mtenv

Release 1.0

Shagun Sodhani, Ludovic Denoyer, Pierre-Alexandre Kamienny, C

May 30, 2021

GETTING STARTED

CHAPTER

ONE

MTENV

MultiTask Environments for Reinforcement Learning.

1.1 Introduction

MTEnv is a library to interface with environments for multi-task reinforcement learning. It has two main components:

- A core API/interface that extends the gym interface and adds first-class support for multi-task RL.
- A collection of environments that implement the API.

Together, these two components should provide a standard interface for multi-task RL environments and make it easier to reuse components and tools across environments.

You can read more about the difference between `MTEnv` and single-task environments here.

1.1.1 List of publications & submissions using MTEnv (please create a pull request to add the missing entries):

- Learning Adaptive Exploration Strategies in Dynamic Environments Through Informed Policy Regularization
- Learning Robust State Abstractions for Hidden-Parameter Block MDPs
- *Multi-Task Reinforcement Learning with Context-based Representations] <https://arxiv.org/abs/2102.06177>* –

1.1.2 License

- MTEnv uses MIT License.
- Terms of Use
- Privacy Policy

1.1.3 Citing MTEnv

If you use MTEnv in your research, please use the following BibTeX entry:

```
@Misc{Sodhani2021MTEnv,
  author =       {Shagun Sodhani and Ludovic Denoyer and Pierre-Alexandre Kamienny␣
↪and Olivier Delalleau},
  title =        {MTEnv - Environment interface for mulit-task reinforcement learning}
↪,
  howpublished = {Github},
  year =         {2021},
  url =          {https://github.com/facebookresearch/mtenv}
}
```

1.2 Installation

MTEnv has two components - a core API and environments that implement the API.

The **Core API** can be installed via `pip install mtenv` or `pip install git+https://github.com/facebookresearch/mtenv.git@main#egg=mtenv`

The **list of environments**, that implement the API, is available here. Any of these environments can be installed via `pip install git+https://github.com/facebookresearch/mtenv.git@main#egg="mtenv[env_name]"`. For example, the `MetaWorld` environment can be installed via `pip install git+https://github.com/facebookresearch/mtenv.git@main#egg="mtenv[metaworld]"`.

All the environments can be installed at once using `pip install git+https://github.com/facebookresearch/mtenv.git@main#egg="mtenv[all]"`. However, note that some environments may have incompatible dependencies.

MTEnv can also be installed from the source by first cloning the repo (`git clone git@github.com:facebookresearch/mtenv.git`), *cding* into the directory `cd mtenv`, and then using the pip commands as described above. For example, `pip install mtenv` to install the core API, and `pip install "mtenv[env_name]"` to install a particular environment.

1.3 Usage

MTEnv provides an interface very similar to the standard gym environments. One key difference between multitask environments (that implement the MTEnv interface and single tasks environments is in terms of observation that they return.

1.3.1 MultiTask Observation

The multitask environments returns a dictionary as the observation. This dictionary has two keys: (i) *env_obs* which maps to the observation from the environment (i.e. the observation that a single task environments return) and (ii) *task_obs* which maps to the task-specific information from the environment. In the simplest case, *task_obs* can be an integer denoting the task index. In other cases, *task_obs* can provide richer information.

```
from mtenv import make
env = make("MT-MetaWorld-MT10-v0")
obs = env.reset()
```

(continues on next page)

(continued from previous page)

```
print(obs)
# {'env_obs': array([-0.03265039,  0.51487777,  0.2368754 , -0.06968209,  0.6235982 ,
#    0.01492813,  0.        ,  0.        ,  0.        ,  0.03933976,
#    0.89743189,  0.01492813]), 'task_obs': 1}
action = env.action_space.sample()
print(action)
# array([-0.76422   , -0.15384133,  0.74575615, -0.11724994], dtype=float32)
obs, reward, done, info = env.step(action)
print(obs)
# {'env_obs': array([-0.02583682,  0.54065546,  0.22773503, -0.06968209,  0.6235982 ,
#    0.01494118,  0.        ,  0.        ,  0.        ,  0.03933976,
#    0.89743189,  0.01492813]), 'task_obs': 1}
```

1.4 Documentation

https://mtenv.readthedocs.io

1.5 Contributing to MTEnv

There are several ways to contribute to MTEnv.

1. Use MTEnv in your research.
2. Contribute a new environment. We support many environments via MTEnv and are looking forward to adding more environments. Contributors will be added as authors of the library. You can learn more about the workflow of adding an environment here.
3. Check out the good-first-issues on GitHub and contribute to fixing those issues.
4. Check out additional details here.

1.6 Community

Ask questions in the chat or github issues:

- Chat
- Issues

1.7 Glossary

1.7.1 Task State

Task State contains all the information that the environment needs to switch to any other task.

