BBRL foundations

Olivier Sigaud

Sorbonne Université
http://www.isir.upmc.fr/personnel/sigaud

Outline

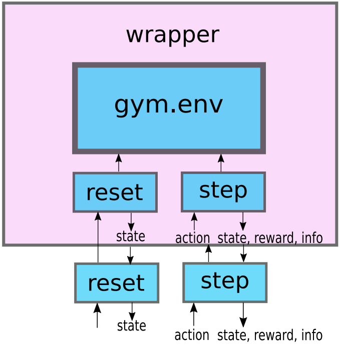
- Part 1: a standard RL model: Stable Baselines 3 (SB3)
- Limitations of the SB3 model
- Part 2: the BBRL model (inherited from SaLiNa)
- Overview of the main choices

The gym interaction loop

```
# Retrieve first observation
obs = env.reset()
done = False
total_reward = 0

while not done:
  # The agent predicts the action to take given the observation
  action, _ = agent.predict(obs, deterministic)
  # Check that predict is properly used: we use discrete actions,
  # therefore `action` should be an int here
  assert env.action_space.contains(action)

  # The environment performs a step and produces the next state, the reward
  # and whether the episode is over. The info return is a placeholder for
  # any supplementary information that one may need.
  obs, reward, done, info = env.step(action)

  # The total reward over the episode is the sum of rewards at each step
  # no discount here, discount is used in the reinforcement learning process
  total_reward += reward
return total_reward
```

- The gym interaction loop is central to evo and RL libraries
- It can be deep inside these libraries, we don't want users to add code into this core
- Two options:
  - From the environment side: wrappers
  - From the outside: callbacks
- Video presenting these SB3 aspects: https://www.youtube.com/watch?v=I8bskJuI9qU (in french)
- And the corresponding colab: https://colab.research.google.com/drive/1sBZLs-GaM8Xx7MsF6sUH7LIj6GwCq5VW?usp=sharing

Gym env wrappers



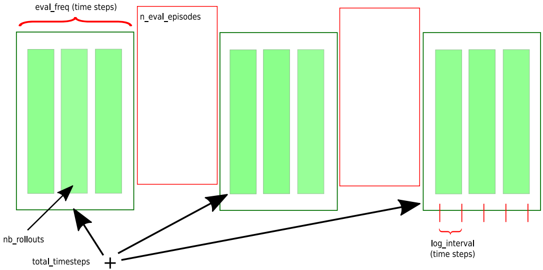
- Similar to the **Decorator** pattern
- Makes it possible to do additional (hidden) things when interacting with the environment (e.g. RewardScalingWrapper)
- Or to modify the interactions with the environment
- Main interest: the main loop is unaffected

Callbacks

```
    # Retrieve first observation
    obs = env.reset()
    done = False
    total_reward = 0

    while not done:
      # The agent predicts the action to take given the observation
      action, _ = agent.predict(obs, deterministic)
      # Check that predict is properly used: we use discrete actions,
      # therefore `action` should be an int here
      assert env.action_space.contains(action)

      # The environment performs a step and produces the next state, the reward
      # and whether the episode is over
      obs, reward, done, info = env.step(action)
      callback._on_step()

      # The total reward over the episode is the sum of rewards at each step
      total_reward += reward
    return total_reward


def train_with_callbacks(
    agent: BaseAlgorithm,
    env: gym.Env,
    nb_episodes: int,
    deterministic: bool = False,
    callback: BaseCallback
)-> float: #TODO check type

    # In SB3, training consists of several rollouts. Here we simplified this
    callback._on_training_start()
    reward_buffer = np.zeros(nb_episodes)
    for i in range(nb_episodes):
      callback._on_rollout_start()
      reward_buffer[i] = evaluate_agent_with_callbacks(agent, env, deterministic, callback)
      callback._on_rollout_end()

    callback._on_training_end()
    return reward_buffer.mean()
```

- Similar to the **Visitor** pattern
- Some objects deriving from the Callback class are registered
- One callback is the CallbackList (if we need several)
- Example callback: the eval callback
- Good practice: separate evaluation from training

Data collection: separating evaluation from training

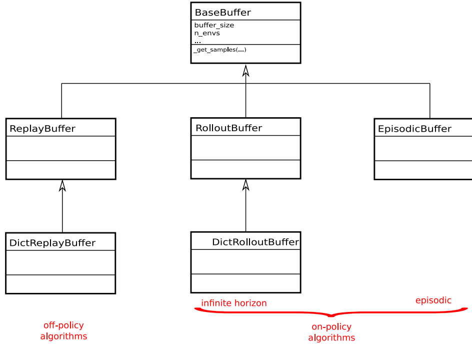


- Training curve: what do we evaluate?
- Dimension everything in time steps

Wrappers vs Callbacks

- Callbacks require additional code (wrappers don't)
- Callbacks cannot get data from the main loop (no parameters)
- Better to do things unrelated to the training loop (e.g. eval)

