Reinforcement Learning for Guiding the E Theorem Prover

Jack McKeown, Geoff Sutcliffe
University of Miami, Miami Florida, USA
jam771@miami.edu, geoff@cs.miami.edu

Abstract

Automated Theorem Proving (ATP) systems search for a proof in a rapidly growing space of possibilities. Heuristics have a profound impact on search, and ATP systems make heavy use of heuristics. This work uses reinforcement learning to learn a metaheuristic that decides which heuristic to use at each step of a proof search in the E ATP system. Proximal policy optimization is used to dynamically select a heuristic from a fixed set, based on the current state of E. The approach is evaluated on its ability to reduce the number of inference steps used in successful proof searches, as an indicator of intelligent search.

Introduction

An Automated Theorem Proving (ATP) problem is specified as a set of axioms and a conjecture to prove. Saturation-based ATP systems (Bachmair et al. 2001) search for a proof-by-contradiction, by repeatedly performing satisfiability-preserving inferences on the axioms and negated conjecture. The negated conjecture and axioms are typically converted into *clauses* (Nonnengart and Weidenbach 2001) that sound inference rules operate on directly. A proof has been found when the ATP system infers the empty clause, which serves as an explicit witness of a contradiction. This paper explores the use of reinforcement learning for guiding the proof search in the saturation-based theorem ATP system E (Schulz, Cruanes, and Vukmirovic 2019).

E (like most saturation-based ATP systems) does not maintain one expanding set of clauses. Instead, it maintains a set of *processed* clauses and a set of *unprocessed* clauses. The unprocessed set initially contains the axiom and negated-conjecture clauses. The processed set is initially empty. E searches for a proof by repeatedly selecting a *given clause* from the unprocessed set to move into the processed set. The given clause interacts with all the clauses in the processed set to infer new clauses, which are then put into the unprocessed set, modulo redundancy elimination techniques (Bachmair and Ganzinger 1994; Waldmann et al. 2020). This process continues until the empty clause is inferred, a resource limit is reached, or the unprocessed set becomes empty (in which case the conjecture cannot be proven from the axioms). Roughly speaking, some given clauses end up moving the ATP system toward a contradiction, while others simply clutter up the processed and unprocessed sets, making the search for a contradiction more difficult.

Most ATP research focusses on increasing the number of problems solved in a data set with some resource limit (usually time). For instance, this is how the CADE ATP System Competition (CASC) evaluates system submissions (Sutcliffe 2016). In order to improve the performance (including solving more problems) of any search-based AI system, the primary necessity is to prune the search space, to allow more of the space to be searched within the same resource limits; this is a core goal for building intelligent systems (Patel et al. 2019). Intelligent guidance of graph and tree search has been behind some of the most amazing breakthroughs in AI, such as AlphaGo/AlphaZero (Silver, Hubert, and others 2017). Provided that a technique does not reduce the number of problems solved, evaluation according to the reduction of search is central. In saturation-based ATP the amount of space searched can be measured by the number of given clauses selected. Reducing the number of given clauses used to find a proof is a direct indication of intelligence in the proof search. This is the evaluation approach taken here (and as shown in Table 1, there is no meaningful change to the number of problems solved).

Clause Evaluation Functions and Heuristics

E has a set of built-in parameterized *Clause Evaluation Functions (CEFs)* that are used for given clause selection (Schulz and Möhrmann 2016). The following are two examples of CEFs:

- `FIFOWeight(ConstPrio)`
- `Clauseweight(ConstPrio,20,9999,4)`

`FIFOWeight` is a CEF that selects the oldest (least recently inferred) clause. `ConstPrio` is an example of a *priority function*. Priority functions are used to limit the scope of an E CEF to a certain subset of clauses, effectively eliminating many clauses from consideration for selection. `ConstPrio`, however, does not remove any clauses from consideration by the CEF. The `Clauseweight` CEF roughly prefers shorter clauses. The *weight* of a clause is a generalization of symbol count where different symbol types can contribute to the clause weight to different extents. The "20" in the above example refers to the weight of function symbols, the “9999” is the weight of variables, and the “4” is a multiplier for the weight of positive literals (non-negated parts of clauses).

There are also more sophisticated CEFs. For instance, there are some that take similarity with the negated conjecture clauses into account. Further examples of CEFs and their explanations can be found in E’s documentation. While given clause selection is not the only component guiding the proof search in E, it is often considered the most important component.