CHAPTER

TWO

SUPPORTED ENVIRONMENTS

The following environments are supported:

2.1 Control

Installation

```
pip install git+https://github.com/facebookresearch/mtenv.git@main#egg="mtenv[control]
↪"
```

2.2 HiPBMDP

[ZSKP20] create a family of MDPs using the existing environment-task pairs from DeepMind Control Suite [TTM+20] and change one environment parameter to sample different MDPs. For more details, refer [ZSKP20].

Installation

```
pip install git+https://github.com/facebookresearch/mtenv.git@main#egg="mtenv[hipbmdp]
↪"
```

Usage

```
from mtenv import make
env = make("MT-HiPBMDP-Finger-Spin-vary-size-v0")
env.reset()
```

2.3 MetaWorld

[YQH+20] proposed an open-source simulated benchmark for meta-reinforcement learning and multi-task learning consisting of 50 distinct robotic manipulation tasks. For more details, refer [YQH+20]. MTEnv provides a wrapper for the multi-task learning environments.

Installation

```
pip install git+https://github.com/facebookresearch/mtenv.git@main#egg=
↪"mtenv[metaworld]"
```

Usage

```
from mtenv import make
env = make("MT-MetaWorld-MT10-v0") # or MT-MetaWorld-MT50-v0 or MT-MetaWorld-MT1-v0
env.reset()
```

2.4 MPTE

Installation

```
pip install git+https://github.com/facebookresearch/mtenv.git@main#egg="mtenv[mpte]"
```

2.5 References

CHAPTER

THREE

HOW TO CREATE NEW ENVIRONMENTS

There are two workflows:

1. You have a standard gym environment, which you want to convert into a multitask environment. For example, `examples/bandit.py` implements `BanditEnv` which is a standard multi-arm bandit, without an explicit notion of task. The user has the following options:
    - Write a new subclass, say `MTBanditEnv` (which subclasses `MTEnv`) as shown in `examples/mtenv_bandit.py`.
    - Use the `EnvToMTEnv` wrapper and wrap the existing single task environment. In some cases, the wrapper may have to be extended, as is done in `examples/wrapped_bandit.py`.
2. If you do not have a single-task gym environment to start with, it is recommended that you directly extend the `MTEnv` class. Implementations in `mtenv/envs` can be seen as a reference.

If you want to contribute an environment to the repo, checkout the Contribution Guide.

CHAPTER

FOUR

MTENV

4.1 mtenv package

4.1.1 Subpackages

mtenv.envs package

Subpackages

mtenv.envs.control package

Submodules

mtenv.envs.control.acrobot module

class mtenv.envs.control.acrobot.**Acrobot**

Bases: *mtenv.envs.control.acrobot.MTAcrobot*

The original acrobot environment in the MTEnv fashion

Main class for multitask RL Environments.

This abstract class extends the OpenAI Gym environment and adds support for return the task-specific information from the environment. The observation returned from the single task environments is encoded as *env_obs* (environment observation) while the task specific observation is encoded as the *task_obs* (task observation). The observation returned by *mtenv* is a dictionary of *env_obs* and *task_obs*. Since this class extends the OpenAI gym, the *mtenv* API looks similar to the gym API.

```
import mtenv
env = mtenv.make('xxx')
env.reset()
```

Any multitask RL environment class should extend/implement this class.

Parameters

- **action_space** (*Space*) –
- **env_observation_space** (*Space*) –
- **task_observation_space** (*Space*) –

sample_task_state()
Sample a *task_state*.

task_state contains all the information that the environment needs to switch to any other task.

The subclasses, extending this class, should ensure that the task seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_task_seed_is_set()*.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

class mtenv.envs.control.acrobot.**MTAcrobot**
Bases: *mtenv.core.MTEnv*

A acrobot environment with varying characteristics The task descriptor is composed of values between -1 and +1 and mapped to acrobot physical characcteristics in the self._mu_to_vars function.

Main class for multitask RL Environments.

This abstract class extends the OpenAI Gym environment and adds support for return the task-specific information from the environment. The observation returned from the single task environments is encoded as *env_obs* (environment observation) while the task specific observation is encoded as the *task_obs* (task observation). The observation returned by *mtenv* is a dictionary of *env_obs* and *task_obs*. Since this class extends the OpenAI gym, the *mtenv* API looks similar to the gym API.

```
import mtenv
env = mtenv.make('xxx')
env.reset()
```

Any multitask RL environment class should extend/implement this class.

Parameters

- **action_space** (*Space*) –
- **env_observation_space** (*Space*) –
- **task_observation_space** (*Space*) –

MAX_VEL_1 = 15.707963267948966

MAX_VEL_2 = 34.55751918948772

action_arrow = None

actions_num = 3

book_or_nips = 'book'
use dynamics equations from the nips paper or the book

domain_fig = None

dt = 0.2

get_task_obs()
Get the current value of task observation.

