aiVLE Gym - Separating Agents From Environment

GitHub repo: https://github.com/edu-ai/aivle-gym

Background

A typical OpenAI Gym agent is structured as below:

Python

```python
import gym

env = gym.make('CartPole-v0')
for i_episode in range(100):
    observation = env.reset()
    for t in range(100):
        action = decide(observation, reward)
        observation, reward, done, info = env.step(action)
env.close()
```

where `decide` refers to any form of decision-making logic (e.g. rules, neural networks, etc.)

Note that when `gym.make()` is called, you actually instantiated the simulation environment where `env.reset()` and `env.step()` will reset and take action in the simulation respectively. In this case, the simulation and agent's decision-making happens within one program. It works perfectly fine when the task is **offline** and **single-agent**. However, consider the two-agent scenario, we normally write agent code like this:

> ❗ OpenAI Gym doesn't officially support multi-agent environment. What's discussed here is a "convention" of doing multi-agent tasks under Gym API specification.

Python

```python
env = gym.make("PongDuel-v0") # Two-player Ping Pong game
for ep_i in range(100):
    done_n = [False for _ in range(env.n_agents)]
    ep_reward = 0

    obs_n = env.reset()
    env.render()

    while not all(done_n):
        # >>>>>> NEW >>>>>>
        action_0 = decide_0(obs_n[0], reward_n[0])
        action_1 = decide_1(obs_n[1], reward_n[1])
        action_n = [action_0, action_1]
        # <<<<<< NEW <<<<<<
        obs_n, reward_n, done_n, info = env.step(action_n)
        ep_reward += sum(reward_n)
        env.render()

    print('Episode #{} Reward: {}'.format(ep_i, ep_reward))
env.close()
```

Note that in a multi-agent scenario, `observation`, `reward` and `done` are all lists - each agent should receive their own observation, etc. Similarly, when you call `env.step()`, you should provide actions for every agent in this simulation. Such design is acceptable when we perform these multi-agent experiments offline because we can provide all agent implementations (i.e. `decide_0` and `decide_1`). However, aiVLE aims to allow users to submit their agent implementation with nearly zero modification. When it comes to multi-agent tasks, because you don't need to make decisions for your opponent agents when you're competing against each other on aiVLE, separating agents (i.e. decision-making logic) from the environment simulation is the only solution. After this separation, user's submission only needs to make decision for his own agent, while the environment will be responsible for collecting actions from other agents, run the simulation and respond new observation and reward to all agents. From the perspective of each agent, implementation is just like a single-agent environment. The only difference is that the environment is affected by actions taken by other agents as well.