E also combines CEFs into *heuristics*, which are used to guide its search for a contradiction. A *heuristic* is a set of CEFs along with positive integers representing the frequency with which each CEF should be used. An example of a heuristic would be $(3 * CEF_1,\ 2 * CEF_2,\ 7 * CEF_3)$. Using this heuristic, E would select and process three given clauses according to CEF_1, two according to CEF_2, and then seven according to CEF_3, before looping back to CEF_1. This work describes a reinforcement learning approach that learns when to use each CEF, instead of leaving it up to the fixed scheduling just described.

Reinforcement Learning

Reinforcement learning (RL) (Sutton and Barto 2018) is an area of machine learning concerned with discovering how intelligent agents should act to maximize reward. An agent exists in a particular *state*, from which it takes an *action* and receives a *reward* (which can be positive, zero, or even negative). The possible states and actions as well as the dynamics of how actions lead to new states and rewards, defines the *environment*. The agent has a *policy* that it uses to select actions depending on the current state. The policy is trained to select actions that maximize the sum of rewards the agent will receive from the environment in the future. Typically the future rewards are *discounted*, which means that rewards in the distant future have less impact on learning than rewards in the near future. Once a policy has been learned, it can be used to select actions in future uses. Reinforcement learning is very naturally applied to games (Mnih et al. 2015), but also has important real-world applications (Nagaraj, Sood, and Patil 2022; Bojarski et al. 2016). RL training from human feedback is also used as a part of ChatGPT (Christiano et al. 2017), in order to get it to act like a useful assistant and not simply as a language model.

During training a reward is given after each action. However, agents have difficulty learning to assign appropriate credit/blame to the actions that cause positive/negative rewards in the future (Minsky 1961). A common solution is to add more fine-grained rewards to incentivize helpful short-term behavior. This is called *reward shaping*. Reward shaping has to be done carefully as it can cause RL agents to fixate on short term rewards, ignoring the true objective of the environment.

Reinforcement Learning for E

The RL environment for E is its theorem proving process including selecting a given clause, performing inferences, redundancy elimination, and placing the inferred clauses into the unprocessed set. This process changes E’s state. Given clause selection within E can be posed as an RL problem by deciding on the state representation, actions, and rewards. For these decisions, it is important to strike a balance between expressivity and feasibility. For instance, with a state representation that is too simple, the RL agent might be unable to learn to recognize situations where certain actions should be preferred. On the other hand, with a state representation that is too complex, training may be prohibitively slow, or require a larger number of proof attempts. While deciding how to represent E’s states, actions, and rewards, it is also important to consider how the policy will be parameterized, and what RL algorithm will be used to learn those parameters.

Keeping all of these factors simple is important as well. E’s given clause selection loop is very efficient and fast. Adding an expensive neural network evaluation into the loop would reduce the rate at which E can process clauses. Any guidance gained by RL must be sufficiently helpful to make up for the time it takes to compute. The representations chosen in this work are very simple. In addition to being faster, simple representations are chosen to stabilize the learning, make it quicker to train, and increase generalization.

States

It is desirable that all information that is relevant to choosing an action should be in a state. In E, this would mean including every processed and unprocessed clause along with the history of their derivations. This would be a very complex state, and processing that much state information would be prohibitively slow. At the other end of the spectrum, previous work in ATP has used a static state of only the negated conjecture clauses (Crouse et al. 2020; Chvalovsky et al. 2019). This work strikes a middle ground by using a simple but dynamic state of five features:

1. The number of given clause selections so far.
2. The number of clauses in the processed set.
3. The number of clauses in the unprocessed set.
4. The average symbol count of clauses in the processed set.
5. The average symbol count of clauses in the unprocessed set.

While this state representation is simple, it enables the agent to choose different CEFs in different states. One benefit of this state representation is that it avoids the cumbersome and dataset-dependant task of learning clause representations directly (Crouse et al. 2020).

