reinforcement runs, we stored the weights of the best TNNs at each generation. To show the gradual learning progress, we select the TNNs from the key generations. The gen_0 TNN provides a baseline as it has its weights randomly initialized. The percentage of test problems solved is shown in Table 3. For the rewriting task, we restrict the test set to the same range as the training set. This creates a set of 3791 problems noted $T_{\leq 89}$. We also test some extrapolation property, by evaluating the performance of the TNNs on the set T_{90-130} of 692 terms t for which $90 \leq lopl(t) \leq 130$. To determine how performance scales as the number of search steps increases to the results we run tests with different numbers of simulations (sim). Since we are evaluating thousands of problems, we did not test the performance for a larger number of simulations but we believe that it would provide another increase in the success rate. For a comparison, to get the best performance, chess engines such as **LeelaChessZero** [26] run with millions of simulations during computer chess tournaments. In both tasks, with a large number of simulations, the final TNN is better than the intermediate TNN considered. This is interestingly not the always case with few simulations. Considering the way levels are created, it is possible that the TNN slowly forgets how to solve easy problems in favor of better guidance for harder ones. And since mostly easy problems are solved with few simulations, this is reflected in the final numbers. Moreover with many simulations, the fully trained TNN can compensate for its limitations on shallow problems.

Table 3: Percentage of test problems solved

Rewriting task						Synthesis task			
sim	$T_{\leq 89}$			T_{90-130}		10180 problems			
	gen_0	gen_{13}	gen_{31}	gen_{13}	gen_{31}	sim	gen_0	gen_{10}	gen_{99}
1	8.4	81.3	79.6	40.6	62.1	1	10.1	32.8	21.7
16	9.0	85.0	83.7	46.1	66.3	16	10.1	45.2	41.3
160	16.2	87.9	90.7	47.0	71.2	160	13.3	71.9	80.4
1600	18.8	90.8	95.4	51.6	78.6	1600	14.5	70.2	92.1

The performance of the secondary minimization objective on these testing runs is displayed in Table 4. The compression rate for an expression t_0 is computed by dividing the length of the derivation (number of big steps) by $lopl(t_0)$ for the rewriting task and by $size(t_0)$ for the synthesis task. This represents how much better is the strategy discovered by our framework compared to designed strategies: left-outermost for rewriting and copying for synthesis. The ratio presented in Table 4 is the average of the compression rate of successful attempts. The second part of the table shows that problems with larger $lopl$ can be compressed more. The random TNN at generation 0 on the synthesis task has an extraordinary compression rate of 0.09 with 1600 simulations because it is only able to solve problems that have a short solution. In order to compare the compression performance fairly across all parameters, we recalculate the averages by including failed attempts and giving them a compression rate of 2 which is the worst compression rate that a successful attempt can achieve. This fairer compression rate (shown in parentheses) reveals that more generations and more simulations help.

Table 4: Average compression rate on the test set

Rewriting task				Synthesis task			
3791 problems of $T_{\leq 89}$				10180 problems			
sim	gen ₀	gen ₁₃	gen ₃₁	sim	gen ₀	gen ₁₀	gen ₉₉
1	1.00 (1.92)	0.90 (1.11)	0.88 (1.11)	1	0.07 (1.81)	0.39 (1.47)	0.29 (1.63)
16	1.21 (1.93)	0.90 (1.06)	0.88 (1.06)	16	0.07 (1.81)	0.45 (1.30)	0.46 (1.36)
160	1.58 (1.93)	0.89 (1.02)	0.90 (1.00)	160	0.09 (1.74)	0.70 (1.07)	0.74 (0.99)
1600	0.95 (1.80)	0.88 (0.99)	0.88 (0.93)	1600	0.09 (1.72)	0.57 (0.99)	0.73 (0.82)

7 Related Work

The related work can be classified into three categories: machine learning guidance inside ATPs, learning assisted reasoning in ITPs and neural network models for encoding formulas. We present the most promising projects in each category separately. Their description shows how they compare to our approach and influence our methods.

First, the work that comes closest to achieving our end goal of a competitive learning-guided theorem prover is described in [21]. There, a guided MCTS algorithm is trained with reinforcement learning. Its objective is to prove first-order formulas from Mizar [18] problems using a connection-style search [25]. The experiments show that gradual improvement stops on the test set after the fifth generation. This is probably for two reasons: the small number of problems relative to the diversity of the domains considered and the inherent limitations of the feature-based predictor [7] they rely on. Another approach is to modify state-of-the-art ATPs by introducing machine-learned heuristics to influence important choice points in their search algorithms. A major project is the development of ENIGMA [19,9,20] which guides given clause selection in E-prover [27]. There, a fast machine learning model is trained to evaluate clauses from their

contribution to previous proofs. A significant slowdown detrimental to the success rate of **E-prover** occurs when trying to replace the fast predictors by deep neural networks [24].

Second, our work aims to ultimately bring more automation to ITP users. **Hammers** [3] rely on machine learning guided premise selection, translation to first-order and calls to external ATPs to provide powerful push button automation. An instance of such a system is implemented in **HOL4** [15]. Its performance on induction problem is limited by the encoding of the translation. To solve this issue, the tactical prover **TacticToe** [16,17], also implemented in **HOL4**, learns to apply tactics extracted from existing proof scripts. It can perform induction on variables when an induction tactic has been defined for the particular inductive type. Yet, it is currently limited by its inability to synthesize terms as arguments of tactics.

