aiVLE Grader - Evaluating Agents Using Test Suites

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

❗ **Disclaimer**: aivle-grader rewrites the existing APIs defined in aivle-runner-kit (with lots of modifications and improvements). Details about can be found in "Runner-kit" section of Deep dive into existing aiVLE system

Design

Objectives

The objective of this design is to provide a modular framework such that when writing a grader for a new environment:

1. most of the time, using framework's built-in components is sufficient
2. if a custom component is required, each component is self-sufficient and straightforward without complicated inter-dependencies

For the first point, we provide `TestCase` components for both Gym and aiVLE Gym environments in both single-agent and multi-agent tasks. We also provide several `Evaluator` components that cover the most basic evaluation metrics for an agent. Most importantly, all built-in components are well-documented with runnable complete example code. The source code itself can help user learn how to extend their own components.

For the second point, we carefully design our abstractions so that each component's purpose is clear. We only expose parts that are necessary for modification and extension as abstract methods while handling the rest as either concrete implementation in the abstract base class (e.g. `evaluate` in `TestCase`) or entirely concrete class (e.g. `TestSuite`).

Overview

There are 3 components of user's concern:

- agent: replaced by student's submission
- evaluator: records the history of simulation, give scores from the history
- test cases: runs the underlying environment (Gym compatible is sufficient. I also provide examples on how to adapt aiVLE Gym environments. The process is very straightforward as

aiVLE Gym is also Gym compatible.) with runtime and episode count limit, attaches evaluator to the simulation to produce evaluation results.

A typical example of grader program looks like this:

```python
env = gym.make("...")  # any OpenAI Gym compatible environment is acceptable
evaluator = RewardEvaluator()
test_cases = [ReinforcementLearningTestCase(...), ...]
test_suite = TestSuite(suite_id="...", cases=test_cases)
eval_result = test_suite.run(create_agent)
```

Since every abstraction we make should be justified, here's the design considerations for each of them. In particular, I try to answer these two questions:

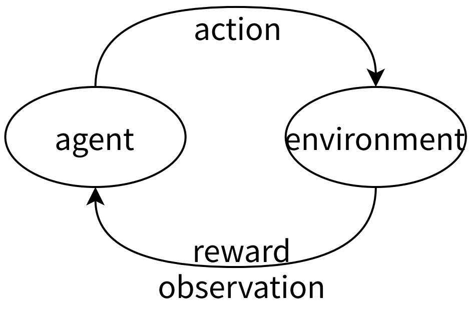
- How can we use this abstraction to improve extensibility and re-usability?
- What happens if this abstraction is removed?

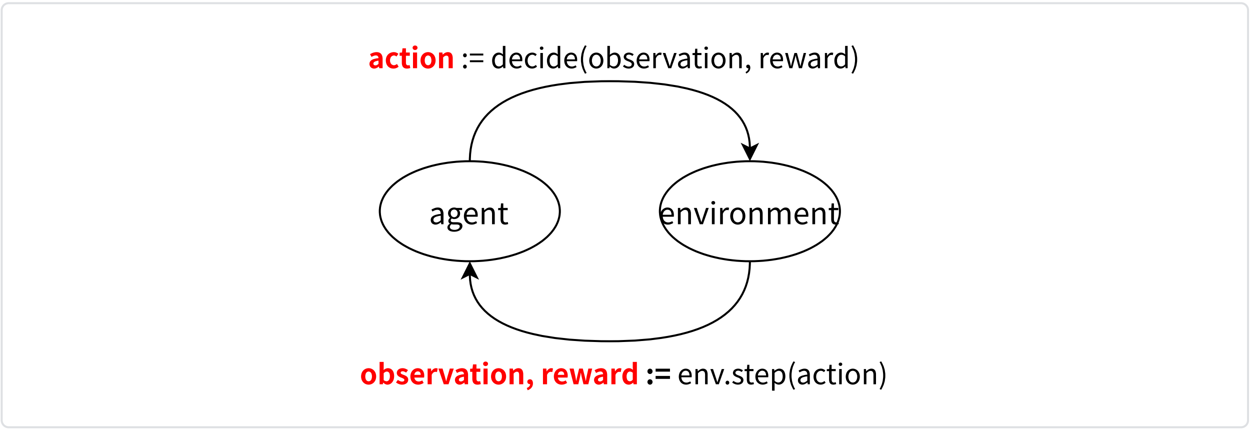
Agent

`Agent` only contains two methods: "reset" to reset internal states, "step" to return an action from provided observation. This is mostly for security reasons: the only student submission we reference is the Agent object. We deliberately hide the access to actual simulation environment from Agent object so that they don't have a chance of unintentionally or maliciously changing the random seed or closing/resetting the environment.