Actions

The policy must guide the selection of an unprocessed clause as the given clause. Doing this directly is difficult because the policy has to be aware of all the unprocessed clauses, and needs to perform computation proportional to the size of the unprocessed set. That approach has been used successfully

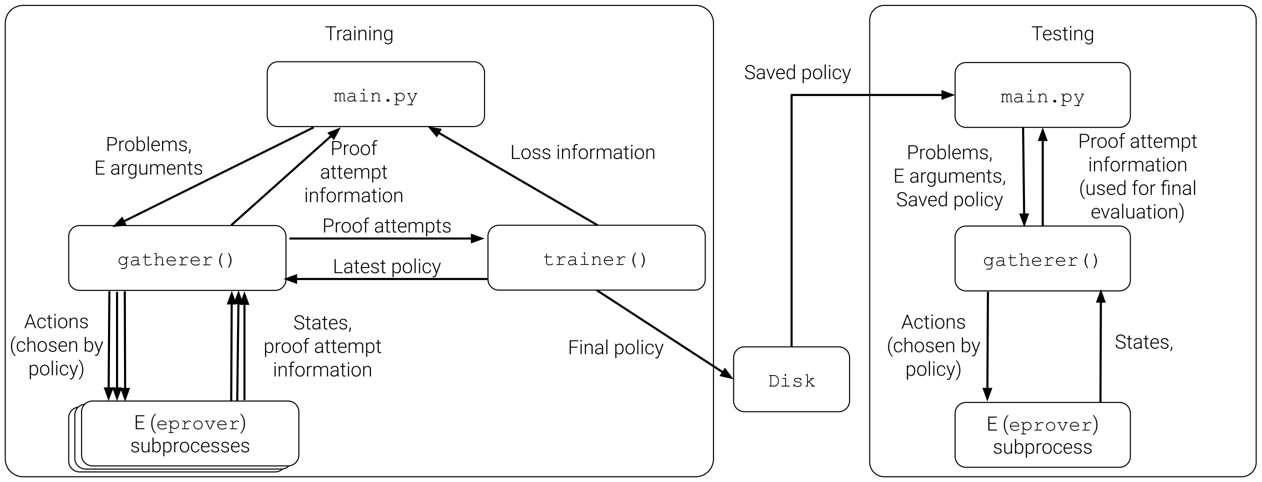

Figure 1: Training and Testing Architecture

in ATP (Abdelaziz et al. 2021), but it is complicated and difficult to implement correctly. Aygün et al. (Aygün, Orseau, and others 2021) directly chose given clauses, used negated conjecture clauses as context, used supervised learning instead of reinforcement learning, and used hindsight experience replay. Although the experimental results presented in this work do not improve on the impressive results of Aygün et al., they represent a simpler approach that warrants further exploration.

In this work an action is a selection from a set of CEFs, rather than a selection from the set of unprocessed clauses. The chosen CEF is then responsible for selecting the given clause. This can be thought of as choosing a delegate to choose the given clause. This action representation takes some pressure off of the state representation because even selecting the worst CEF would likely lead to the selection of a decent clause[1].

Rewards

Rewards are a bit tricky for given clause selection because it is impossible to know which selections are good or bad until a proof is found. One possibility is giving a reward of zero for all selections except for the one that finishes a proof. However, because of the credit assignment problem, the agent would likely struggle to learn to distinguish good actions from bad.

In this work rewards are assigned only after a finished proof attempt. For failed proof attempts, rewards are all zero. For successful proof attempts, rewards are assigned at each step when a clause that is in the eventual proof was selected. It is a bit atypical for the rewards to be undefined until a proof is found, but since no training happens during a proof attempt it is a sound approach. In order to avoid incentivizing the policy to find the longest possible proof, the reward for each of these selections is $\frac{1}{n}$, where n is the number of selected clauses in the proof. This means that discounted future rewards are always between zero and one.

[1]Unfortunately this also implies that selecting the best CEF likely leads to selecting a clause that is worse than the best possible clause, especially as the unprocessed set grows.

RL Algorithm and Architecture

The reinforcement learning algorithm used is Proximal Policy Optimization (PPO) (Schulman et al. 2017). Generalized advantage estimation is also used to reduce the variance in the policy gradient estimates, as is common in PPO. PPO is an on-policy reinforcement learning method that belongs to a class of methods called "actor-critic methods". In actor-critic methods, there are two separate models: an *actor* and a *critic*. The *actor* is the RL policy (picks an action from a given state) and the *critic* estimates the expected future reward that the agent will receive from that state forward acting according to its policy. The critic is not used during proof attempts, and is used only to reduce the variance of the policy gradient estimate during training.

