Kalpa Publications in Computing
Volume 21, 2025, Pages 1–7
Proceedings of the 14th and 15th International Workshops on the Implementation of Logics

A Framework for Running Reinforcement Learning Experiments in E

Jack McKeown

University of Miami, Miami, Florida, U.S.A.
jam771@miami.edu

Abstract

As machine learning methods for guiding theorem proving become increasingly popular, it is important that there are frameworks for facilitating rapid development and prototyping of new machine learning models in this context. This paper describes a framework for training and testing reinforcement learning models for guiding given clause selection within the E theorem prover.

1 Introduction

This paper describes a framework for reinforcement learning (RL) with the E theorem prover [3]. This framework includes one Python script for training and testing, multiple scripts for analyzing different aspects of testing results, and a modified version of E to support the interaction between E's given clause selection loop and the RL models in Python. E was chosen because it is a performant and well-known theorem prover. While the framework could be likely be modified to work with other provers, the details of this would come down to how the proposed prover implements its proof search and how it could be modified to interact with the framework.

E normally selects given clauses using a schedule over *Clause Evaluation Functions* (CEFs). This schedule is called a *heuristic* in E. In E's `--auto` mode, the heuristic is chosen based on properties of the input problem. In the presented framework, a heuristic is provided to E, but E has been modified to ignore the schedule. Instead of the schedule, E uses a learned RL policy to select a CEF from the input heuristic based on the current RL state. This happens each time that E needs to select a given clause, and the CEF chosen by the policy is used to select the given clause. The available RL actions therefore map to a fixed set of CEFs that are chosen before training. For the experiments described in Section 5, a set of twenty relevant CEFs was chosen by considering the CEFs most preferred by E's `--auto` mode over the set of problems being solved.

While many aspects of the framework are generic, the currently implemented RL environment represents the *state* of E simply as a list of the following 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 weight (roughly symbol count[1]) of clauses in the processed set.
5. The average weight (roughly symbol count) of clauses in the unprocessed set.

The rewards are all zeros for an unsuccessful proof attempt. For a successful proof attempt, a reward of $1/n$ is given for every given clause that is in the final proof, where n is the number of given clauses in the proof. The given clause selections which don't select a proof clause are given a reward of zero. The $1/n$ reward scaling is to prevent learning to prefer longer proofs.

2 Architecture

Figure 1 shows how training and testing of RL models are performed within the framework. The main entry point for both training and testing is the `main.py` script. `main.py` performs training by default, and is used for testing by providing the `--test` flag. `main.py` supports many command line arguments for customizing its behavior. These command line arguments are parsed using `argparse` from the Python standard library.

During training, `main.py` starts up two subprocesses: a `gatherer` process that performs proof attempts using the latest RL policy and a `trainer` process that learns the policy using the results of the proof attempts. The subprocesses communicate with each other (and with their parent process) using Python multiprocessing queue objects. The parent process oversees the training, saving the proof attempts and latest policy periodically as well as displaying relevant summary information in the dashboard described in Section 4. RL algorithms are referred to as on-policy or off-policy depending on whether the policy being learned is the same as the policy being used to generate training data. When using an on-policy RL algorithm such as Proximal Policy Optimization (PPO), it is important that the training data is generated using the latest policy and that the policy is trained using data generated by the latest policy. This synchronization is handled by the `gatherer` and the `trainer`: the `gatherer` runs only one training batch of proof attempts using its current policy, and then waits for an updated policy from the `trainer`. Likewise, the `trainer` performs only so many training steps on the same proof attempts before waiting for new proof attempts (the number of epochs and batches are hyperparameters of PPO).

During testing, `main.py` starts only the `gatherer`, which uses the final policy from training to guide proof attempts. `main.py` saves the information from these proof attempts to disk. This information can be analyzed/visualized using the tools described in Section 6.

