rlgridworld

Release latest

Mohan Zhang

Aug 21, 2022

CONTENTS

This is a simple yet efficient, highly customizable grid-world implementation to run reinforcement learning algorithms.

The official documentation is here https://rlgridworld.readthedocs.io/

install with

```
pip install rlgridworld
```

CHAPTER

ONE

ENVIRONMENT

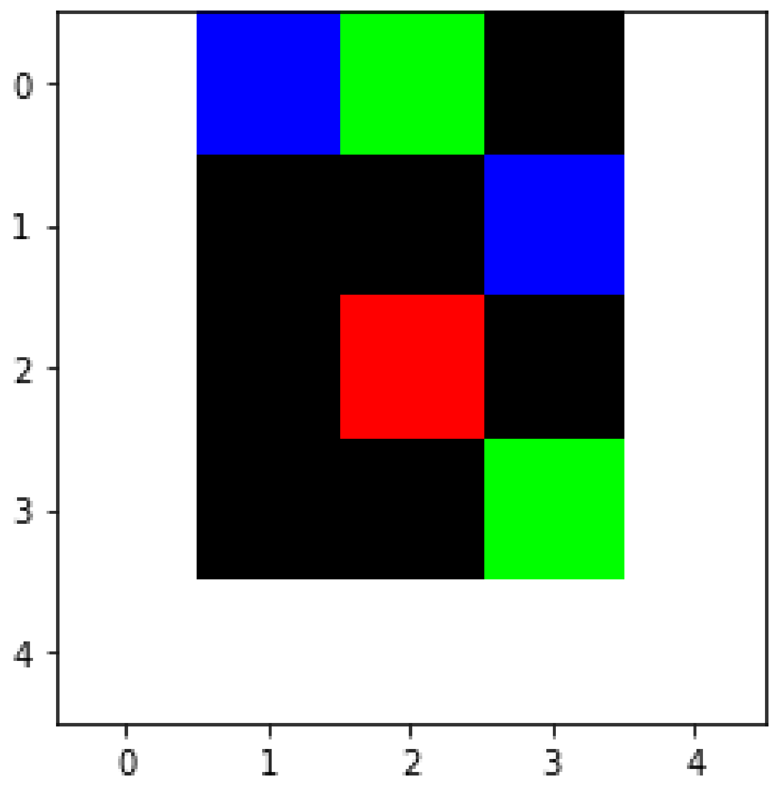
You can simply use a string like

```
W H T O W
W O O H W
W O A O W
W O O T W
W W W W W
```

to represent a grid-world, where

- A: Agent
- T: Target location
- O: Empty Ground spot (where the agent can step on and stay)
- W: Wall
- H: Hole (where the agent will fall if it steps in)

The single_rgb_array rendering of which is:



The goal of the agent is to reach one of the Target locations without falling into a hole or falling out of the edge. (More pre-configured environments can be found in EnvSettings)

CHAPTER

TWO

ACTIONS

The actions can be continuous or discrete. The agent can also move diagonally. The details can be found in the Action class in rlgridworld/gridenv.py

- Continuous: Action is a tuple of length 2, where the first element is the x-axis and the second element is the y-axis
- Discrete: Action can be chosen from [‘UP’, ‘DOWN’, ‘RIGHT’, ‘LEFT’, ‘UPRIGHT’, ‘UPLEFT’, ‘DOWNRIGHT’, ‘DOWNLEFT’]

CHAPTER THREE

REWARD

Customizable with r_fall_off, r_reach_target, r_timeout, r_continue. The details can be found in the __init__ function of class GridEnv in rlgridworld/gridenv.py

MODULES

class rlgridworld.gridenv.**Actions**

Bases: `object`

Contiguous: Action is a tuple of length 2, where the first element is the x-axis and the second element is the y-axis. UP/DOWN -> action[0] LEFT/RIGHT -> action[1] Agent will: action[0]>0.5 -> try to go UP action[0]<=-0.5 -> try to go DOWN action[1]>0.5 -> try to go RIGHT action[1]<=-0.5 -> try to go LEFT

Discrete: Action can be chosen from [‘UP’, ‘DOWN’, ‘RIGHT’, ‘LEFT’, ‘UPRIGHT’, ‘UPLEFT’, ‘DOWNRIGHT’, ‘DOWNLEFT’]

DOWN()

DOWNLEFT()

DOWNRIGHT()

LEFT()

RIGHT()

UP()

UPLEFT()

UPRIGHT()

class rlgridworld.gridenv.**GridEnv**(*load_chars_rep_fromd_dir=''*, *init_chars_representation='O O O\nO A O\nO O T'*, *max_steps=100*, *r_fall_off=-1*, *r_reach_target=1*, *r_timeout=0*, *r_continue=0*, *render_mode='human'*, *obs_mode='single_rgb_array'*, *render_width=0*, *render_height=0*)

Bases: `Env`

action_space: Space[ActType]

chars_to_world(*chars_representation*)

chars_world_to_obs(*chars_world*)

chars_world_to_rgb_array(*chars_world*)

close()

Override close in your subclass to perform any necessary cleanup.

