View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repository of the Academy's Library

Update on FLoP, a Reinforcement Learning based Theorem Prover *

Zsolt Zombori[1], Adrián Csiszárik[1], Henryk Michalewski[3], Cezary Kaliszyk[2], and Josef Urban[4]

[1] Alfréd Rényi Institute of Mathematics, Budapest
[2] University of Innsbruck
[3] University of Warsaw, Google
[4] Czech Technical University in Prague

1 Introduction

The FLoP system was built to allow for experimenting with advanced reinforcement learning (RL) methods applied to guide theorem proving. Its particular focus is to enable learning from and generalizing to long proofs, which is a largely unsolved challenge in theorem proving. The system is very flexible in terms of what it can learn from: even a single training environment (proof) can result in meaningful generalization. On the other hand, FLoP is simplistic in several ways: 1) it learns from manually extracted features, 2) it can overfit in some learning scenarios and 3) its merits have so far been demonstrated only on a very simple dataset. Here we only address 1), the problem of feature extraction.

We present ongoing work that aims to use graph neural networks (gnn) [9] for feature extraction. Gnns have been used to learn features of logic formulae on several supervised tasks, e.g. [3, 7, 8, 2]. However, there are very few experiments with such extractors in a reinforcement learning setting. RL models are typically convolutional and dense networks. Related exceptions are [6] and [5] that use graph extractors. However, while these systems use intertwined iterations of proof search and supervised learning, FLoP uses a pure reinforcement learning loop.

We consider learned formula embedding as a stepping stone for more involved projects that combine machine learning and theorem proving. In Appendix A and B we briefly present two such project proposals planned as future work.

2 Feature extraction

Machine learning models require inputs embedded into some Euclidean space $\mathbb{R}^n$. However, when it comes to learning to guide a theorem prover, states and actions are given as logical formulae and it is highly unclear how to turn them into fixed length vectors. An often used approach is to do some manual feature extraction. Currently, FLoP extracts triples of adjacent nodes in the formula trees as features. These features convey some statistically relevant

*ZZ and AC were supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002) and the Hungarian National Excellence Grant 2018-1.2.1-NKP-00008. HM was supported by the Polish National Science Center grant UMO-2018/29/B/ST6/02959. CK was supported by ERC grant no. 714034 *SMART*. JU was funded by the *AI4REASON* ERC Consolidator grant nr. 649043, the Czech project AI&Reasoning CZ.02.1.01/0.0/0.0/15_003/0000466 and the European Regional Development Fund.

information, however, a large part of the semantics is lost. Another approach that is gaining popularity is to represent formulae as graphs and use graph neural networks to produce an embedding vector. Their promise is to adapt feature extraction both to the data and the problem, i.e., to produce an embedding that best fits the current learning task.

3 Embedding with Graph Neural Networks

A gnn takes a labelled graph as input. Each node has some *inital embedding* vector. The initial embedding is refined in multiple iterations using a learnable *updater model*: the new embedding is calculated from previous embeddings of its neighbourhood. Hence, it exploits the structure of the graph, allowing information to propagate along the edges. We perform a fixed number of update operations, called *hops* to obtain the final embedding of the nodes. Finally, some *aggregation* operation creates a single embedding of the graph from that of its nodes.

Projects using gnns show a large variance with respect to how the input is turned into a graph. We present our proposed approach by comparing it to two recent variants: [7] and [8, 6].

NeuroSAT [7] embeds propositional formulae in conjunctive normal form (CNF). The resulting graph has 2 kinds of nodes (clauses, literals) and 2 kinds of edges (from literals to their containing clauses, between negated literal pairs). Thanks to the small number of node/edge labels, each kind of interaction is represented by separate neural networks in the update step.

FormulaNet and Graph Embeddings for HOList [8, 6] embed formulae of higher order logic. The graph is the abstract syntax tree of the formula. The number of different symbols that can occur in the input is not bounded, so a single update operation is performed on all nodes. Node type information is preserved in a learnable initial embedding. Function application is curried in the syntax tree, so each node has at most two children, i.e., we only need two types of edges. Identical subexpressions are merged. A major complication, that was not present in [7] is the representation of variables. [8, 6] collapse all variables into a single "VAR" symbol.

FLoP In FLoP, we embed first order formulae in CNF. Our implementation is very similar to that of [6], we start from the syntax tree, with two differences: 1) The initial embedding is a fixed random vector. 2) Variables are not collapsed into a single node. Rather, they are wrapped into a "VAR" function and are normalised to ensure that they are renaming invariant. This setup ensures that the formula is recoverable from the graph: the initial embedding vector of the nth variable (according to a preorder traversal) is the same for all inputs.

4 Graph Embedding in FLoP

FLoP is built on the leanCoP connection tableau calculus, so its current state is given by the set of valid actions and the partial tableau tree, with the following main components: the current goal, the branch leading to the current goal, remaining open goals and the currently applicable lemmas. At each step, a policy network computes a probability distribution over the valid

actions. Each component of the input is currently a hand crafted feature vector, which can be easily replaced with the embedding network described above.

This is an ongoing effort and in our talk we will present first results using graph embeddings in FLoP. As a proof of concept, we have done some supervised experiments that use our embedding network: we collected theorem proving attempts from FLoP training and trained to predict if a (state, action) pair can lead to success. We achieved 100% training accuracy on a training set of 20000 entries. We are working to see how well it generalizes.