3 Modifications to E

In order to support the external guidance for given clause selection, E has been modified to track the RL state described above. The state is very simple but has the advantage of being able to be updated in constant time for each iteration of the given clause selection loop: the average weights are updated whenever a clause is added or removed from the processed and unprocessed sets instead of being updated by a full pass through these sets before each clause selection. Each time E needs to select a given clause, it sends this tracked state to the `gatherer` via a named pipe. The `gatherer` responds to E with an *action* (an unsigned integer) using

[1]`ClauseStandardWeight()` in E

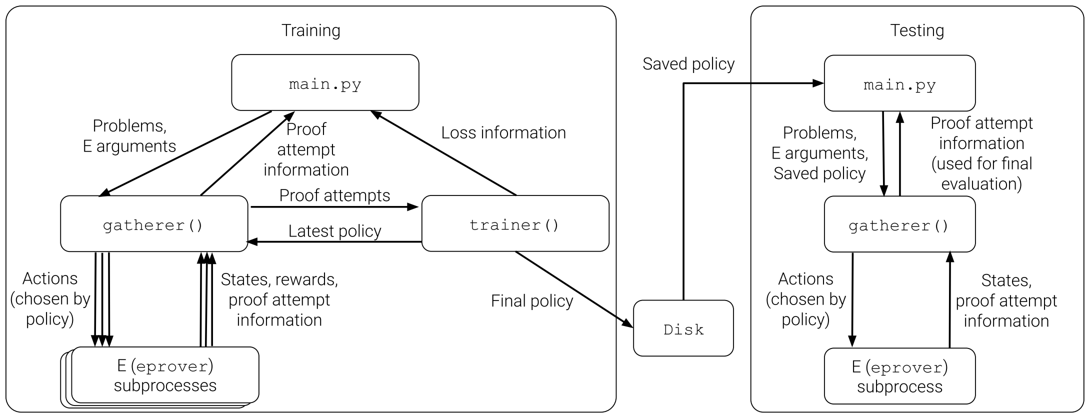

Training
main.py
Problems, E arguments
Proof attempt information
Loss information
gatherer()
Proof attempts
Latest policy
trainer()
Actions (chosen by policy)
States, rewards, proof attempt information
Final policy
E (eprover) subprocesses
Saved policy
Disk
Testing
main.py
Problems, E arguments, Saved policy
Proof attempt information (used for final evaluation)
gatherer()
Actions (chosen by policy)
States, proof attempt information
E (eprover) subprocess

Figure 1: Training and Testing Architecture

another named pipe. E uses this unsigned integer as an index into the list of CEFs and uses that CEF to select the given clause.

While the current representations of states and actions are very simple, this framework can also be adapted to work with different representations. In order to do so, it would be necessary to:

1. Decide upon new representations for states and actions.
2. Define how to serialize states in E (written in C) and deserialize them in Python.
3. Define how to serialize actions in Python and deserialize/interpret them in E.
4. Redefine the RL models accordingly.

4 UI Dashboard

It is helpful to track information during long-running experiments for a variety of reasons. Metrics such as the different PPO loss components can be tracked over time to give insights into the training process, and obvious issues that require a restart of the experiment are discovered early. Platforms like *Tensorboard*[2] and *Weights & Biases*[3] are mature tools for doing this. While these platforms are great for tracking certain supported metrics, tracking custom information is sometimes difficult. They also use a web interface that is tricky to use if the experiments are being run over SSH on a remote server as was the case here — see Section 5. For these reasons, I opted for a more custom terminal-based approach. The *rich* library[4] in Python was used to create a text-based dashboard. Rich supports the creation of titled boxes called panels. Panels can contain text or other panels. Text and panels can both be easily colored to create a visual distinction between dashboard elements.

A `Dashboard` class was created in Python to hold and render all the information being tracked. The class has a master `render` method to print the full dashboard. This method calls

[2]https://www.tensorflow.org/tensorboard/
[3]https://wandb.ai/
[4]https://github.com/Textualize/rich

other methods to return individual panels, and determines a layout for the panels to form the full dashboard. Most of these constituent panel render methods have a corresponding `update` method for updating the dashboard with information to be accessed during rendering. The constituent update methods are called by the `trainer` and `gatherer` processes and the render methods are only called by the master render method, which itself is called in their parent process.[5] In order to make better use of screen space, the panels can be alternated across calls to render.[6]

Line plots are often helpful for visualizing changes to key metrics such as the different components of the PPO loss and the average proof attempt success rate. `Termplotlib`[7] is a simple library for producing line plots as ASCII strings using syntax that is familiar to users of `matplotlib`. While these plots have poor resolution because they are rendered in text, they are still informative.

A dashboard screenshot from a run of `main.py` is shown in Figure 3. The top panel shows the hyperparameters of the current run. This information is pulled from the `argparse` object that stores all of the command line arguments to `main.py`. The various loss components and a running average of them are shown in the upper left panel. The current sizes of the multiprocessing queues used for communication are shown in the lower middle of the dashboard.