Third, the search for suitable deep learning models for learning task on mathematical formulas has recently become an interesting subject for the machine learning community. A first experiment asks if a neural network can learn propositional entailment [12]. In this work, they design an architecture that facilitates learning by letting the network imagine multiple sets of assignments for the boolean variables in the problem. An architecture that propagates embeddings from the root of the formulas to variables in the leafs is proposed to solve the same task in [8]. A graph neural network that encodes a higher-order formulas is presented in [33] and evaluated on a premise selection task. A message passing neural network is trained to behave as a SAT-solver in [28]. The graph structure of the neural network connects literals appearing in the same clause and between complementary literals.

8 Conclusion and Future Work

Our framework exhibits good performance on the four tasks related to theorem proving. The tasks that use supervised learning were used to evaluate the performance of our TNN implementation and to compare it with other machine learning predictors. The TNNs were then used to guide the more general deep reinforcement learning algorithm on the remaining tasks. This approach showcases how self-learning can solve a task by gathering examples from exploratory searches. Compared to supervised learning, this self-learning approach does not require an oracle and is able to produce verifiable search traces.

In the future, we intend to test this reinforcement learning framework on many more tasks and test the possibility of joint training [35]. One domain to explore consists of tasks on higher-order terms such as: beta-reduction, predicate synthesis or higher-order unification. Another interesting development is to apply the ideas of this paper to the **TacticToe** framework. A direct application would give the tactical prover the ability to synthesize the terms appearing as arguments of tactic. More generally, the term synthesis task and the term rewriting could be modified to respectively perform tactic synthesis and tactical proof search in **TacticToe**.

References

1. David Auger, Adrien Couëtoux, and Olivier Teytaud. Continuous upper confidence trees with polynomial exploration - consistency. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part I*, pages 194–209, 2013.
2. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.
3. Jasmin C. Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.
4. Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pages 131–146, 2010.
5. C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.
6. Lukas Bulwahn. The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, pages 92–108, 2012.
7. Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 785–794, 2016.
8. Karel Chvalovský. Top-down neural model for formulae. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.
9. Karel Chvalovský, Jan Jakubuv, Martin Suda, and Josef Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, pages 197–215, 2019.
10. Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In Maria Paola Bonacina, editor, *Conference on Automated Deduction (CADE)*, volume 7898 of *LNCS*, pages 392–406. Springer, 2013.
11. Thomas M. Cover and Peter E. Hart. Nearest neighbor pattern classification. *IEEE Trans. Information Theory*, 13(1):21–27, 1967.
12. Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. Can neural networks understand logical entailment? In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.
13. Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, 2008.
14. Laurent Fribourg. A superposition oriented theorem prover. *Theor. Comput. Sci.*, 35:129–164, 1985.

15. Thibault Gauthier and Cezary Kaliszyk. Premise selection and external provers for HOL4. In Xavier Leroy and Alwen Tiu, editors, *Conference on Certified Programs and Proofs (CPP)*, pages 49–57. ACM, 2015.
16. Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. TacticToe: Learning to reason with HOL4 tactics. In Thomas Eiter and David Sands, editors, *Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 46 of *EPiC*, pages 125–143. EasyChair, 2017.
17. Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. Learning to prove with tactics. *CoRR*, 2018.
18. Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a nutshell. *Journal of Formalized Reasoning*, 3(2):153–245, 2010.
19. Jan Jakubuv and Josef Urban. ENIGMA: efficient learning-based inference guiding machine. In *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*, pages 292–302, 2017.
20. Jan Jakubuv and Josef Urban. Hammering Mizar by learning clause guidance. *CoRR*, abs/1904.01677, 2019.
21. Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olsák. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, pages 8836–8847, 2018.
22. Eliyahu Kiperwasser and Yoav Goldberg. Easy-first dependency parsing with hierarchical tree LSTMs. *TACL*, 4:445–461, 2016.
23. Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J. Smola. Efficient mini-batch training for stochastic optimization. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 661–670, 2014.
24. Sarah M. 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, Maun, Botswana, May 7-12, 2017*, pages 85–105, 2017.
25. Jens Otten and Wolfgang Bibel. leanCoP: lean connection-based theorem proving. *J. Symb. Comput.*, 36(1-2):139–161, 2003.
26. Pascutto, Gian-Carlo and Linscott, Gary. Leela Chess Zero.
27. Stephan Schulz. E - a brainiac theorem prover. *AI Communications*, 15(2-3):111–126, 2002.
28. Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.
29. David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354–, 2017.
30. Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.
31. Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 1st edition, 1998.

32. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.
33. Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 2786–2796, 2017.
34. Qingxiang Wang, Cezary Kaliszyk, and Josef Urban. First experiments with neural translation of informal to formal mathematics. In Florian Rabe, William M. Farmer, Grant O. Passmore, and Abdou Youssef, editors, *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, volume 11006 of *Lecture Notes in Computer Science*, pages 255–270. Springer, 2018.
35. Andrew M. Webb, Charles Reynolds, Dan-Andrei Iliescu, Henry W. J. Reeve, Mikel Luján, and Gavin Brown. Joint training of neural network ensembles. *CoRR*, abs/1902.04422, 2019.