Environment returns task observation everytime we call *step* or *reset*. This function is useful when the user wants to access the task observation without acting in (or resetting) the environment.

Returns

Return type TaskObsType

get_task_state()
Return all the information needed to execute the current task again.

This function is useful when we want to set the environment to a previous task.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

metadata = {'render.modes': ['human', 'rgb_array'], 'video.frames_per_second': 15}

reset()
Reset the environment to some initial state and return the observation in the new state.

The subclasses, extending this class, should ensure that the environment seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_env_seed_is_set()*.

Returns For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type ObsType

sample_task_state()
Sample a *task_state*.

task_state contains all the information that the environment needs to switch to any other task.

The subclasses, extending this class, should ensure that the task seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_task_seed_is_set()*.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

seed(*env_seed*)
Set the seed for the environment's random number generator.

Invoke *seed_task* to set the seed for the task's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the environment's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

seed_task(*task_seed*)
Set the seed for the task's random number generator.

Invoke *seed* to set the seed for the environment's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the task's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

set_task_state(*task_state*)
Reset the environment to a particular task.

task_state contains all the information that the environment needs to switch to any other task.

Parameters **task_state** (*TaskStateType*) – For more information on *task_state*, refer *Task State*.

step(*a*)

Execute the action in the environment.

Parameters **action** (*ActionType*) –

Returns Tuple of *multitask observation*, *reward*, *done*, and *info*. For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type StepReturnType

torque_noise_max = 0.0

mtenv.envs.control.acrobot.**bound**(*x*, *m*, *M=None*)

Parameters **x** – scalar

Either have m as scalar, so bound(x,m,M) which returns m <= x <= M *OR* have m as length 2 vector, bound(x,m, <IGNORED>) returns m[0] <= x <= m[1].

mtenv.envs.control.acrobot.**rk4**(*derivs*, *y0*, *t*, **args*, ***kwargs*)

Integrate 1D or ND system of ODEs using 4-th order Runge-Kutta. This is a toy implementation which may be useful if you find yourself stranded on a system w/o scipy. Otherwise use `scipy.integrate()`. *y0*

initial state vector

t sample times

derivs returns the derivative of the system and has the signature `dy = derivs(yi, ti)`

args additional arguments passed to the derivative function

kwargs additional keyword arguments passed to the derivative function

Example 1 :: ## 2D system def derivs6(x,t):

d1 = x[0] + 2*x[1] d2 = -3*x[0] + 4*x[1] return (d1, d2)

dt = 0.0005 t = arange(0.0, 2.0, dt) y0 = (1,2) yout = rk4(derivs6, y0, t)

Example 2:: ## 1D system alpha = 2 def derivs(x,t):

return -alpha*x + exp(-t)

y0 = 1 yout = rk4(derivs, y0, t)

If you have access to scipy, you should probably be using the scipy.integrate tools rather than this function.

mtenv.envs.control.acrobot.**wrap**(*x*, *m*, *M*)

Parameters

- **x** – a scalar
- **m** – minimum possible value in range
- **M** – maximum possible value in range

Wraps `x` so m <= x <= M; but unlike `bound()` which truncates, `wrap()` wraps x around the coordinate system defined by m,M.

For example, m = -180, M = 180 (degrees), x = 360 –> returns 0.

mtenv.envs.control.cartpole module

class mtenv.envs.control.cartpole.**CartPole**
Bases: *mtenv.envs.control.cartpole.MTCartPole*

The original cartpole environment in the MTEnv fashion

Main class for multitask RL Environments.

This abstract class extends the OpenAI Gym environment and adds support for return the task-specific information from the environment. The observation returned from the single task environments is encoded as *env_obs* (environment observation) while the task specific observation is encoded as the *task_obs* (task observation). The observation returned by *mtenv* is a dictionary of *env_obs* and *task_obs*. Since this class extends the OpenAI gym, the *mtenv* API looks similar to the gym API.

```
import mtenv
env = mtenv.make('xxx')
env.reset()
```

Any multitask RL environment class should extend/implement this class.

Parameters

- **action_space** (*Space*) –
- **env_observation_space** (*Space*) –
- **task_observation_space** (*Space*) –

sample_task_state()
Sample a *task_state*.

task_state contains all the information that the environment needs to switch to any other task.

The subclasses, extending this class, should ensure that the task seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_task_seed_is_set()*.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

class mtenv.envs.control.cartpole.**MTCartPole**
Bases: *mtenv.core.MTEnv*

A cartpole environment with varying physical values (see the self._mu_to_vars function)

Main class for multitask RL Environments.

