gym-saturation: Gymnasium environments for saturation provers (system description)

Boris Shminke

Université Côte d'Azur, CNRS, LJAD, France

8 Sep 2023

Environments?

- gymnasium is 'an API standard for reinforcement learning (RL) with a diverse collection of reference environments'
- originally developed by OpenAI and known as 'gym'
- then transfered to a non-profit and rebranded
- gym-saturation is a 'third-party environment'

Pre-requisites for this demo

- Python 3.8-3.11
- `pip install gym-saturation`
- or `conda install -c conda-forge gym-saturation`
- Vampire and iProver binaries
- TPTP problems to solve

In [1]:

```
# gymnasium includes 'a diverse collection of reference environments'
import gymnasium as gym
# e.g. the Frozen Lake game
frozen_lake = gym.make("FrozenLake-v1", render_mode="ansi")
observation, info = frozen_lake.reset(seed=11)
print(frozen_lake.render())
```

```
SFFF
FHFH
FFFH
HFFG
```

In [2]:

```python
# termination means reaching the goal (marked `G`)
# truncation means falling into a hole (marked `H`)
# reward is always `0` if the goal is not reached
# here we go down (action number `1`)
observation, reward, terminated, truncated, info = frozen_lake.step(1)
print(reward, terminated, truncated)
print(frozen_lake.render())
```

```
0.0 False False
  (Down)
SFFF
FHFH
FFFH
HFFG
```

```
/home/boris/projects/gym-saturation/venv/lib/python3.11/site-packages/gym
nasium/utils/passive_env_checker.py:233: DeprecationWarning: `np.bool8` i
s a deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):
```

In [3]:

```python
# then we go right and fall into a hole, an episode is truncated
observation, reward, terminated, truncated, info = frozen_lake.step(2)
print(reward, terminated, truncated)
print(frozen_lake.render())
```

```
0.0 True False
  (Right)
SFFF
FHFH
FFFH
HFFG
```

In [4]:

```python
# Vampire binary should be on the PATH
# or one can specify it explicitly
import os

vampire_binary_path = os.path.join(
    os.environ["HOME"],
    ".local",
    "bin",
    "vampire"
)
```

In [5]:

```
# a very simple set theory problem
tptp_problem_path = os.path.join(
    os.environ["HOME"],
    "data",
    "TPTP-v8.2.0",
    "Problems",
    "SET",
    "SET001-1.p"
)
```

In [6]:

```python
# to remind, the Frozen Lake game
frozen_lake = gym.make("FrozenLake-v1", render_mode="ansi")
# with `gym-saturation` we create prover environments in the same way
import gym_saturation

vampire = gym.make(
    "Vampire-v0",
    render_mode="ansi",
    # if Vampire binary is on the PATH, this is not necessary
    prover_binary_path=vampire_binary_path,
    # an episode will be truncated
    # when we have more than this number of clauses in total
    max_clauses=30
)
```

In [7]:

```python
# we can set a task from a file
vampire.unwrapped.set_task(tptp_problem_path)
observation, info = vampire.reset()
print("Action mask:", observation["action_mask"])
print(vampire.render())
```

```
Action mask: [1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0]
cnf(1, lemma, member(X0,X2) | ~subset(X1,X2) | ~member(X0,X1), inference
(input, [], [])).
cnf(2, lemma, member(member_of_1_not_of_2(X1,X2),X1) | subset(X1,X2), inf
erence(input, [], [])).
cnf(3, lemma, subset(X1,X2) | ~member(member_of_1_not_of_2(X1,X2),X2), in
ference(input, [], [])).
cnf(4, lemma, subset(X1,X2) | ~equal_sets(X1,X2), inference(input, [],
[])).
cnf(5, lemma, subset(X1,X2) | ~equal_sets(X2,X1), inference(input, [],
[])).
cnf(6, lemma, equal_sets(X4,X3) | ~subset(X4,X3) | ~subset(X3,X4), infere
nce(input, [], [])).
cnf(7, lemma, equal_sets(b,bb), inference(input, [], [])).
cnf(8, lemma, member(element_of_b,b), inference(input, [], [])).
cnf(9, lemma, ~member(element_of_b,bb), inference(input, [], [])).
```

In [8]:

```
# terminated - contradiction inferred
# truncated - more than `max_clauses` (active, passive, and redundant)
terminated, truncated = False, False
step_count = 0
# `Age` tactic, FIFO
action = 0
while not terminated and not truncated:
    # don't try select redundant clauses
    if observation["action_mask"][action] == 1:
        # observation["real_obs"] only grows from step to step
        # observation["action_mask"] reflects clauses being activated
        # or made redundant (subsumed etc)
        observation, reward, terminated, truncated, info = (
            vampire.step(action)
        )
        step_count += 1
    action += 1
```