Evaluator

Our framework assumes Gym environment, therefore the following loop is applicable to all candidate environments:




action := decide(observation, reward)
agent
environment
observation, reward := env.step(action)

This is important because, to evaluate the performance of an agent, all information that the evaluator needs to give a verdict is observation and reward - which conveniently is always returned by the `env.step(action)` method. Therefore we can separate the evaluation logic out of the simulation loop by:

- reset the evaluator before starting an episode so that the evaluator knows when to divide its records by episode
- step the evaluator using all information received (i.e. `full_state dict`) after every step

This design makes evaluators highly reusable. For example, many tasks are evaluated by the average reward received across all episodes. If your test case happens to adopt this evaluation metric, simply using the built-in `RewardEvaluator` and passing a "reward" field in the `full_state` argument would be sufficient. On the other hand, without such abstraction, test suite authors will find themselves writing the same logic (of saving metrics data for evaluation) over and over again. This hurts the readability of `TestCase` implementation as well.

Test Cases

`TestCase` abstraction exists for two reasons:

- To provide a template: for user to implement a test case, he always needs to provide agent(s), Gym environment and an evaluator. `TestCase` abstract base class enforces these properties to be initialized in the constructor. This will force all concrete implementations to use the same variable names for common properties, which greatly improves readability.
- To offload certain chore away from the user: to run a simulation locally, we normally don't care about time limit and memory limit. However, aiVLE grader runs on a shared server that doesn't allow unlimited resource hogging. These restrictions can be dealt with by the framework in the abstraction class so that users can focus on writing normal simulation loops.

This design makes writing test cases for any environment very straightforward: simply copying over code that you use to test the environment and agent locally is almost everything you need to do. The only additional labor is to put reset/step/get_result from the evaluator on the right place.

Multi-agent Design

// TODO

Python

```python
judge_env = PongJudgeEnv()
judge_proc = Process(target=judge_env.start, args=())
judge_proc.start()

tc = MultiAgentTestCase
```

Documentation

`Agent` abstract class

To be considered a gradable agent, one needs to provide:

- `step(state)`: returns an action from observed state
- `reset()`: resets internal state for a new episode - `reset` is guaranteed to be called once before every episode

`Evaluator` abstract class

💡 You may consider `Evaluator` as a storage of evaluation results (most of the time, equivalent to reward) for each episode/run. At the end of evaluation (after `n_runs` episodes concluded), you can get a summary of this evaluation session using `get_result()` method.

There are 4 abstract methods that need to be implemented:

- `reset()`: called once at the beginning of each run
- `step(full_state: dict)`: called once after taking one action in the environment. You should give everything you received from `env.step` in `full_state` argument. The evaluator will decide which information to use.
- `get_result()`: returns an `EvaluationResult` object that summarizes all executed episodes

Built-in concrete class

You may refer to these concrete implementations before creating your own custom concrete subclass.

- RewardEvaluator: computes average reward across episodes
    - ensure `reward` field is provided in `full_state`
- StepCountEvaluator: computes average steps across episodes
    - no special requirements

`TestCase` abstract class

There are 6 properties that need initialization:

- case_id
- time_limit
- n_runs: number of episodes to run
- agent_init: init params passed to `__init__` method of `Agent`
- env: Gym compatible environment
- evaluator: `Evaluator` object

There is one abstract method that needs to be implemented:

- run: runs the `env` environment for `n_runs` times with `evaluator` attached to the execution.

Built-in concrete class

You may refer to these concrete implementations before creating your own custom concrete subclass.

- ReinforcementLearningTestCase