This abstract class extends the OpenAI Gym environment and adds support for return the task-specific information from the environment. The observation returned from the single task environments is encoded as *env_obs* (environment observation) while the task specific observation is encoded as the *task_obs* (task observation). The observation returned by *mtenv* is a dictionary of *env_obs* and *task_obs*. Since this class extends the OpenAI gym, the *mtenv* API looks similar to the gym API.

```
import mtenv
env = mtenv.make('xxx')
env.reset()
```

Any multitask RL environment class should extend/implement this class.

Parameters

- **action_space** (*Space*) –

- **env_observation_space** (*Space*) –
- **task_observation_space** (*Space*) –

get_task_obs()
Get the current value of task observation.

Environment returns task observation everytime we call *step* or *reset*. This function is useful when the user wants to access the task observation without acting in (or resetting) the environment.

Returns

Return type TaskObsType

get_task_state()
Return all the information needed to execute the current task again.

This function is useful when we want to set the environment to a previous task.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

metadata = {'render.modes': ['human', 'rgb_array'], 'video.frames_per_second': 50}

reset(***args*)
Reset the environment to some initial state and return the observation in the new state.

The subclasses, extending this class, should ensure that the environment seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_env_seed_is_set()*.

Returns For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type ObsType

sample_task_state()
Sample a *task_state*.

task_state contains all the information that the environment needs to switch to any other task.

The subclasses, extending this class, should ensure that the task seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_task_seed_is_set()*.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

seed(*env_seed*)
Set the seed for the environment's random number generator.

Invoke *seed_task* to set the seed for the task's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the environment's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

seed_task(*task_seed*)
Set the seed for the task's random number generator.

Invoke *seed* to set the seed for the environment's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the task's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

set_task_state(*task_state*)
Reset the environment to a particular task.

task_state contains all the information that the environment needs to switch to any other task.

Parameters **task_state** (*TaskStateType*) – For more information on *task_state*, refer *Task State*.

step(*action*)
Execute the action in the environment.

Parameters **action** (*ActionType*) –

Returns Tuple of *multitask observation*, *reward*, *done*, and *info*. For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type StepReturnType

mtenv.envs.control.setup module

Module contents

mtenv.envs.hipbmdp package

Subpackages

mtenv.envs.hipbmdp.wrappers package

Submodules

mtenv.envs.hipbmdp.wrappers.dmc_wrapper module

mtenv.envs.hipbmdp.wrappers.framestack module

Wrapper to stack observations for single task environments.

class mtenv.envs.hipbmdp.wrappers.framestack.**FrameStack**(*env: gym.core.Env*, *k: int*)
Bases: gym.core.Wrapper

Wrapper to stack observations for single task environments.

Parameters

- **env** (*gym.core.Env*) – Single Task Environment
- **k** (*int*) – number of frames to stack.

reset() → numpy.ndarray
Resets the environment to an initial state and returns an initial observation.

Note that this function should not reset the environment's random number generator(s); random variables in the environment's state should be sampled independently between multiple calls to *reset()*. In other

words, each call of *reset()* should yield an environment suitable for a new episode, independent of previous episodes.

Returns the initial observation.

Return type observation (object)

step(*action: Union[str, int, float, numpy.ndarray]*) → Tuple[numpy.ndarray, float, bool, Dict[str, Any]]
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 a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the agent

Returns agent's observation of the current environment reward (float) : amount of reward returned after previous action done (bool): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

Return type observation (object)

mtenv.envs.hipbmdp.wrappers.sticky_observation module

Wrapper to enable sitcky observations for single task environments.

class mtenv.envs.hipbmdp.wrappers.sticky_observation.**StickyObservation**(*env: gym.core.Env*, *sticky_probability: float*, *last_k: int*)

Bases: gym.core.Wrapper

Env wrapper that returns a previous observation with probability *p* and the current observation with a probability *1-p*. *last_k* previous observations are stored.

Parameters

- **env** (*gym.Env*) – Single task environment.
- **sticky_probability** (*float*) – Probability *p* for returning a previous observation.
- **last_k** (*int*) – Number of previous observations to store.

Raises **ValueError** – Raise a ValueError if *sticky_probability* is not in range *[0, 1]*.

reset()
Resets the environment to an initial state and returns an initial observation.

Note that this function should not reset the environment's random number generator(s); random variables in the environment's state should be sampled independently between multiple calls to *reset()*. In other words, each call of *reset()* should yield an environment suitable for a new episode, independent of previous episodes.

Returns the initial observation.

Return type observation (object)

step(*action*)
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 a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the agent

Returns agent's observation of the current environment reward (float) : amount of reward returned after previous action done (bool): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

Return type observation (object)

Module contents

Submodules

mtenv.envs.hipbmdp.dmc_env module

mtenv.envs.hipbmdp.dmc_env.**build_dmc_env**(*domain_name: str*, *task_name: str*, *seed: int*, *xml_file_id: str*, *visualize_reward: bool*, *from_pixels: bool*, *height: int*, *width: int*, *frame_skip: int*, *frame_stack: int*, *sticky_observation_cfg: Dict[str, Any]*) → gym.core.Env

Build a single DMC environment as described in [TTM+20].