In [9]:

```
print(f"{reward=}, {terminated=}, {truncated=}")
print(step_count)
print("Action mask:", observation["action_mask"])
print(vampire.render())
```

```
reward=1.0, terminated=True, truncated=False
16
Action mask: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0
0]
cnf(1, lemma, member(X0,X2) | ~subset(X1,X2) | ~member(X0,X1), inference
(input, [], [])).
cnf(2, lemma, member(member_of_1_not_of_2(X1,X2),X1) | subset(X1,X2), inf
erence(input, [], [])).
cnf(3, lemma, subset(X1,X2) | ~member(member_of_1_not_of_2(X1,X2),X2), in
ference(input, [], [])).
cnf(4, lemma, subset(X1,X2) | ~equal_sets(X1,X2), inference(input, [],
[])).
cnf(5, lemma, subset(X1,X2) | ~equal_sets(X2,X1), inference(input, [],
[])).
cnf(6, lemma, equal_sets(X4,X3) | ~subset(X4,X3) | ~subset(X3,X4), infere
nce(input, [], [])).
cnf(7, lemma, equal_sets(b,bb), inference(input, [], [])).
cnf(8, lemma, member(element_of_b,b), inference(input, [], [])).
cnf(9, lemma, ~member(element_of_b,bb), inference(input, [], [])).
cnf(10, lemma, subset(X0,X0) | subset(X0,X0), inference(resolution, [],
[3, 2])).
cnf(11, lemma, subset(X0,X0), inference(duplicate_literal_removal, [], [1
0])).
cnf(12, lemma, subset(bb,b), inference(resolution, [], [7, 5])).
cnf(13, lemma, subset(b,bb), inference(resolution, [], [7, 4])).
cnf(14, lemma, equal_sets(X0,X0) | ~subset(X0,X0), inference(resolution,
[], [11, 6])).
cnf(15, lemma, member(X1,X2) | ~member(X1,X2), inference(resolution, [],
[11, 1])).
cnf(16, lemma, equal_sets(X0,X0), inference(subsumption_resolution, [],
[14, 11])).
cnf(17, lemma, equal_sets(bb,b) | ~subset(b,bb), inference(resolution,
[], [12, 6])).
cnf(18, lemma, member(X0,b) | ~member(X0,bb), inference(resolution, [],
```

```
[12, 1])).
cnf(19, lemma, equal_sets(bb,b), inference(subsumption_resolution, [], [1
7, 13])).
cnf(20, lemma, equal_sets(b,bb) | ~subset(bb,b), inference(resolution,
[], [13, 6])).
cnf(21, lemma, member(X0,bb) | ~member(X0,b), inference(resolution, [],
[13, 1])).
cnf(22, lemma, subset(X0,X0), inference(resolution, [], [16, 5])).
cnf(23, lemma, subset(X1,X1), inference(resolution, [], [16, 4])).
cnf(24, lemma, member(member_of_1_not_of_2(bb,X0),b) | subset(bb,X0), inf
erence(resolution, [], [18, 2])).
cnf(25, lemma, subset(b,bb), inference(resolution, [], [19, 5])).
cnf(26, lemma, subset(bb,b), inference(resolution, [], [19, 4])).
cnf(27, lemma, member(member_of_1_not_of_2(b,X0),bb) | subset(b,X0), infe
rence(resolution, [], [21, 2])).
cnf(28, lemma, member(element_of_b,bb), inference(resolution, [], [21,
8])).
cnf(29, lemma, $false, inference(subsumption_resolution, [], [28, 9])).
```

In [10]:

```
# these lines are needed when running iProver guidance from Jupyter
# not needed in a non-Jupyter script
import nest_asyncio

nest_asyncio.apply()
# guiding iProver doesn't look different
iprover = gym.make(
    "iProver-v0",
    render_mode="ansi",
    max_clauses=30
)
```