The actor in this work is a four layer neural network with one output unit for each CEF, with $ReLU(x) = max(0, x)$ for the activations on the hidden layers. Layer normalization and a residual connection are used to help stabilize training, although they have minimal impact for such a shallow network.

The critic in this work is a three layer neural network with one output unit, with ReLU for the activations on the hidden layers. The critic's evaluation is explicitly forced to be between zero and one by using the logistic sigmoid function ($\sigma(x) = \frac{1}{1+\exp(-x)}$) for the activation on the critic's final layer. The hidden layers of both the actor and critic networks were chosen to have 100 units, which was experimentally determined to be a good compromise between expressivity and efficient computation.

A simpler actor network has also been considered, which ignores the state completely and simply samples actions from a learned but fixed categorical distribution. Although this model is simple and naive, it has one interesting advantage: it can be *distilled* into an E heuristic. This distillation is done by mapping each action probability to an E heuristic weight using $w(p_i) = \lfloor p_i * 5/p_{min} \rfloor$, where p_{min} is the smallest action probability. The "5" was chosen experimentally as the smallest integer large enough to reduce the effect of rounding on the weights to an acceptable level, as the learned categorical model is fairly uniform. This strategy ensures that every CEF is still included in the heuristic, with the smallest CEF having a weight of 5.

The training and testing architecture is shown in Figure 1. The training begins with an invocation of `main.py`, which starts a `trainer` process for updating the policy based on proof attempts, and a `gatherer` process that runs E using the latest policy for given clause selection, and gathers proof attempts to pass back to the `trainer`. Multiple instances of E are run concurrently to minimize the time spent by the `trainer` waiting for proof attempt data. In order to ensure that the proof attempts are generated using the latest policy, the `gatherer` waits for an updated policy from the `trainer`. E queries the policy to select actions by communicating with the `gatherer`. Each time E needs to select a given clause, it sends the current state to the `gatherer`, which responds with the index of the chosen CEF. This communication is performed using named pipes that are created by the `gatherer` before each call to E. The path of the named pipes each instance of E should use for communication are provided as an environment variable. E is run with the command line argument `--training-examples=3`, which makes E print the given clauses that end up in the proof. E has also been modified to print every given clause as its selected. Rewards can then be extracted from the E's output by aligning the proof clauses with the list of all given clauses.

During testing, `main.py` is invoked with the saved policy. `main.py` passes the policy to the `gatherer`, which runs one E instance at a time to avoid any potential issue involving CPU scheduling having an effect on proof attempts.

Experiments & Results

The trained policies were evaluated and compared with the performance of E's `--auto` mode. The data used for training and testing was the "bushy" problems from the MPTPTP2078 dataset, which is a TPTP-compliant (Sutcliffe 2017) version of MPTP2078 (Alama et al. 2011). A static set of 20 CEFs to be used as actions was chosen before training: the chosen CEFs are those most frequently used by E's `--auto` mode over the MPTPTP2078 dataset. In addition to the trained policies and E's `--auto` mode, E was also run using a heuristic consisting of these 20 CEFs each with a weight of "1", so that E uses the CEFs in a round-robin fashion. For all experiments other than `--auto` (which ignores flags), E was invoked using command line arguments suggested by E's creator, Stephan Schulz. E's `--auto` mode and the distilled categorical model were run using a time limit of 60 seconds. The learned models were given a bit more time to account for the overhead of the named pipe communication between E and the Python code that runs the models. This time extension was given because the named pipes are simply an implementation detail and could be eliminated.[2]

In order to fairly evaluate the approaches, five-fold cross validation was used. The MPTPTP2078 dataset contains 2078 problems, 415 of which were in each fold's test set. The averaged results from the experiments are shown in Table 1.

The first row shows the number of problems solved (conjectures proved) by each method averaged across the testing folds. The second row shows the number of clauses that were processed during successful proof attempts using that method. All approaches solve roughly the same number of problems. The trained neural network model solved the problems it solved in the fewest number of given clause selections on average.