Parameters

- **domain_name** (*str*) – name of the domain.
- **task_name** (*str*) – name of the task.
- **seed** (*int*) – environment seed (for reproducibility).
- **xml_file_id** (*str*) – id of the xml file to use.
- **visualize_reward** (*bool*) – should visualize reward ?
- **from_pixels** (*bool*) – return pixel observations?
- **height** (*int*) – height of pixel frames.
- **width** (*int*) – width of pixel frames.
- **frame_skip** (*int*) – should skip frames?
- **frame_stack** (*int*) – should stack frames together?
- **sticky_observation_cfg** (*Dict[str, Any]*) – Configuration for using sticky observations. It should be a dictionary with three keys, *should_use* which specifies if the config should be used, *sticky_probability* which specifies the probability of choosing a previous task and *last_k* which specifies the number of previous frames to choose from.

Returns

Return type Env

mtenv.envs.hipbmdp.env module

mtenv.envs.hipbmdp.env.**build**(*domain_name: str*, *task_name: str*, *seed: int*, *xml_file_ids: List[str]*, *visualize_reward: bool*, *from_pixels: bool*, *height: int*, *width: int*, *frame_skip: int*, *frame_stack: int*, *sticky_observation_cfg: Dict[str, Any]*, *initial_task_state: int = 1*) → *mtenv.core.MTEnv*

Build multitask environment as described in HiPBMDP paper. See [ZSKP20] for more details.

Parameters

- **domain_name** (*str*) – name of the domain.
- **task_name** (*str*) – name of the task.
- **seed** (*int*) – environment seed (for reproducibility).
- **xml_file_ids** (*List[str]*) – ids of xml files.
- **visualize_reward** (*bool*) – should visualize reward ?
- **from_pixels** (*bool*) – return pixel observations?
- **height** (*int*) – height of pixel frames.
- **width** (*int*) – width of pixel frames.
- **frame_skip** (*int*) – should skip frames?
- **frame_stack** (*int*) – should stack frames together?
- **sticky_observation_cfg** (*Dict[str, Any]*) – Configuration for using sticky observations. It should be a dictionary with three keys, *should_use* which specifies if the config should be used, *sticky_probability* which specifies the probability of choosing a previous task and *last_k* which specifies the number of previous frames to choose from.
- **initial_task_state** (*int, optional*) – intial task/environment to select. Defaults to 1.

Returns

Return type *MTEnv*

mtenv.envs.hipbmdp.setup module

Module contents

mtenv.envs.metaworld package

Subpackages

mtenv.envs.metaworld.wrappers package

Submodules

mtenv.envs.metaworld.wrappers.normalized_env module

class mtenv.envs.metaworld.wrappers.normalized_env.**NormalizedEnvWrapper**(*env*, *scale_reward=1.0*, *normalize_obs=False*, *normalize_reward=False*, *expected_action_scale=1.0*, *flatten_obs=True*, *obs_alpha=0.001*, *reward_alpha=0.001*)

Bases: gym.core.Wrapper

An environment wrapper for normalization.

This wrapper normalizes action, and optionally observation and reward.

Parameters

- **env** (*garage.envs.GarageEnv*) – An environment instance.
- **scale_reward** (*float*) – Scale of environment reward.
- **normalize_obs** (*bool*) – If True, normalize observation.
- **normalize_reward** (*bool*) – If True, normalize reward. scale_reward is applied after normalization.
- **expected_action_scale** (*float*) – Assuming action falls in the range of [-expected_action_scale, expected_action_scale] when normalize it.
- **flatten_obs** (*bool*) – Flatten observation if True.
- **obs_alpha** (*float*) – Update rate of moving average when estimating the mean and variance of observations.
- **reward_alpha** (*float*) – Update rate of moving average when estimating the mean and variance of rewards.

reset(***kwargs*)

Reset environment.

Parameters ****kwargs** – Additional parameters for reset.

Returns

- observation (np.ndarray): The observation of the environment.
- reward (float): The reward acquired at this time step.
- **done (boolean): Whether the environment was completed at this** time step.
- infos (dict): Environment-dependent additional information.

Return type tuple

step(*action*)

Feed environment with one step of action and get result.

Parameters **action** (*np.ndarray*) – An action fed to the environment.

Returns

- observation (np.ndarray): The observation of the environment.
- reward (float): The reward acquired at this time step.
- **done (boolean): Whether the environment was completed at this** time step.
- infos (dict): Environment-dependent additional information.