In [11]:

```python
# we set the task in absolutely the same manner as for Vampire
iprover.unwrapped.set_task(tptp_problem_path)
observation, info = iprover.reset()
print("Action mask:", observation["action_mask"])
print(iprover.render())
```

```
Action mask: [1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0]
cnf(c_50, lemma, member(element_of_b,b), inference(input, [], [])).
cnf(c_49, lemma, equal_sets(b,bb), inference(input, [], [])).
cnf(c_51, lemma, ~member(element_of_b,bb), inference(input, [], [])).
cnf(c_56, lemma, ~equal_sets(X0,X1)|subset(X1,X0), inference(input, [],
[])).
cnf(c_55, lemma, ~equal_sets(X0,X1)|subset(X0,X1), inference(input, [],
[])).
cnf(c_53, lemma, member(member_of_1_not_of_2(X0,X1),X0)|subset(X0,X1), in
ference(input, [], [])).
cnf(c_54, lemma, ~member(member_of_1_not_of_2(X0,X1),X1)|subset(X0,X1), i
nference(input, [], [])).
cnf(c_57, lemma, ~subset(X0,X1)|~subset(X1,X0)|equal_sets(X1,X0), inferen
ce(input, [], [])).
cnf(c_52, lemma, ~member(X0,X1)|~subset(X1,X2)|member(X0,X2), inference(i
nput, [], [])).
```

In [12]:

```python
terminated, truncated = False, False
step_count = 0
while not terminated and not truncated:
    # apply random available actions in `gymnasium` idiomatic way
    action = iprover.action_space.sample(
        mask=observation["action_mask"]
    )
    observation, reward, terminated, truncated, info = (
        iprover.step(action)
    )
    step_count += 1
```

In [13]:

```
print(f"{reward=}, {terminated=}, {truncated=}")
print(f"{step_count=}")
print("Action mask:", observation["action_mask"])
print(iprover.render())
```

```
reward=1.0, terminated=True, truncated=False
step_count=13
Action mask: [0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0]
cnf(c_50, lemma, member(element_of_b,b), inference(input, [], [])).
cnf(c_49, lemma, equal_sets(b,bb), inference(input, [], [])).
cnf(c_51, lemma, ~member(element_of_b,bb), inference(input, [], [])).
cnf(c_56, lemma, ~equal_sets(X0,X1)|subset(X1,X0), inference(input, [],
[])).
cnf(c_55, lemma, ~equal_sets(X0,X1)|subset(X0,X1), inference(input, [],
[])).
cnf(c_53, lemma, member(member_of_1_not_of_2(X0,X1),X0)|subset(X0,X1), in
ference(input, [], [])).
cnf(c_54, lemma, ~member(member_of_1_not_of_2(X0,X1),X1)|subset(X0,X1), i
nference(input, [], [])).
cnf(c_57, lemma, ~subset(X0,X1)|~subset(X1,X0)|equal_sets(X1,X0), inferen
ce(input, [], [])).
cnf(c_52, lemma, ~member(X0,X1)|~subset(X1,X2)|member(X0,X2), inference(i
nput, [], [])).
cnf(c_72, lemma, subset(X0,X0), inference(superposition, [], [c_53, c_5
4])).
cnf(c_74, lemma, subset(bb,b), inference(superposition, [], [c_49, c_5
6])).
cnf(c_81, lemma, ~subset(b,X0)|member(element_of_b,X0), inference(superpo
sition, [], [c_50, c_52])).
cnf(c_82, lemma, ~subset(X0,X1)|member(member_of_1_not_of_2(X0,X2),X1)|su
bset(X0,X2), inference(superposition, [], [c_53, c_52])).
cnf(c_95, lemma, equal_sets(X0,X0), inference(forward_subsumption_resolut
ion, [], [c_94, c_72])).
cnf(c_101, lemma, subset(b,bb), inference(superposition, [], [c_49, c_5
5])).
cnf(c_104, lemma, equal_sets(bb,b), inference(forward_subsumption_resolut
ion, [], [c_103, c_74])).
cnf(c_110, lemma, member(element_of_b,bb), inference(superposition, [],
```

```
[c_101, c_81])).
cnf(c_111, lemma, $false, inference(forward_subsumption_resolution, [],
[c_110, c_51])).
```

How to train an agent for such an environment

- RL algorithms usually expect observations to be tensors
- we can extract tensors from clauses using feature engineering
- `gym-saturation` includes observation and action wrappers
- one of them transforms a prover into a multi-armed bandit
- for this part `pip install ray[rllib] torch`

In [14]:

```python
from gym_saturation.wrappers import AgeWeightBandit

iprover_bandit = AgeWeightBandit(iprover)
# we have only two action: select the oldest or the shortest clause
iprover_bandit.action_space
```