Buffers

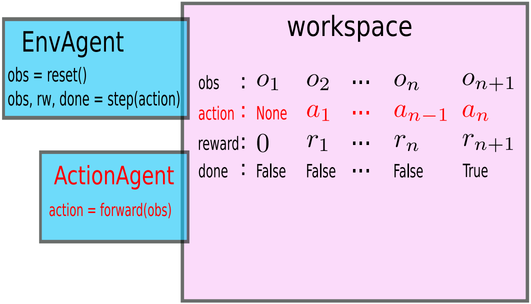


- On-policy algorithms use the RolloutBuffer
- Off-policy algorithms use the ReplayBuffer
- REINFORCE uses the EpisodicBuffer
- Need to store data from the main loop

Limitations of the SB3 model

- The main loop must be equipped with callback-related code
- Needs storing into buffers (unnecessary in evolutionary methods)
- Possible alternative: move data collection into dedicated wrappers (large refactoring)
- SB3 does not support training from multiple environments at a time
- It supports evaluating from several environments at a time (VecEnv)
- SB3 is not appropriate for teaching RL: too many things "under the hood", large code, hard to dig in
- Best for using RL as a non-expert (black box approach)

BBRL overview



- BBRL stands for "BlackBoard RL"
- It is a derivation from SaLinA, all properties come from there
- The workspace is a black board where all agents read and write temporal data
- Everything else is an agent
- Agents are pytorch `nn.Modules`: easy to move to CPU/GPU, to distribute, etc.
- Data is organized into temporal tensors which facilitate gradient processing

RL in BBRL

- By contrast to SaLinA, BBRL is limited to RL
- One agent is the Gym environment: NoAutoResetGymAgent or AutoResetGymAgent
- Other agents are RL agents
- There might be additional agents (e.g. PrintAgent for debug)
- GymAgents support training and evaluating over several environments

Why NoAutoReset and AutoReset?

- When running an agent in several environments, some environments may finish sooner than others (e.g. CartPole, when the pole falls)
- What shall we do?
- Wait until all environments end? $\rightarrow$ NoAutoResetGymAgent
- This is simpler, but a waste of time
- Restart each environment when it finishes? $\rightarrow$ AutoResetGymAgent
- Raises additional difficulties...

Gym environments: NoAutoReset

- Finished environments repeat their data until the end of all episodes

state :	s_0	s_1	$\cdots$	s_n	s_n	s_n	s_n
action :	a_0	a_1	$\cdots$	a_n	a_n	a_n	a_n
reward:	r_0	r_1	$\cdots$	r_n	r_n	r_n	r_n
done :	False	False	$\cdots$	True	True	True	True

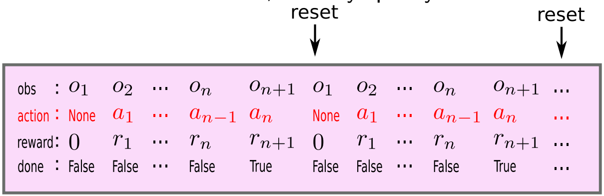
- This facilitates checking all is finished and collecting results in the end

Env_1	done:	False	False	$\cdots$	False	True	True		$\cdots$	True
	cumulated reward:	0.8	1.2	2.4	3.8	3.8	3.8			3.8
Env_2	done :	False	False	$\cdots$	True	True	True		$\cdots$	True
	cumulated reward:	1.2	3.8		5.1	5.1	5.1			5.1
Env_3	done :	False	False	$\cdots$	False	False		$\cdots$	False	True
	cumulated reward:	1.7	4.0		5.1	6.3			9.2	9.2

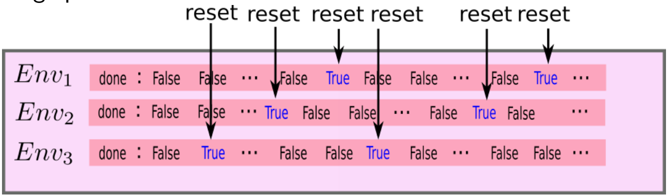
- use `stop_variable="env/done"`
- Perfect for evaluating an agent over N episodes
- The N episodes are run in parallel

Gym environments: AutoReset

- If all environments restart, we may specify blocks of arbitrary duration



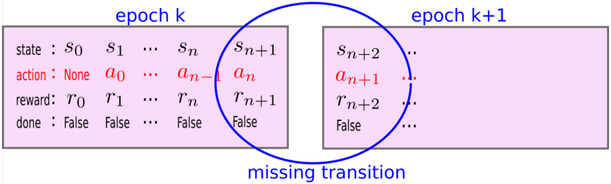
- This will make it possible to learn after each block, more often than with long episodes



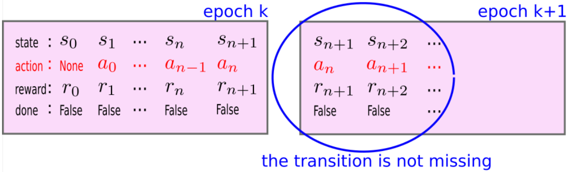
- This will raise other difficulties...