Return type tuple

Module contents

Submodules

mtenv.envs.metaworld.env module

mtenv.envs.metaworld.setup module

Module contents

mtenv.envs.mpte package

Submodules

mtenv.envs.mpte.setup module

mtenv.envs.mpte.two_goal_maze_env module

Module contents

mtenv.envs.shared package

Subpackages

mtenv.envs.shared.wrappers package

Submodules

mtenv.envs.shared.wrappers.multienv module

Wrapper to (lazily) construct a multitask environment from a list of constructors (list of functions to construct the environments).

class mtenv.envs.shared.wrappers.multienv.**MultiEnvWrapper**(*funcs_to_make_envs: List[Callable[], gym.core.Env]]*, *initial_task_state: int*)

Bases: *mtenv.core.MTEnv*

Wrapper to (lazily) construct a multitask environment from a list of constructors (list of functions to construct the environments).

The wrapper enables activating/slecting any environment (from the list of environments that can be created) and that environment is treated as the current task. The environments are created lazily.

Note that this wrapper is experimental and may change in the future.

Parameters

- **funcs_to_make_envs** (*List[EnvBuilderType]*) – list of constructor functions to make the environments.
- **initial_task_state** (*TaskStateType*) – intial task/environment to select.

assert_env_seed_is_set() → None
The seed is set during the call to the constructor of self.env

assert_task_seed_is_set() → None
Check that seed (for the task) is set.

sample_task_state function should invoke this function before sampling a new task state (for reproducibility).

get_task_state() → int
Return all the information needed to execute the current task again.

This function is useful when we want to set the environment to a previous task.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

reset() → Dict[str, Union[numpy.ndarray, str, int, float]]
Reset the environment to some initial state and return the observation in the new state.

The subclasses, extending this class, should ensure that the environment seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_env_seed_is_set()*.

Returns For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type ObsType

reset_task_state() → None
Sample a new task_state and set the environment to that *task_state*.

For more information on *task_state*, refer *Task State*.

sample_task_state() → int
Sample a *task_state*.

task_state contains all the information that the environment needs to switch to any other task.

The subclasses, extending this class, should ensure that the task seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_task_seed_is_set()*.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

seed(*seed: Optional[int] = None*) → List[int]
Set the seed for the environment's random number generator.

Invoke *seed_task* to set the seed for the task's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the environment's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

set_task_state(*task_state: int*) → None
Reset the environment to a particular task.

task_state contains all the information that the environment needs to switch to any other task.

Parameters **task_state** (*TaskStateType*) – For more information on *task_state*, refer *Task State*.

step(*action: Union[str, int, float, numpy.ndarray]*) → Tuple[Dict[str, Union[numpy.ndarray, str, int, float]], float, bool, Dict[str, Any]]
Execute the action in the environment.

Parameters **action** (*ActionType*) –

Returns Tuple of *multitask observation*, *reward*, *done*, and *info*. For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type StepReturnType

Module contents

Module contents

mtenv.envs.tabular_mdp package

Submodules

mtenv.envs.tabular_mdp.setup module

mtenv.envs.tabular_mdp.tmdp module

class mtenv.envs.tabular_mdp.tmdp.**TMDP**(*n_states*, *n_actions*)
Bases: *mtenv.core.MTEnv*

Defines a Tabuular MDP where task_state is the reward matrix,transition matrix reward_matrix is n_states*n_actions and gies the probability of having a reward = +1 when choosing action a in state s (matrix[s,a]) transition_matrix is n_states*n_actions*n_states and gives the probability of moving to state s' when choosing action a in state s (matrix[s,a,s'])

Parameters **MTEnv** (*[type]*) – [description]

Main class for multitask RL Environments.

This abstract class extends the OpenAI Gym environment and adds support for return the task-specific information from the environment. The observation returned from the single task environments is encoded as *env_obs* (environment observation) while the task specific observation is encoded as the *task_obs* (task observation). The observation returned by *mtenv* is a dictionary of *env_obs* and *task_obs*. Since this class extends the OpenAI gym, the *mtenv* API looks similar to the gym API.

```
import mtenv
env = mtenv.make('xxx')
env.reset()
```

Any multitask RL environment class should extend/implement this class.

Parameters

- **action_space** (*Space*) –
- **env_observation_space** (*Space*) –
- **task_observation_space** (*Space*) –

get_task_obs()
Get the current value of task observation.

Environment returns task observation everytime we call *step* or *reset*. This function is useful when the user wants to access the task observation without acting in (or resetting) the environment.

Returns

Return type TaskObsType

get_task_state()
Return all the information needed to execute the current task again.

This function is useful when we want to set the environment to a previous task.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

reset()
Reset the environment to some initial state and return the observation in the new state.

The subclasses, extending this class, should ensure that the environment seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_env_seed_is_set()*.

Returns For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type ObsType

sample_task_state()
Sample a *task_state*.

task_state contains all the information that the environment needs to switch to any other task.

The subclasses, extending this class, should ensure that the task seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_task_seed_is_set()*.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

seed(*env_seed*)
Set the seed for the environment's random number generator.