The last row shows how many fewer given clause selections were used by each approach for problems solved by both that approach and `--auto`. The round-robin approach surprisingly also reduces the average number of given clause selections. This is partially due to the way that the 20 CEFs were selected, and partially due to the command line arguments received from Stephan Schulz. Interestingly, the distilled categorical achieves better results than the original learned categorical model across each row, even marginally beating the trained neural network model in terms of how many fewer given clause selections used than E's `--auto` mode. Perhaps this is an indication that the performance of the learned networks could be improved by biasing their sampling to be more even across time. This could be done by learning a policy that takes recent past actions into account, in addition to the state.

Analysis of the Critic

Figure 2 shows two histograms of the output of the trained critic model. The green histogram with lines sloping up from left to right shows the distribution of the critic's outputs for the initial states of successful proof attempts. The red histogram with lines sloping down from left to right shows the distribution of outputs for failed proof attempts. Every MPTPTP2078 problem is represented in these histograms, but each problem's initial state was evaluated by only the critic that was learned when that problem was in the testing set of the five-fold cross-validation. The green histogram is clearly shifted to the right, indicating that the critic has learned something about the difference between the initial states of successful and unsuccessful proof attempts. The bimodal nature of the histograms is caused by a disagreement between cross validation folds indicating that some folds had harder training data in their training folds than others. Perhaps this information could be used within Monte-Carlo Tree Search (MCTS) (Coulom 2006) to explore regions of the proof search space before committing to them. A similar approach was taken by the tableaux-based ATP system `rlCop` (Kaliszyk et al. 2018), but to our knowledge, MCTS

[2]To come up with an appropriate time extension to compensate for named pipe communication time, E was run with and without pipe communication, but with an immediate response from python. From the results, a timeout of 75 seconds was decided upon.

	–auto	Round Robin	Learned Categorical	Distilled Categorical	Neural Network
Problems Proved	228.2	232.0	231.6	**232.2**	231.3
Given Clauses	4407.8	2329.0	2377.4	2262.6	**2013.0**
Fewer Given Clauses than `--auto`	0	1895.6	1743.6	**1899.0**	1897.6

Table 1: Experimental Results: The best result in each row is bolded.

has not yet been applied to saturation based theorem proving.

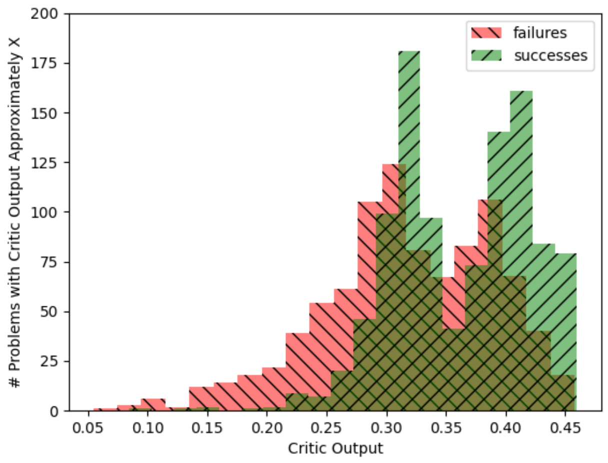

Figure 2: Critic output histogram for initial states

Analysis of the Actor

To analyze the actor, a single representative problem and the learned policy from the first cross validation fold were used to create Figures 3 and 4. `MPT1152+1.p` was chosen randomly from the set of problems that matched these criteria:

1. The chosen actor was not trained on the problem.
2. The problem was proved during testing.
3. The proof attempt was sufficiently long to demonstrate the behavior of the policy over the course of a long proof attempt.

In both Figure 3 and Figure 4, each band represents a different CEF and their heights represent the probability of that CEF being selected by the actor. Figure 3 shows the actor's CEF preferences for the *initial* state of `MPT1152+1.p` over the course of training. (After each learning step, the latest model was saved so that the intermediate models could be revisited after training.) It shows that the actor learned to prefer one particular CEF to use for the beginning of proof searches. At the beginning of training, no CEF was strongly preferred for the initial state, but over the course of training, the policy learned to prefer one CEF strongly for the initial state. Interestingly, in the middle of training the policy also switched from somewhat strongly preferring one CEF to very strongly prefer another one.