The currently implemented line plot panels show the three different components of the PPO loss and the proof attempt success rate over time. A very generally useful panel is shown at the bottom in the middle. This panel prints arbitrary strings sent from the `gatherer` or `trainer` with timestamps, acting as a log for important events. The panel in the upper left summarizes the loss information for the latest training batch as well as an exponentially smoothed average of this information over time. A custom Python context manager for profiling is used throughout `main.py`. This profiler simply keeps track of how much time is spent in various regions of code within the `gatherer` and `trainer` and this information is available as a panel (but it is not included in Figure 3). This profiling information is helpful for debugging and also for estimating hyperparameters that reduce the time that the `gatherer` and `trainer` spend waiting for each other.

The dashboard is fairly straightforward to modify. Custom dashboard panels can easily be made by implementing a method for rendering (it must return a rich `Panel` object) as well as an optional method for updating any relevant state for that panel. The main dashboard render method has to be adjusted to use that newly created panel in its layout. For instance, adding a loss graph to a dashboard could be done by creating `updateLossGraphInfo(info)` and `renderLossGraph()`. The result of calling `renderLossGraph()` has to be incorporated into the full dashboard formed by the main render method.

5 Initial Experimentation

This framework has been used to run experiments training policies using PPO in PyTorch [2]. The details of these experiments are provided in the cited paper. The data for the experiments comes from the "bushy" problems of the MPTPTP2078 dataset, which is a TPTP-compliant [4] version of the MPTP2078 dataset [1]. A round-robin schedule over this set of chosen CEFs was evaluated as a baseline. For these experiments, a remote server was used because it had many CPU cores that could each be used for different calls to E in parallel for generating

[5]Because different processes do not share memory, the updates from the dashboard copies in `trainer` and `gatherer` are sent to the parent process using Python multiprocessing queues.

[6]Like those screens at fast-food restaurants...

[7]https://github.com/nschloe/termplotlib

proof attempts quickly. A neural network policy trained using PPO was compared against this baseline and E's `--auto` mode. In addition to this neural network model, a constant categorical distribution was also learned as a policy. This policy can be *distilled* into an E heuristic by translating probabilities into positive integers such that the resulting E heuristic uses each CEF with roughly the same frequency that would be expected under the random sampling. The results of evaluating each of these policies is shown in Table 1. Each of the models is evaluated in terms of the number of problems solved in the testing set, as well as the number of given clauses required to find the average proof. The results are averaged across the testing sets of a five-fold cross-validation setup. While this simple RL experiment failed to solve significantly more problems, the proofs found were found in fewer given clause selections. This represents a more efficient search for the empty clause.

	–auto	**Round Robin**	**Learned Categorical**	**Distilled Categorical**	**Neural Network**
Problems Solved	228	**232**	231	**232**	231
Given Clauses	4407	2329	2377	2262	**2013**
Fewer Given Clauses than `--auto`	0	1895	1743	**1899**	1897

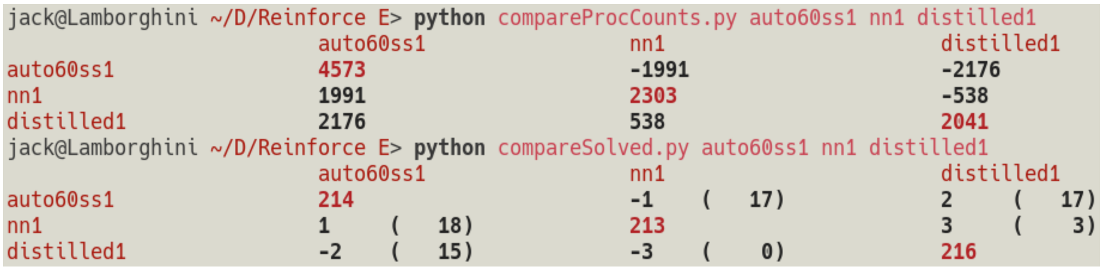
Table 1: Experimental Results: The best result in each row is bolded.

