gym-saturation: an OpenAI Gym environment for saturation provers

Boris Shminke

1 Laboratoire J.A. Dieudonné, CNRS and Université Côte d’Azur, France

DOI: 10.21105/joss.03849

Software

- Review
- Repository
- Archive

Statement of need

Current applications of RL to saturation-based ATPs like Enigma (Jakubuv et al., 2020) or Deepire (Suda, 2021) are similar in that the environment and the agent are not separate pieces of software but parts of larger systems that are hard to disentangle. The same is true for non saturation-based RL-friendly provers too (e.g. lazyCoP, Rawson & Reger (2021)). This monolithic approach hinders free experimentation with novel machine learning (ML) models and RL algorithms and creates unnecessary complications for ML and RL experts willing to contribute to the field. In contrast, for interactive theorem provers, projects like HOList (Bansal, Loos, Rabe, Szegedy, & Wilcox, 2019) or GamePad (Huang et al., 2019) separate the concepts of environment and agent. Such modular architecture may lead to the development of easily comparable agents based on diverse approaches (see, e.g. Paliwal et al. (2020) or Bansal, Loos, Rabe, & Szegedy (2019)). gym-saturation is an attempt to implement a modular environment-agent architecture of an RL-based ATP. In addition, some RL empowered saturation ATPs are not accompanied with their source code (Abdelaziz et al., 2022), while gym-saturation is open-source software.

Usage example

Suppose we want to prove an extremely simple theorem with a very basic agent. We can do that in the following way:

```python
# first we create and reset a OpenAI Gym environment
from importlib.resources import files
import gym

env = gym.make("gym_saturation:saturation-v0",)
```
# we will try to find a proof shorter than 10 steps
step_limit=10,
# for a classical syllogism about Socrates
problem_list=[
    files("gym_saturation").joinpath("resources/TPTP-mock/Problems/TST/TST003-1.p"),
],
env.reset()
# we can render the environment (that will become the beginning of the proof)
print("starting hypotheses:")
print(env.render("human"))
# our 'age' agent will always select clauses for inference
# in the order they appeared in current proof attempt
action = 0
done = False
while not done:
    observation, reward, done, info = env.step(action)
    action += 1
# SaturationEnv has an additional method
# for extracting only clauses which became parts of the proof
# (some steps were unnecessary to find the proof)
print("refutation proof:")
print(env.tstp_proof)
print(f"number of attempted steps: {action}"
The output of this script includes a refutation proof found:

starting hypotheses:
cnf(p_imp_q, hypothesis, ¬man(X0) ∨ mortal(X0)).
cnf(p, hypothesis, man(socrates)).
cnf(q, hypothesis, ¬mortal(socrates)).
refutation proof:
cnf(_0, hypothesis, mortal(socrates), inference(resolution, [], [p_imp_q, p])).
cnf(_2, hypothesis, $false, inference(resolution, [], [q, _0])).
number of attempted steps: 6

Architecture

gym-saturation includes several sub-packages:

- parsing (happens during env.reset() in example code snippet)
- logic operations (happen during env.step(action) in the example)
- AI Gym environment implementation
- agent testing (a bit more elaborated version of the while loop from the example)

gym-saturation relies on a deduction system of four rules which is known to be refutationally complete (Brand, 1975):

\[
\frac{C_1 \lor A_1, C_2 \lor \neg A_2}{\sigma(C_1 \lor C_2)}, \sigma = mgu(A_1, A_2) \quad \text{(resolution)}
\]

\[
\frac{C_1 \lor s \approx t, C_2 \lor L[r]}{\sigma(L[t] \lor C_1 \lor C_2)}, \sigma = mgu(s, r) \quad \text{(paramodulation)}
\]

https://doi.org/10.21105/joss.03849
\[
\frac{C \lor A_1 \lor A_2}{\sigma (C \lor L_1)}, \sigma = mgu(A_1, A_2) \quad \text{(factoring)}
\]

\[
\frac{C \lor s \not\approx t}{\sigma (C)}, \sigma = mgu(s, t) \quad \text{(reflexivity resolution)}
\]

where \(C, C_1, C_2\) are clauses, \(A_1, A_2\) are atomic formulae, \(L\) is a literal, \(r, s, t\) are terms, and \(\sigma\) is a substitution (most general unifier). \(L[t]\) is a result of substituting the term \(t\) in \(L[r]\) for the term \(r\) at only one chosen position.

For parsing, we use the LARK parser (Shinan, 2021). We represent the clauses as Python classes forming tree-like structures. gym-saturation also includes a JSON serializer/deserializer for those trees. For example, a TPTP clause

\[
\text{cnf}(a2, \text{hypothesis},
\quad (\neg q(a) | f(X) = X)).
\]

becomes

```
Clause(
    literals=[
        Literal(
            negated=True,
            atom=Predicate(
                name="q", arguments=[Function(name="a", arguments=[])])
        ),
        ),
        Literal(
            negated=False,
            atom=Predicate(
                name="=",
                arguments=[
                    Function(name="f", arguments=[Variable(name="X")]),
                    Variable(name="X"),
                ],
            ),
        ],
    label="a2",
)
```

This grammar serves as the glue for gym-saturation sub-packages, which are, in principle, independent of each other. After switching to another parser or another deduction system, the agent testing script won’t break, and RL developers won’t need to modify their agents for compatibility (for them, the environment will have the same standard OpenAI Gym API).
Figure 1: A diagram showing interactions between four main subpackages of gym-saturation: 1) parsing; 2) logic operations (including the given clause algorithm); 3) OpenAI Gym Env implementation; 4) the agent testing script.

Agent testing is a simple episode pipeline (see Figure 1). It is supposed to be run in parallel (e.g., using GNU Parallel, Tange (2021)) for a testing subset of problems. See the following table for the testing results of two popular heuristic-based agents on TPTP v7.5.0 (trained RL agents should strive to be more successful than those primitive baselines):

<table>
<thead>
<tr>
<th></th>
<th>size agent</th>
<th>age agent</th>
<th>size&amp;age agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>proof found</td>
<td>509</td>
<td>206</td>
<td>688</td>
</tr>
<tr>
<td>step limit</td>
<td>1385</td>
<td>35</td>
<td>223</td>
</tr>
<tr>
<td>out of memory</td>
<td>148</td>
<td>149</td>
<td>148</td>
</tr>
<tr>
<td>5 min time out</td>
<td>6215</td>
<td>7867</td>
<td>7198</td>
</tr>
<tr>
<td>total</td>
<td>8257</td>
<td>8257</td>
<td>8257</td>
</tr>
</tbody>
</table>

- **size agent** is an agent which always selects the shortest clause.
- **age agent** is an agent which always selects the clause which arrived first to the set of unprocessed clauses (‘the oldest one’).
- **size&age agent** is an agent which selects the shortest clause five times in a row and then one time — the oldest one.

‘Step limit’ means an agent didn’t find proof after 1000 steps (the longest proof found consists of 287 steps). This can work as a 'soft timeout.'
Mentions

At the moment of writing this paper, gym-saturation was used by its author during their PhD studies for creating experimental RL-based ATPs.

Acknowledgements

This work has been supported by the French government, through the 3IA Côte d’Azur Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-19-P3IA-0002. This work was performed using HPC resources from GENCI-IDRIS (Grant 2021-AD011013125).

References