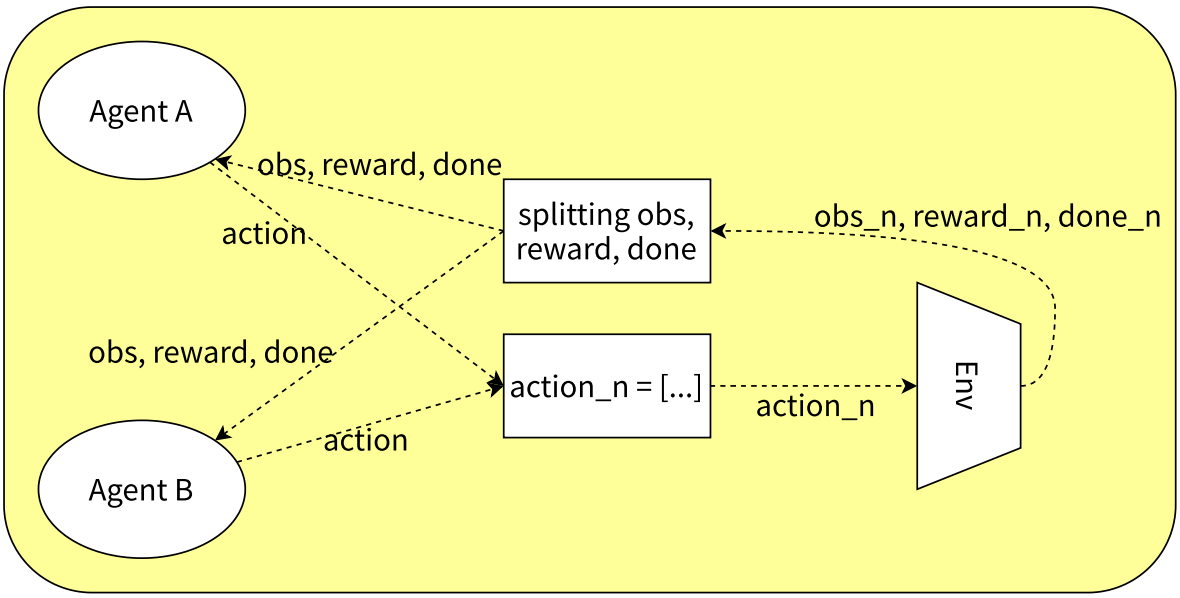
Diagram for comparison

💡 Each rounded rectangle represents one individual program.

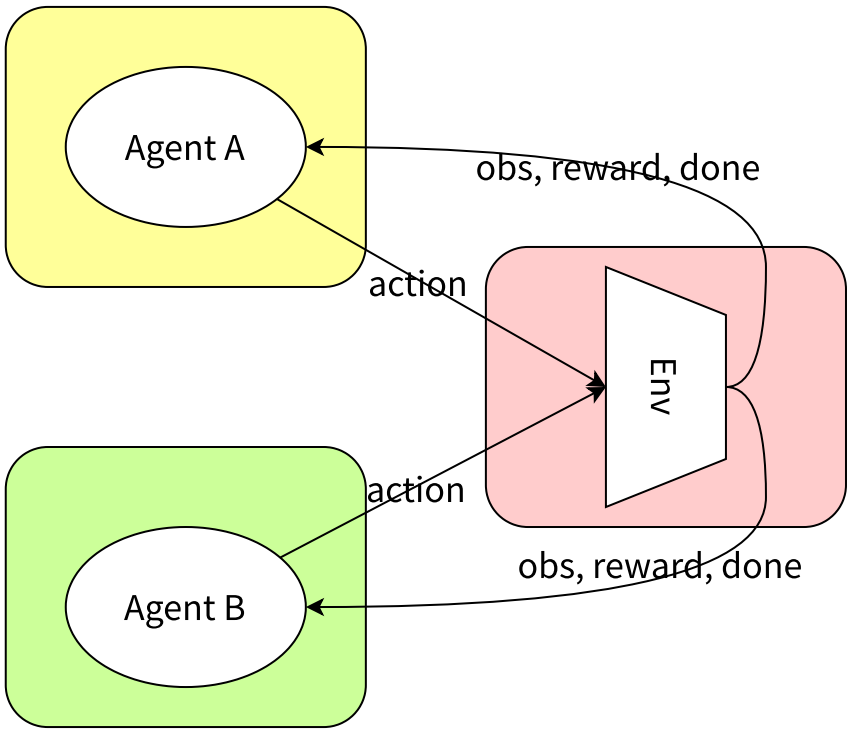
Dotted lines represents function calls (arguments/return values).

Solid lines represent inter-process/thread or even network communication.

Before separation



After separation:



Design

OpenAI Gym API

An OpenAI Gym compatible environment needs to implement these methods:

1. `step(action) -> observation, reward, done, info`
2. `reset() -> observation`
3. `render(mode)`
4. `close()`
5. `seed(seed)`

And specify these properties:

1. action_space
2. observation_space
3. reward_range: a 2-tuple of `float`
4. metadata: a `dict`
5. spec

All environments mentioned below are OpenAI Gym compatible (i.e. direct or indirect subclass of `gym.Env`).

Design goal

1. In single-agent case, on the agent side, traditional environment (simulation happens within agent process) and aiVLE environment (simulation happens outside of agent process) should be interchangeable. That is:

Python

```python
local = False
if local:
    env = gym.make("CartPole-v0")
else:
    env = CartPoleAgent()

# two env behaves exactly the same
```

On the environment side, author can reuse existing OpenAI Gym compatible environment by implementing a serializer that serializes action, observation and info to JSON compatible objects. (e.g. `numpy.ndarray` cannot be serialized automatically. The author would be responsible for marshal and unmarshal these objects) Details will be described in the following section.