6 Tools for Analysis of Results

During a training or testing run of `main.py`, a tag is provided to be used as the name of that run. These tag names dictate where the data is saved. `main.py` keeps track of the history of proof attempts using an instance of a custom class called `ECallerHistory`. After `main.py` finishes (and occasionally as it runs) this object is saved using `torch.save()`. This is a convenient way to save arbitrary objects since the corresponding call to `torch.load()` reproduces the object that was saved with no need for cumbersome manual parsing. While the saved format enables somewhat quick analysis in IPython or using a Jupyter notebook, a few simple scripts have been written for summarizing particular aspects of training/testing runs:

- `compareArgs.py` - takes in run tags as command line arguments and outputs the command line arguments passed for the corresponding calls to `main.py`.
- `compareProcCount.py` - takes in run tags as command line arguments and outputs a matrix showing how many given clause selections are performed (on average) before a proof is found. Off diagonal numbers are the average number of given clause selections needed by the run corresponding to the row over problems solved by both the run corresponding to the row and the run corresponding to the column.
- `compareSolved.py` - takes in run tags as command line arguments and outputs a matrix showing the number of problems solved by each run on the diagonal and the size of the set difference in problems solved on the off-diagonal.

A screenshot of `compareProcCount.py` and `compareSolved.py` running is shown in Figure 2.



```
jack@Lamborghini ~/D/Reinforce E> python compareProcCounts.py auto60ss1 nn1 distilled1
                          auto60ss1                  nn1                        distilled1
auto60ss1                 4573                       -1991                      -2176
nn1                       1991                       2303                       -538
distilled1                2176                       538                        2041
jack@Lamborghini ~/D/Reinforce E> python compareSolved.py auto60ss1 nn1 distilled1
                          auto60ss1                  nn1                        distilled1
auto60ss1                 214                        -1    (    17)             2     (    17)
nn1                       1     (    18)             213                        3     (     3)
distilled1                -2    (    15)             -3    (     0)             216
```

Figure 2: A screenshot of `compareProcCount.py` and `compareSolved.py`

7 Conclusion

The framework described in this paper aims to enable future research into how reinforcement learned policies can guide given clause selection. It is a simple but extensible framework that is useful for tracking experiments and analyzing results on a remote server. The code is currently available on github at https://github.com/jackeown/Reinforce_E.

The modified version of E is also available at https://github.com/jackeown/eprover in the branch "reinforcement_learning". While this version of E does not receive automatic updates from the official eprover repository, there are few enough changes that it should be relatively easy to merge in future versions of E. The modifications are mostly contained within one source code file: `cco_proofproc.c`.