Out[14]:

```
Discrete(2)
```

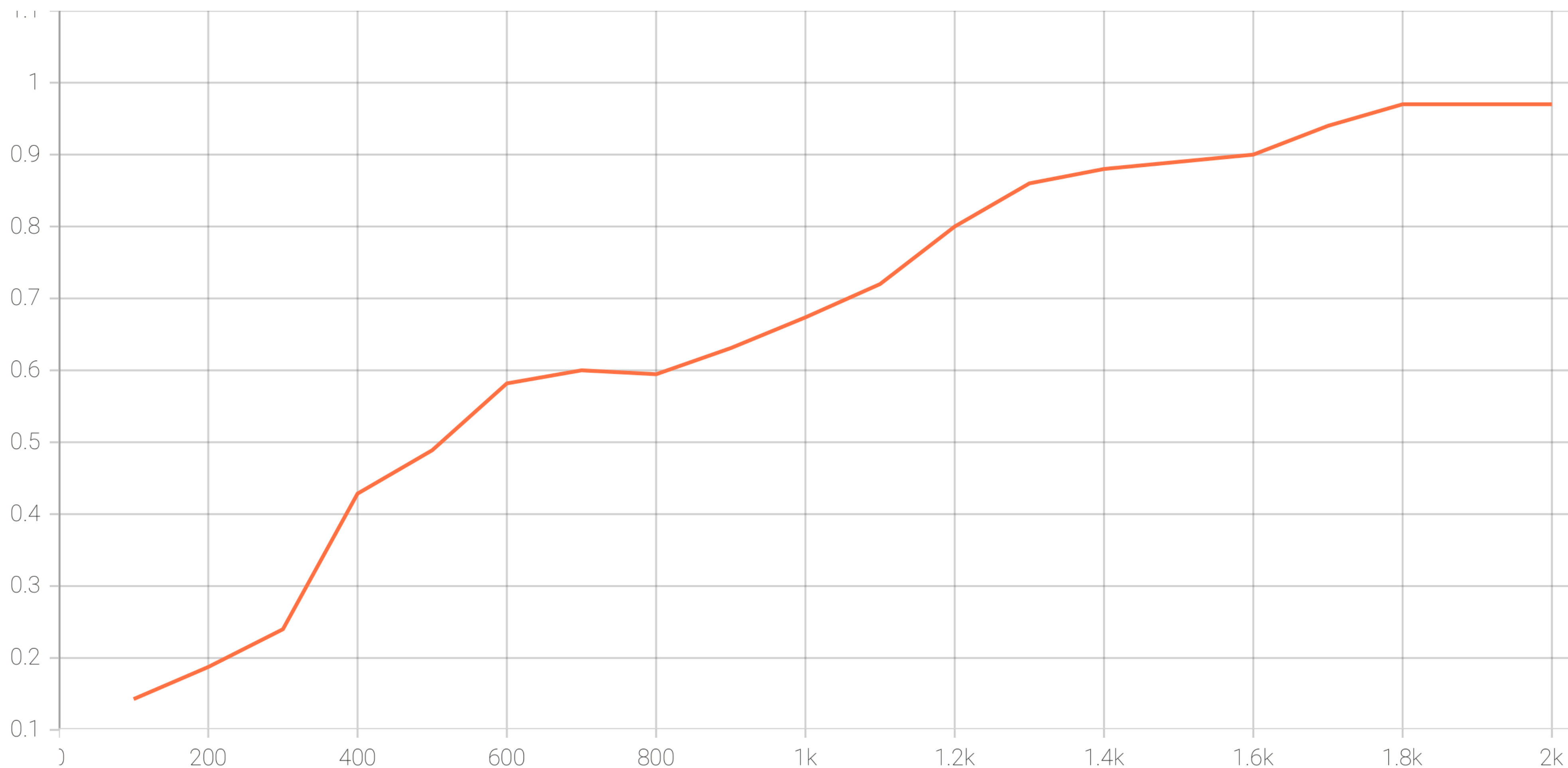
In [15]:

```python
# we will add another wrapper and register environment
# to make it compatible with Ray RLlib
from gym_saturation.wrappers import ConstantParametricActionsWrapper
from ray import tune

def env_creator(env_config):
    env = ConstantParametricActionsWrapper(
        AgeWeightBandit(gym.make("iProver-v0", max_clauses=15)),
        avail_actions_key="item",
    )
    env.set_task(tptp_problem_path)
    return env

tune.register_env("iProverBandit", env_creator)
```

In [ ]:

```
# now we can apply a Ray implementation of Thompson sampling algorithm
from ray.rllib.algorithms.bandit import BanditLinTSConfig

algo = BanditLinTSConfig().environment("iProverBandit").build()
for _ in range(20):
    algo.train()
```

Typical training chart



Good news

- the training API is prover-independent
- the state representation is prover-independent
- manually programmed features
- or code embeddings (recently tested with `huggingface`)

Bad news and possible future research

- each clause needs to be represented
- deep learning embeddings have latency in milliseconds
- Vampire can generate over million clauses in a minute
- only too simple problems solvable
- non-given-clause saturation?

Thank you for your attention!