2. In multi-agent case, on the agent side, the APIs behaves just like normal single-agent OpenAI Gym environment (i.e. interchangeable).

On the environment side, unlike in single-agent case where locally users can opt to use a normal OpenAI Gym environment by `gym.make()`, in addition to providing the serializer, there are a few major differences:

a. The base OpenAI Gym compatible multi-agent environment needs to follow these conventions:
   i. Has `n_agents` property that specifies number of agents in this environment
   ii. `step(action) -> observation, reward, done, info` now becomes `step(action_n) -> obs_n, reward_n, done_n, info` where `xxx_n` is a list of length `n` with observation/reward/done for each agent
   iii. Similarly, `reset()` now returns a list of observations
   iv. As for other properties like `action_space`, `observation_space` and `reward_range`, to return one object or a list is up to the author - aiVLE Gym doesn't rely on these properties for communication.
b. Need to provide an additional map of `uid_to_idx` that maps user ID to agent index used in the base environment.
c. Environment and each participating agent run separately (i.e. instead of in a for-loop of episodes, a main program collects the action from all agents into a list of `action_n` and send it to `env.step(action_n)`, in this case, each agent calls `env.step(action)` with her own action only)

📣 **Q: Why is it OK for the multi-agent environment to have so many differences?**

Ans: OpenAI Gym doesn't officially support multi-agent. We have to define our own convention to make multi-agent work under the Gym API specification.

The convention defined above is actually a pretty good balance: most users on our platform are participants, they use the `AgentEnv` where our design keeps their experience exactly the same as single-agent case. They merely needs to start the multi-agent environment using a provided script before running their own agent(s).

It's reasonable to ask for a few extra steps for environment authors (e.g. TAs), isn't it?

Messaging pattern

In aiVLE Gym implementation, we use a lightweight yet powerful messaging library called ZeroMQ (https://zeromq.org/). The details of messaging pattern supported by ZeroMQ can be found here (https://zguide.zeromq.org/docs/chapter2/#Messaging-Patterns). These messaging patterns are general concepts in computer science in a sense that they are not limited to ZeroMQ's implementation.

Whether multi-agent or not, aiVLE Gym adopts request-response pattern. The only difference is whether the socket is synchronous (REQ for client, REP for server) or asynchronous (DEALER

for client, ROUTER for server). In specific:

For single-agent: synchronous server (environment) and client (agent). This is obvious.

For multi-agent: asynchronous server (environment) + synchronous client (agent). This is because the environment needs to collect actions from all actions before replying to the agents.

> 💡 From the design goal and messaging pattern specification, we conclude that the server (`JudgeEnv`) abstract class differs between single-agent task and multi-agent task, while the client (`AgentEnv`) abstract class can be shared between two types.

Serializer (universal): `EnvSerializer`

As shown in the diagram above, arrows represent sending/receiving message. In summary, such message may include:

- action
- observation
- reward: must be `float` type according to OpenAI Gym
- done: must be `bool` type according to OpenAI Gym
- info

That leaves three types of object that can be of arbitrary type and might not be serializable (in our case, JSON-serializable). Therefore we define the `EnvSerializer` class that asks user for marshaling and unmarshaling methods for action, observation and info objects. (One typical example is observation is given as `numpy.array` by the environment, while `numpy.array` cannot be serialized by `json` package by default.)

Six abstract methods that needs to be implemented are:

1. `action_to_json`
2. `json_to_action`
3. `observation_to_json`
4. `json_to_observation`
5. `info_to_json`
6. `json_to_info`

❗ `xxx_to_json` are marshaling methods that returns a serializable Python object (**NOT** a JSON string)

Similarly, `json_to_xxx` are unmarshaling methods that do the reverse (input is **NOT** a JSON string)

Agent-side (universal): `AgentEnv`

Communication

As mentioned in the "Messaging pattern" section, agent-side client socket is synchronous.