Environments will automatically `close()` themselves when garbage collected or when the program exits.

move_to (*y*, *x*, *result*)

This is the distance=1 move action. move result should be one of: 'fall' 'fail' 'success' 'target' that is represented by the first, second, third and last element of result.

observation_space: Space[ObsType]

render (**args: Tuple[Any]*, ***kwargs: Dict[str, Any]*) → Optional[Union[RenderFrame, List[RenderFrame]]]

Compute the render frames as specified by render_mode attribute during initialization of the environment.

The set of supported modes varies per environment. (And some third-party environments may not support rendering at all.) By convention, if render_mode is:

- None (default): no render is computed.
- human: render return None. The environment is continuously rendered in the current display or terminal. Usually for human consumption.
- single_rgb_array: return a single frame representing the current state of the environment. A frame is a numpy.ndarray with shape (x, y, 3) representing RGB values for an x-by-y pixel image.
- rgb_array: return a list of frames representing the states of the environment since the last reset. Each frame is a numpy.ndarray with shape (x, y, 3), as with single_rgb_array.
- ansi: Return a list of strings (str) or StringIO.StringIO containing a terminal-style text representation for each time step. The text can include newlines and ANSI escape sequences (e.g. for colors).

Note: Rendering computations is performed internally even if you don't call render(). To avoid this, you can set render_mode = None and, if the environment supports it, call render() specifying the argument 'mode'.

Note: Make sure that your class's metadata 'render_modes' key includes the list of supported modes. It's recommended to call super() in implementations to use the functionality of this method.

reset (*seed=None*, *return_info=False*, *options=None*)

Resets the environment to an initial state and returns the initial observation.

This method can reset the environment's random number generator(s) if `seed` is an integer or if the environment has not yet initialized a random number generator. If the environment already has a random number generator and `reset()` is called with `seed=None`, the RNG should not be reset. Moreover, `reset()` should (in the typical use case) be called with an integer seed right after initialization and then never again.

Parameters

- **seed** (*optional int*) – The seed that is used to initialize the environment's PRNG. If the environment does not already have a PRNG and `seed=None` (the default option) is passed, a seed will be chosen from some source of entropy (e.g. timestamp or /dev/urandom). However, if the environment already has a PRNG and `seed=None` is passed, the PRNG will *not* be reset. If you pass an integer, the PRNG will be reset even if it already exists. Usually, you want to pass an integer *right after the environment has been initialized and then never again*. Please refer to the minimal example above to see this paradigm in action.
- **return_info** (*bool*) – If true, return additional information along with initial observation. This info should be analogous to the info returned in `step()`
- **options** (*optional dict*) – Additional information to specify how the environment is reset (optional, depending on the specific environment)

Returns

Observation of the initial state. This will be an element of `observation_space`
(typically a numpy array) and is analogous to the observation returned by `step()`.

info (optional dictionary): This will *only* be returned if `return_info=True` is passed.
It contains auxiliary information complementing `observation`. This dictionary should be analogous to the `info` returned by `step()`.

Return type
observation (object)

`step`(*action: ndarray*)

Run one timestep of the environment's dynamics.

When end of episode is reached, you are responsible for calling `reset()` to reset this environment's state. Accepts an action and returns either a tuple *(observation, reward, terminated, truncated, info)*, or a tuple (observation, reward, done, info). The latter is deprecated and will be removed in future versions.

Parameters
`action` (*`ActType`*) – an action provided by the agent

Returns

this will be an element of the environment's `observation_space`.
This may, for instance, be a numpy array containing the positions and velocities of certain objects.

reward (float): The amount of reward returned as a result of taking the action. terminated (bool): whether a *terminal state* (as defined under the MDP of the task) is reached.

In this case further step() calls could return undefined results.

truncated (bool): whether a truncation condition outside the scope of the MDP is satisfied.
Typically a timelimit, but could also be used to indicate agent physically going out of bounds. Can be used to end the episode prematurely before a *terminal state* is reached.

info (dictionary): *info* contains auxiliary diagnostic information (helpful for debugging, learning, and logging).
This might, for instance, contain: metrics that describe the agent's performance state, variables that are hidden from observations, or individual reward terms that are combined to produce the total reward. It also can contain information that distinguishes truncation and termination, however this is deprecated in favour of returning two booleans, and will be removed in a future version.

(deprecated) done (bool): A boolean value for if the episode has ended, in which case further `step()` calls will return undefined results.

A done signal may be emitted for different reasons: Maybe the task underlying the environment was solved successfully, a certain timelimit was exceeded, or the physics simulation has entered an invalid state.

Return type
observation (object)