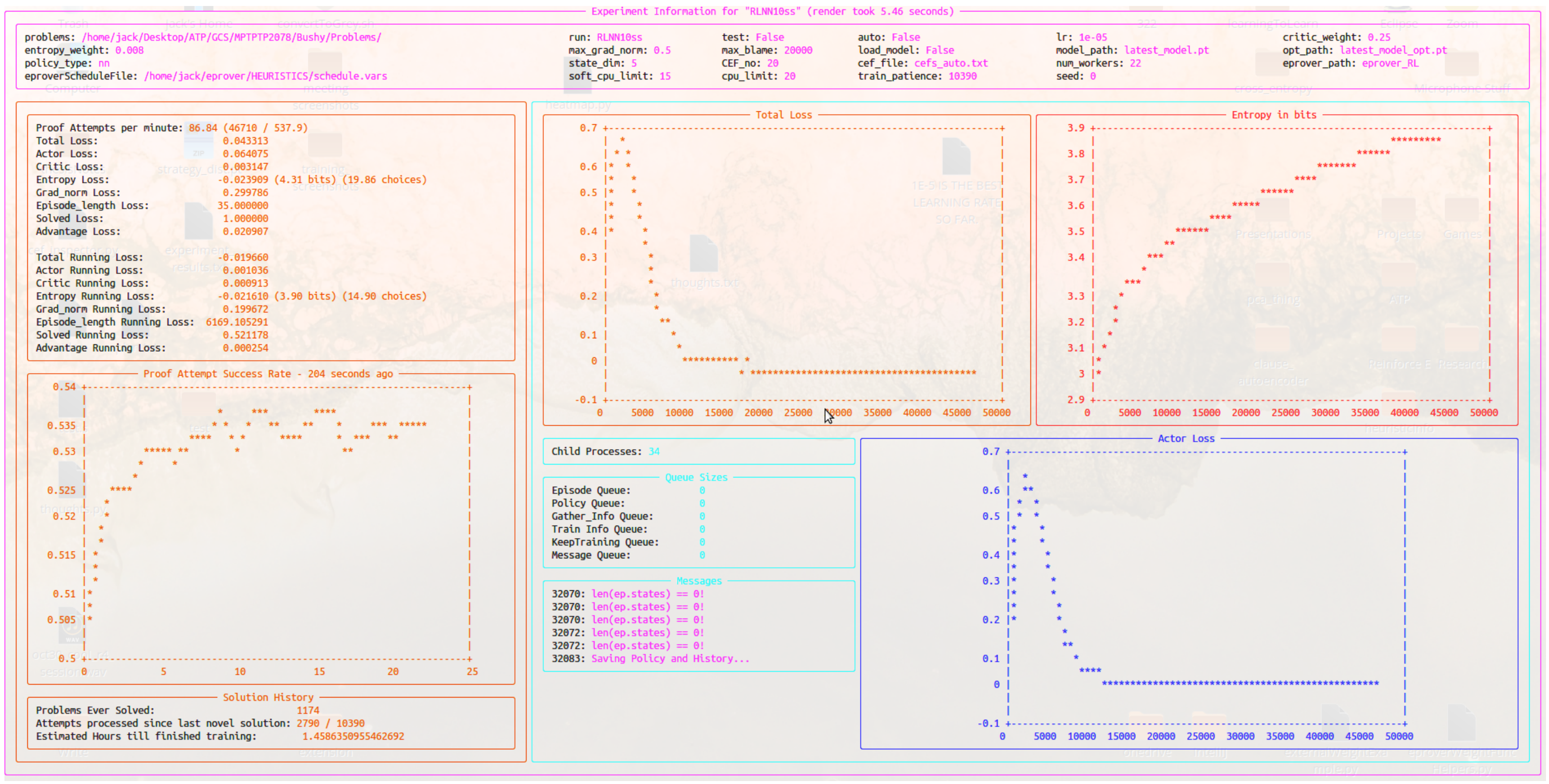
References

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

[2] J. McKeown and G. Sutcliffe. Reinforcement Learning for Guiding the E Theorem Prover. In A. Ae Chun and M. Franklin, editors, *Proceedings of the 36th International FLAIRS Conference*, page To appear, 2023.

[3] S. Schulz, S. Cruanes, and P. Vukmirovic. Faster, Higher, Stronger: E 2.3. In *Proceedings of the 27th International Conference on Automated Deduction*, number 11716 in Lecture Notes in Computer Science, pages 495–507. Springer-Verlag, 2019.

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



```
Experiment Information for "RLNN10ss" (render took 5.46 seconds)
problems: /home/jack/Desktop/ATP/GCS/MPTPTP2078/Bushy/Problems/    run: RLNN10ss           test: False          auto: False                 lr: 1e-05                       critic_weight: 0.25
entropy_weight: 0.008                                              max_grad_norm: 0.5      max_blame: 20000     load_model: False           model_path: latest_model.pt     opt_path: latest_model_opt.pt
policy_type: nn                                                    state_dim: 5            CEF_no: 20           cef_file: cefs_auto.txt     num_workers: 22                 eprover_path: eprover_RL
eproverScheduleFile: /home/jack/eprover/HEURISTICS/schedule.vars   soft_cpu_limit: 15      cpu_limit: 20        train_patience: 10390       seed: 0

Proof Attempts per minute: 86.84 (46710 / 537.9)
Total Loss:                   0.043313
Actor Loss:                   0.064075
Critic Loss:                  0.003147
Entropy Loss:                -0.023909 (4.31 bits) (19.86 choices)
Grad_norm Loss:               0.299786
Episode_length Loss:         35.000000
Solved Loss:                  1.000000
Advantage Loss:               0.020907

Total Running Loss:          -0.019660
Actor Running Loss:           0.001036
Critic Running Loss:          0.000913
Entropy Running Loss:        -0.021610 (3.90 bits) (14.90 choices)
Grad_norm Running Loss:       0.199672
Episode_length Running Loss: 6169.105291
Solved Running Loss:          0.521178
Advantage Running Loss:       0.000254

Solution History
Problems Ever Solved:                              1174
Attempts processed since last novel solution: 2790 / 10390
Estimated Hours till finished training:       1.4586350955462692

Child Processes: 34

Queue Sizes
Episode Queue:          0
Policy Queue:           0
Gather_Info Queue:      0
Train Info Queue:       0
KeepTraining Queue:     0
Message Queue:          0

Messages
32070: len(ep.states) == 0!
32070: len(ep.states) == 0!
32070: len(ep.states) == 0!
32072: len(ep.states) == 0!
32072: len(ep.states) == 0!
32083: Saving Policy and History...
```

Figure 3: UI Dashboard Screenshot