Figure 4 shows the CEF preferences of the actor for each state of `MPT1152+1.p` over the course of a proof attempt using the final model. While the x-axes of Figure 3 and Figure 4 are not the same, the first vertical slice of Figure 4 is the same as the last vertical slice of Figure 3 as they both refer to the actor's CEF preferences for the initial state after training. Figure 4 shows that CEF preferences become more uniform as a proof attempt proceeds. This transition to uniform CEF preferences is at least partially due to the entropy term in the PPO loss that encourages exploration in RL environments. When looking at the CEF preferences during proof attempts of other problems, the general pattern is the same, but the transition to uniform CEF preferences takes different numbers of given clause selections. This indicates that the transition to uniform CEF preferences is not induced only by the number of given clause selections, but also by other state features.

Conclusion

We have shown that using a relatively simple state space and simple models, E's proof search can be improved by learning a metaheuristic to select CEFs, rather than trying to learn the selection of given clauses directly. This approach has two main advantages. The first advantage is that it performs a fixed amount of computation per given clause selection, and therefore does not become a bottleneck as the proof search proceeds and the clause sets grow. The second advantage is that it sidesteps the issue of representing clauses as inputs to a neural network (Crouse et al. 2020).

There are many ways to incrementally improve the setup described in this work. The most apparent way is to add more features to the state representation, and add more actions (CEFs). Adding more features to the state representation seems promising because the actor's component of the loss reached its minimum value rather quickly during training. This is likely an indication that the state representation was too simple to enable learning the most effective policy.

Reinforcement learning problems are more formally referred to as Markov Decision Processes (MDPs) (Sutton and Barto 2018). The defining property of MDPs is that the probability of ending up in state s' from taking action a in state s should not depend on any previous states. The simple state representation described above likely does not meet that assumption. This has often not been an issue in practice, as many reinforcement learning problems that are technically

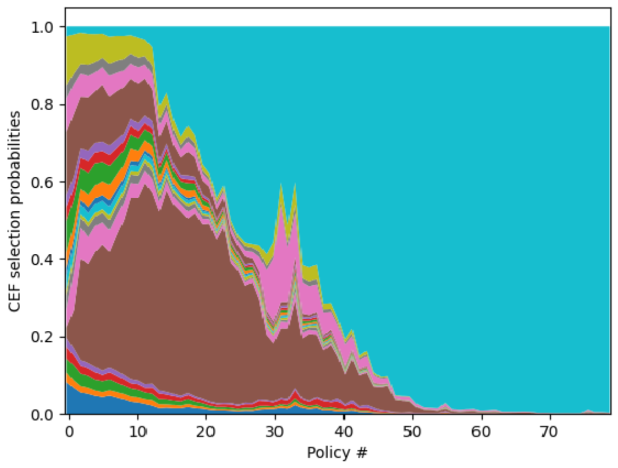

Figure 3: CEF Preferences for the *initial* state of `MPT1152+1.p` during *training*

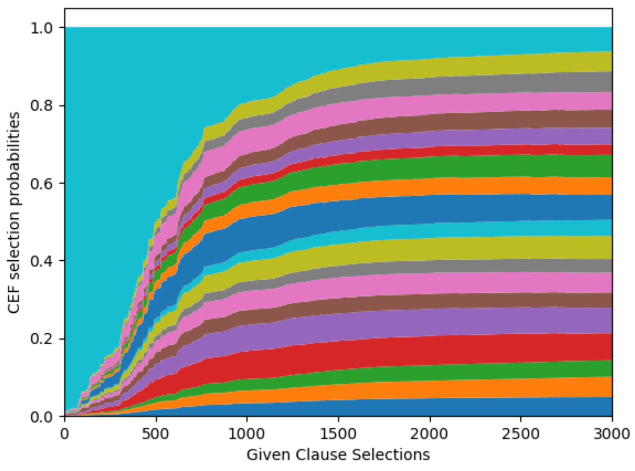

Figure 4: CEF Preferences for *each* state of `MPT1152+1.p` during *testing*

not MDPs have still been solved using standard RL techniques (Mnih et al. 2015). Our approach could be modified to consider the list of states received so far, making the problem a legitimate MDP. This would allow the policy to be adjusted based on how the previous given clause selections affected the sizes of the unprocessed and processed sets.

References

Abdelaziz, I.; Crouse, M.; ; et al. 2021. Learning to Guide a Saturation-Based Theorem Prover. *CoRR* abs/2106.03906.