Invoke *seed_task* to set the seed for the task's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the environment's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

seed_task(*task_seed*)
Set the seed for the task's random number generator.

Invoke *seed* to set the seed for the environment's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the task's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

set_task_state(*task_state*)
Reset the environment to a particular task.

task_state contains all the information that the environment needs to switch to any other task.

Parameters **task_state** (*TaskStateType*) – For more information on *task_state*, refer *Task State*.

step(*action*)
Execute the action in the environment.

Parameters **action** (*ActionType*) –

Returns Tuple of *multitask observation*, *reward*, *done*, and *info*. For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type StepReturnType

class mtenv.envs.tabular_mdp.tmdp.**UniformTMDP**(*n_states*, *n_actions*)
Bases: *mtenv.envs.tabular_mdp.tmdp.TMDP*

Main class for multitask RL Environments.

This abstract class extends the OpenAI Gym environment and adds support for return the task-specific information from the environment. The observation returned from the single task environments is encoded as *env_obs* (environment observation) while the task specific observation is encoded as the *task_obs* (task observation). The observation returned by *mtenv* is a dictionary of *env_obs* and *task_obs*. Since this class extends the OpenAI gym, the *mtenv* API looks similar to the gym API.

```
import mtenv
env = mtenv.make('xxx')
env.reset()
```

Any multitask RL environment class should extend/implement this class.

Parameters

- **action_space** (*Space*) –
- **env_observation_space** (*Space*) –
- **task_observation_space** (*Space*) –

sample_task_state()
Sample a *task_state*.

task_state contains all the information that the environment needs to switch to any other task.

The subclasses, extending this class, should ensure that the task seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_task_seed_is_set()*.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

Module contents

Submodules

mtenv.envs.registration module

class mtenv.envs.registration.**MultiEnvRegistry**

Bases: gym.envs.registration.EnvRegistry

register(*id: str*, ***kwargs: Any*) → None

class mtenv.envs.registration.**MultitaskEnvSpec**(*id: str*, *entry_point: Optional[str] = None*, *reward_threshold: Optional[int] = None*, *kwargs: Optional[Dict[str, Any]] = None*, *nondeterministic: bool = False*, *max_episode_steps: Optional[int] = None*, *test_kwargs: Optional[Dict[str, Any]] = None*)

Bases: gym.envs.registration.EnvSpec

A specification for a particular instance of the environment. Used to register the parameters for official evaluations.

Parameters

- **id** (*str*) – The official environment ID
- **entry_point** (*Optional[str]*) – The Python entrypoint of the environment class (e.g. module.name:Class)
- **reward_threshold** (*Optional[int]*) – The reward threshold before the task is considered solved
- **kwargs** (*dict*) – The kwargs to pass to the environment class
- **nondeterministic** (*bool*) – Whether this environment is non-deterministic even after seeding
- **max_episode_steps** (*Optional[int]*) – The maximum number of steps that an episode can consist of
- **test_kwargs** (*Optional[Dict[str, Any]], optional*) – Dictionary to specify parameters for automated testing. Defaults to None.

property kwargs

mtenv.envs.registration.**make**(*id: str*, ***kwargs: Any*) → gym.core.Env

mtenv.envs.registration.**register**(*id: str*, ***kwargs: Any*) → None

mtenv.envs.registration.**spec**(*id: str*) → *mtenv.envs.registration.MultitaskEnvSpec*

Module contents

mtenv.utils package

Submodules

mtenv.utils.seeding module

mtenv.utils.seeding.**np_random**(*seed: Optional[int]*) → Tuple[numpy.random.mtrand.RandomState, int]

Set the seed for numpy's random generator.

Parameters **seed** (*Optional[int]*) –

Returns Returns a tuple of random state and seed.

Return type Tuple[RandomState, int]

mtenv.utils.setup_utils module

mtenv.utils.setup_utils.**parse_dependency**(*filepath: pathlib.Path*) → List[str]

Parse python dependencies from a file.

The list of dependencies is used by *setup.py* files. Lines starting with "#" are ingored (useful for writing comments). In case the dependnecy is host using git, the url is parsed and modified to make suitable for *setup.py* files.

Parameters **filepath** (*Path*) –

Returns List of dependencies

Return type List[str]

mtenv.utils.types module

Module contents

mtenv.wrappers package

Submodules

mtenv.wrappers.env_to_mtenv module

Wrapper to convert an environment into multitask environment.

class mtenv.wrappers.env_to_mtenv.**EnvToMTEnv**(*env: gym.core.Env*, *task_observation_space: gym.spaces.space.Space*)

Bases: *mtenv.core.MTEnv*

Wrapper to convert an environment into a multitak environment.

Parameters

- **env** (*Env*) – Environment to wrap over.