AutoReset: collecting blocks of data

- When collecting blocks of data, one should not loose the inter-block transition



- Solution: copy the last data of the previous block



Avoiding learning from inter-episode transitions

- Some transitions correspond to the last data from an episode and the first data of the next
- The agent should not learn from such transitions (it is teleported)
- SaLinA had bugs with this case
- Solution: reorganize data and remove these transitions

$$\left[\begin{array}{c} \mathrm{step}_0 \\ \mathrm{step}_1 \\ \mathrm{step}_2 \\ \hline \mathrm{step}_3 \\ \mathrm{step}_4 \end{array}\right] => \begin{bmatrix} \mathrm{step}_0, \mathrm{step}_1 \\ \mathrm{step}_1, \mathrm{step}_2 \\ \cancel{\mathrm{step}_2, \mathrm{step}_3} \\ \mathrm{step}_3, \mathrm{step}_4 \end{bmatrix} => \begin{bmatrix} \mathrm{step}_0, \mathrm{step}_1 \\ \mathrm{step}_1, \mathrm{step}_2 \\ \mathrm{step}_3, \mathrm{step}_4 \end{bmatrix}$$

- In practice, call `worskspace.get_transitions()`

get_transitions(): more details

- If `n_env` > 1, before `get_transitions()`: $[step_0^1 \; step_0^2 \ldots step_0^{n_env}]$
- After `get_transitions()`, the vector is broken into pieces:
- Each key of the returned workspace has dimensions [2, `n_transitions`, `key_dim`]
- $key[0][0], key[1][0] = (step_1, step_2) \,\#\, for\, env\, 1$
  $key[0][1], key[1][1] = (step_1, step_2) \,\#\, for\, env\, 2$
  $key[0][2], key[1][2] = (step_2, step_3) \,\#\, for\, env\, 1$
  $key[0][3], key[1][3] = (step_2, step_3) \,\#\, for\, env\, 2$
  ...

Standard or Sutton&Barto's notation?

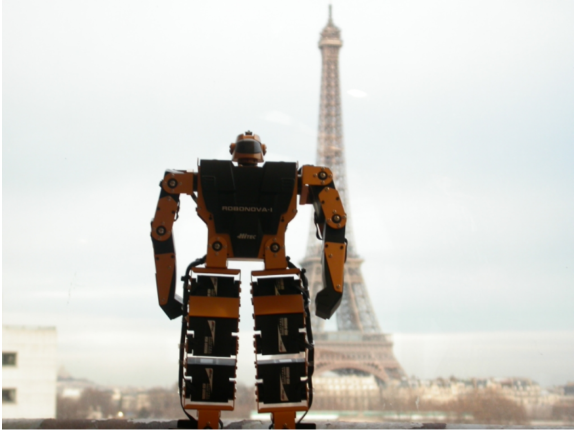
- Most often (as in my slides), one writes transitions $< s_t, a_t, r_t, s_{t+1} >$
- I.e. the reward is at the same time step than the action taken, but not the next state
- It would make more sense to write $< s_t, a_t, r_{t+1}, s_{t+1} >$ (that's what Sutton&Barto do, cf. footnote 3 page 54 of the 2018 edition)
- BBRL offers both options:

state :	s_0	s_1	$\cdots$	s_n	s_{n+1}
action :	a_0	a_1	$\cdots$	a_n	a_{n+1}
reward :	0	r_1	$\cdots$	r_n	r_{n+1}
done :	False	False	$\cdots$	False	True

state :	s_0	s_1	$\cdots$	s_n	s_{n+1}
action :	a_0	a_1	$\cdots$	a_n	a_{n+1}
reward :	r_0	r_1	$\cdots$	r_n	$\cancel{r_{n+1}}$
done :	False	False	$\cdots$	False	True

- Use `bbrl.agents.gymb` and `bbrl.utils.functionalb` instead of `bbrl.agents.gyma` and `bbrl.utils.functional` to use the standard notation
- Change the reward index accordingly...

Any question?



Send mail to: `Olivier.Sigaud@upmc.fr`

ISIR
INSTITUT
DES SYSTÈMES
INTELLIGENTS
ET DE ROBOTIQUE