Alama, J.; Kühlwein, D.; Tsivtsivadze, E.; Urban, J.; and Heskes, T. 2011. Premise Selection for Mathematics by Corpus Analysis and Kernel Methods. *CoRR* abs/1108.3446.

Aygün, E.; Orseau, L.; et al. 2021. Proving Theorems using Incremental Learning and Hindsight Experience Replay.

Bachmair, L., and Ganzinger, H. 1994. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation* 3(4):217–247.

Bachmair, L.; Ganzinger, H.; McAllester, D.; and Lynch, C. 2001. Resolution Theorem Proving. In Robinson, A., and Voronkov, A., eds., *Handbook of Automated Reasoning*. Elsevier Science. 19–99.

Bojarski, M.; Del Testa, D.; Dworakowski, D.; Firner, B.; Flepp, B.; Goyal, P.; Jackel, L. D.; Monfort, M.; Muller, U.; Zhang, J.; Zhang, X.; Zhao, J.; and Zieba, K. 2016. End to End Learning for Self-Driving Cars.

Christiano, P.; Leike, J.; Brown, T.; Martic, M.; Legg, S.; and Amodei, D. 2017. Deep Reinforcement Learning from Human Preferences.

Chvalovsky, K.; Jakubuv, J.; Suda, M.; and Urban, J. 2019. ENIGMA-NG: Efficient Neural and Gradient-Boosted Inference Guidance for E. In *Proceedings of the 27th International Conference on Automated Deduction*, number 11716 in Lecture Notes in Computer Science, 197–215. Springer-Verlag.

Coulom, R. 2006. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games*.

Crouse, M.; Abdelaziz, I.; Cornelio, C.; Thost, V.; Wu, L.; Forbus, K.; and Fokoue, A. 2020. Improving Graph Neural Network Representations of Logical Formulae with Subgraph Pooling.

Kaliszyk, C.; Urban, J.; Michalewski, H.; and Olsák, M. 2018. Reinforcement Learning of Theorem Proving. *CoRR* abs/1805.07563.

Minsky, M. 1961. Steps Toward Artificial Intelligence. In *Proceedings of the IRE*, volume 49, 8–30.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; et al. 2015. Human-level Control Through Deep Reinforcement Learning. 518.

Nagaraj, A.; Sood, M.; and Patil, B. 2022. A Concise Introduction to Reinforcement Learning in Robotics.

Nonnengart, A., and Weidenbach, C. 2001. Computing Small Clause Normal Forms. In Robinson, A., and Voronkov, A., eds., *Handbook of Automated Reasoning*. Elsevier Science. 335–367.

Patel, Y.; Misra, R.; Mishra, M.; and B.S.P., M. 2019. Intelligent Computational Techniques for the Better World 2020: Concepts, Methodologies, Tools, and Applications. In Mishra, M.; B., M.; Patel, Y.; and Misra, R., eds., *Smart Techniques for a Smarter Planet*, number 374 in Studies in Fuzziness and Soft Computing. Springer. 1–17.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms.

Schulz, S., and Möhrmann, M. 2016. Performance of Clause Selection Heuristics for Saturation-based Theorem Proving. In Olivetti, N., and Tiwari, A., eds., *Proceedings of the 8th International Joint Conference on Automated Reasoning*, number 9706 in Lecture Notes in Artificial Intelligence, 330–345.

Schulz, S.; Cruanes, S.; and Vukmirovic, P. 2019. Faster, Higher, Stronger: E 2.3. In *Proceedings of the 27th Interna-*

tional Conference on Automated Deduction, number 11716 in Lecture Notes in Computer Science, 495–507. Springer-Verlag.

Silver, D.; Hubert, T.; et al. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.

Sutcliffe, G. 2016. The CADE ATP System Competition - CASC. *AI Magazine* 37(2):99–101.

Sutcliffe, G. 2017. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning* 59(4):483–502.

Sutton, R., and Barto, A. 2018. *Reinforcement Learning: An Introduction*. The MIT Press.

Waldmann, U.; Tourret, S.; Robillard, S.; and Blanchette, J. 2020. A Comprehensive Framework for Saturation Theorem Proving. In Peltier, N., and Sofronie-Stokkermans, V., eds., *Proceedings of the 10th International Joint Conference on Automated Reasoning*, number 12166 in Lecture Notes in Computer Science, 316–334.