- Socket type: `zmq.REQ`
- Target address: `tcp://localhost:{port}` where port defaults to 5555

It later use this socket to communicate with the simulation process. Note that `AgentEnv` automatically uses `EnvSerializer` to convert objects back and forth so that

- When sending the message via the socket, the message is JSON-serializable
- Arguments provided to `step`, and return value of `step` and `reset` are of the same type of the underlying environment (e.g. `numpy.array`, which is not JSON-serializable by default)

Concrete class implementation

Besides the corresponding `EnvSerializer`, user only need to provide

- action_space
- observation_space
- reward_range
- uid
- (optional) port: defaults to 5555

to implement a concrete class of `AgentEnv` abstract class. In other words, you don't need to instantiate the base environment when creating an agent-side environment (after all, the simulation happens in another program), nor do you need to provide implementation of `step`, `reset`, etc.

Usage

A concrete implementation of `AgentEnv` can be used just like any other OpenAI Gym compatible environments (given that you launched the `JudgeEnv` on corresponding port).

Environment-side (single-agent): `JudgeEnvBase` and `JudgeEnv`

Communication

As mentioned in "Messaging pattern" section, single-agent judge environment server socket is also synchronous.

- Socket type: `zmq.REP`
- Listen address: `tcp://*:{port}` where port also defaults to 5555

Concrete class implementation

- Properties to initialize
  - serializer: an `EnvSerializer` instance
  - action_space
  - observation_space
  - reward_range
  - (optional) port: defaults to 5555
- Methods to implement (generally, simply reusing base environment's corresponding method is enough)
  - step
  - reset
  - render
  - close
  - seed

Usage

`env.start()` will start a `while True` loop that responds to incoming agent-side requests.

Environment-side (multi-agent): `JudgeEnvBase` and `JudgeMultiEnv`

Design considerations

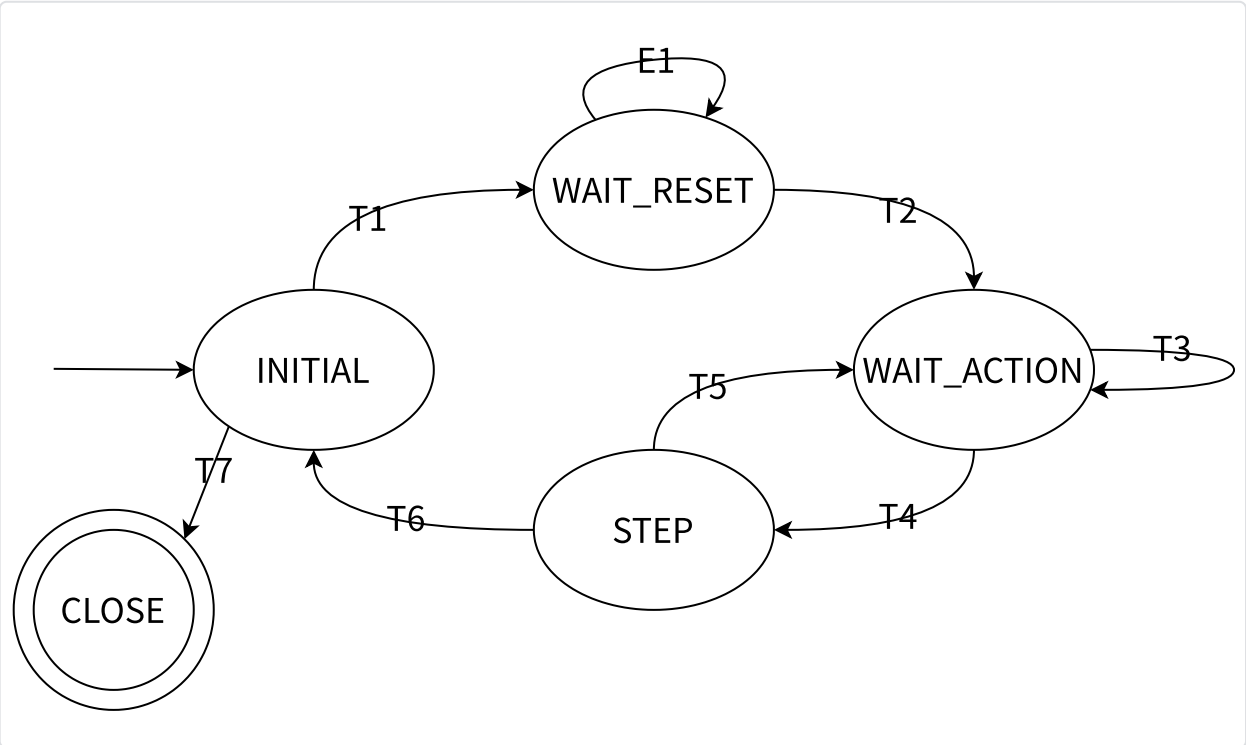
Multi-agent case is significantly more complicated because we want the agent-side to behave exactly the same as a single-agent environment (i.e. only need to provide one action, and receives one set of observation and reward for each step). Meanwhile we want the library to do most of the heavy-lifting and leave as little modification work as possible to the environment author. There are two primary challenges:

1. Judge-side should receive and respond to requests asynchronously - it needs to wait for all agents' actions before stepping forward in the environment, then decide what observations/rewards/done to respond to each of the agents.

2. Certain operations (e.g. reset) must be performed once and only once for each episode, but since each agent will initialize the episode on their own, judge-side will unavoidably receive multiple requests.

To summarize, the judge-side environment needs to implement a "barrier" synchronization mechanism that not only realizes synchronous rendezvous of agent requests, but also performs additional tasks upon the "first-comer" and "last-leaver".

Therefore, we propose the following deterministic finite automaton (DFA):



- States: INITIAL, WAIT_RESET, WAIT_ACTION, STEP, CLOSE
- Initial state q_0: INITIAL
- Accept (terminal) states F: CLOSE
- Input symbols: method (e.g. reset/step/close) and other conditions

To make this DFA a mathematically rigorous one, the domain of transition function needs to be the Cartesian product of input symbols and states. For the sake of simplicity, we omitted many self-transitioning paths - if transitioning condition is not satisfied, we assume there's a self-transition path. Meaningful transitions are described below:

- T1
  - condition: method is "reset"
  - artifact: reset the underlying base Gym environment, label this agent as already reset, save this agent's router ID
- E1
  - condition: method is "reset" and this sender hasn't reset before

    - artifact: label this agent as already reset, save this agent's router ID, trigger an input symbol of "E1" (This trigger is conceptually equivalent immediately checking if we can transit to the next state. Details can refer to the implementation.)
- T2
    - condition: input symbol of "E1", all agents are labelled as have reset
    - artifact: send initial observation to all agents, clear reset labels, clear router ID mappings
- T3
    - condition: method is "step"
    - artifact: label this agent as already stepped, save this agent's router ID, save this agent's action, trigger an input symbol of "T3"
- T4
    - condition: input symbol of "T3", all agents are labelled as have stepped
    - artifact: step forward in the base environment, send observation/reward/done/info to all agents, trigger an input symbol of "T4"
- T5
    - condition: input symbol is "T4", some of the agents still have ongoing episode
    - artifact: clear "have stepped" labels on all agents, clear router ID mappings
- T6
    - condition: input symbol is "T4", none of the agents still have ongoing episode
    - artifact: same as T5
- T7 (not implemented as of v0.1)
    - condition: method is "close"
    - artifact: close the underlying environment

By implementing this DFA carefully, the multi-agent judge environment abstract base class is capable of handling any order of incoming agent requests. Most importantly, agent-side can expect responses synchronously therefore keep all their expectations about a normal single-agent Gym environment. Note that these intricate details are not of users' (both the agent author and environment author) concern. The environment author only needs to provide implementations for the abstract methods and our library will handle the rest.

Communication

As mentioned in "Messaging pattern" section, multi-agent judge environment server socket is asynchronous.

- Socket type: `zmq.ROUTER`
- Listen address: `tcp://*:{port}` where port also defaults to 5555

Concrete class implementation

- Properties to initialize
  - serializer: an `EnvSerializer` instance
  - action_space
  - observation_space
  - reward_range
  - n_agents: number of agents
  - uid_to_idx: a map from user ID to agent index (0-based)
  - (optional) port: defaults to 5555
- Methods to implement (generally, simply reusing base environment's corresponding method is enough)
  - step
  - reset
  - render
  - seed
  - close

> ❗ `render`, `seed` and `close` remain unimplemented as of version 0.1

Usage

`env.start()` will start a `while True` loop that responds to incoming agent-side requests.