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.

[2] Karel Chvalovsky. Top-down neural model for formulae. In *International Conference on Learning Representations*, 2019.

[3] Karel Chvalovský, Jan Jakubuv, Martin Suda, and Josef Urban. Enigma-ng: Efficient neural and gradient-boosted inference guidance for e. In *CADE*, 2019.

[4] The MPTP Challenge. http://www.tptp.org/Seminars/MizarVerification/TheMPTPChallenge.html. Accessed: 2019-05-20.

[5] Miroslav Olšák, Cezary Kaliszyk, and Josef Urban. Property invariant embedding for automated reasoning. *CoRR*, 2019.

[6] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. *CoRR*, abs/1905.10006, 2019.

[7] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. *CoRR*, abs/1802.03685, 2018.

[8] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. 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 2783–2793, 2017.

[9] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks, 2019. cite arxiv:1901.00596Comment: updated tables and references.

Appendix A Project Plan: Bolzano-Weierstrass Theorem

The MPTP Challenge [4] consists of the Bolzano-Weierstrass theorem and its 252 auxiliary lemmas, constituting a relatively small, consistent problem domain. One part of the challenge is to prove the theorem and all lemmas from scratch using in each derivation only basic axioms, hence forcing long proofs. In this setup, we believe that curriculum learning can be very useful and we intend to try to tackle the challenge with FLoP.

Appendix B Project Plan: Backward Hindsight Experience Replay

Hindsight Experience Replay (HER) [1] is a clever approach to alleviate reward sparsity problems in RL environments. Its core idea is to take an unsuccessful exploration trajectory, observe state S that it reached (as opposed to target state T) and then replay the same trajectory, while pretending that the target state is now S. During the replay, the agent is rewarded for reaching the new target.

HER is directly applicable to a theorem proving environment that performs forward reasoning: each theorem proving attempt some valid consequences of the axioms, even if not the target conjecture, so it makes sense to assume the new target in the replay. However, it is not obvious how to do it for backward reasoning.

The aim of our project is to redesign HER for the setting of a backward theorem prover.

B.1 Setup

We want to train a backward theorem prover, i.e., one that starts from a target formula (goal) and at each inference step reduces the current goal to a list of other goals. Once a goal is identical to some axiom or previously known lemma, the goal is closed and we can proceed to try to prove the remaining goals. The proof is complete when all goals have been closed.

B.2 Core Idea

We use Hindsight Experience Replay to provide denser reward to the guidance model. Consider a single theorem proving attempt. If all goals are closed, then we have obtained a proof of the target and we can give positive reward to the policy. If there are some open goals, then we can pretend that those goals were among the initial axioms and give positive reward in this modified setting.

B.3 Components

The system has four components:

1. **Embedder e**: takes a formula and maps it into a vector in $\mathbb{R}^n$. This is most likely a graph neural network.
2. **Aggregator c**: takes a set of formula (axiom) embeddings $e(a_1), e(a_2) \dots e(a_k)$ and maps it into a single aggregate embedding, which represents the conjuction of the formulae. This could be a recurrent neural network, though some permutation invariant solution would be best.
3. **Policy p**: takes a goal embedding **and an aggregate axiom embedding** and returns an action probability distribution.
4. **Value v**: takes a goal embedding **and an aggregate axiom embedding** and returns a scalar value of the goal, **given the axioms**.

These components are trained together, end-to-end.

B.4 Training

We iterate the steps below:

1. Select a problem with goal g_0 and axioms (lemmas) $a_1, a_2, \dots a_k$
2. Compute initial goal embedding $e_{g_0} = e(g_0)$ and aggregate axiom embedding $e_a = c(e(a_0), e(a_1), \dots e(a_k))$
3. Try to prove the goal based on the current policy $p(e_{g_i}, e_a)$
4. Perform a gradient step based on the proving attempt for the value and policy, propagating the gradients all the way to the aggreagator and embedder as well.
5. If the proof attempt failed, i.e., we are left with open goals $og_0, og_1, \dots og_l$, then
   (a) Compute new aggregate embedding $e_{\hat{a}} = c(e(a_0), e(a_1), \dots e(a_k), e(og_0), e(og_1), \dots e(og_l))$
   (b) Replay the same inference steps with the new aggregate axiom embedding in the policy $p(e_{g_i}, e_{\hat{a}})$
   (c) The open goals (og_i) are now axioms, so the proof is complete, hence we give positive reward and perform a gradient step.

B.5 Benefits

- We provide positive reward for every single theorem proving attempt.
- The policy receives a representation of the axiom set (knowledge base) and can make more informed decisions.
- This works with any RL algorithm. There is no need of a DAGGER like setup, with separate phases of data collection and supervised learning.

B.6 Difficulties

- Building a meaningful aggregate embedding of the available knowledge base (all axioms and lemmas) might be hard and might be very slow. Some ideas to address this:
  - Use premise selection to restrict the aggregator to a handful of lemmas.
  - Precompute the aggregate embedding of all the lemmas and only "incorporate" the embeddings of the axioms to the aggregate lemma embedding for each problem
- Different open goals might be related due to sharing some variables. When we add the open goals as new axioms in HER, we have to make sure that the axioms are consistent. E.g. when we have two open goals $f(X)$ and $\neg f(X)$, there is no way to add axioms that satisfy both.