- **task_observation_space** (*Space*) – Task observation space for the resulting multitask environment.

classmethod class_name() → str

close() → Any
Override close in your subclass to perform any necessary cleanup.

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

get_task_obs() → Union[str, int, float, numpy.ndarray]
Get the current value of task observation.

Environment returns task observation everytime we call *step* or *reset*. This function is useful when the user wants to access the task observation without acting in (or resetting) the environment.

Returns

Return type TaskObsType

get_task_state() → Any
Return all the information needed to execute the current task again.

This function is useful when we want to set the environment to a previous task.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

render(*mode: str = 'human'*, ***kwargs: Dict[str, Any]*) → Any
Renders the environment.

reset(***kwargs: Dict[str, Any]*) → Dict[str, Union[numpy.ndarray, str, int, float]]
Reset the environment to some initial state and return the observation in the new state.

The subclasses, extending this class, should ensure that the environment seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_env_seed_is_set()*.

Returns For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type ObsType

reset_task_state() → None
Sample a new task_state and set the environment to that *task_state*.

For more information on *task_state*, refer *Task State*.

sample_task_state() → Any
Sample a *task_state*.

task_state contains all the information that the environment needs to switch to any other task.

The subclasses, extending this class, should ensure that the task seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_task_seed_is_set()*.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

seed(*seed: Optional[int] = None*) → List[int]
Set the seed for the environment's random number generator.

Invoke *seed_task* to set the seed for the task's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the environment's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

set_task_state(*task_state: Any*) → None
Reset the environment to a particular task.

task_state contains all the information that the environment needs to switch to any other task.

Parameters **task_state** (*TaskStateType*) – For more information on *task_state*, refer *Task State*.

property spec

step(*action: Union[str, int, float, numpy.ndarray]*) → Tuple[Dict[str, Union[numpy.ndarray, str, int, float]], float, bool, Dict[str, Any]]
Execute the action in the environment.

Parameters **action** (*ActionType*) –

Returns Tuple of *multitask observation*, *reward*, *done*, and *info*. For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type StepReturnType

property unwrapped
Completely unwrap this env.

Returns The base non-wrapped gym.Env instance

Return type gym.Env

mtenv.wrappers.multitask module

Wrapper to change the behaviour of an existing multitask environment.

class mtenv.wrappers.multitask.**MultiTask**(*env:* mtenv.core.MTEnv)
Bases: *mtenv.core.MTEnv*

Wrapper to change the behaviour of an existing multitask environment

Parameters **env** (MTEnv) – Multitask environment to wrap over.

assert_env_seed_is_set() → None
Check that the env seed is set.

assert_task_seed_is_set() → None
Check that the task seed is set.

get_task_obs() → Union[str, int, float, numpy.ndarray]
Get the current value of task observation.

Environment returns task observation everytime we call *step* or *reset*. This function is useful when the user wants to access the task observation without acting in (or resetting) the environment.

Returns

Return type TaskObsType

get_task_state() → Any
Return all the information needed to execute the current task again.

This function is useful when we want to set the environment to a previous task.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

reset() → Dict[str, Union[numpy.ndarray, str, int, float]]
Reset the environment to some initial state and return the observation in the new state.

The subclasses, extending this class, should ensure that the environment seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_env_seed_is_set()*.

Returns For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type ObsType

reset_task_state() → None
Sample a new task_state and set the environment to that *task_state*.

For more information on *task_state*, refer *Task State*.

sample_task_state() → Any
Sample a *task_state*.

task_state contains all the information that the environment needs to switch to any other task.

The subclasses, extending this class, should ensure that the task seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_task_seed_is_set()*.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

seed(*seed: Optional[int] = None*) → List[int]
Set the seed for the environment's random number generator.

Invoke *seed_task* to set the seed for the task's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the environment's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

seed_task(*seed: Optional[int] = None*) → List[int]
Set the seed for the task's random number generator.

Invoke *seed* to set the seed for the environment's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the task's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

set_task_state(*task_state: Any*) → None
Reset the environment to a particular task.

task_state contains all the information that the environment needs to switch to any other task.

Parameters **task_state** (*TaskStateType*) – For more information on *task_state*, refer *Task State*.

step(*action: Union[str, int, float, numpy.ndarray]*) → Tuple[Dict[str, Union[numpy.ndarray, str, int, float]], float, bool, Dict[str, Any]]
Execute the action in the environment.

Parameters **action** (*ActionType*) –

Returns Tuple of *multitask observation*, *reward*, *done*, and *info*. For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type StepReturnType

mtenv.wrappers.ntasks module

Wrapper to fix the number of tasks in an existing multitask environment.

class mtenv.wrappers.ntasks.**NTasks**(*env:* mtenv.core.MTEnv, *n_tasks: int*)
Bases: *mtenv.wrappers.multitask.MultiTask*

Wrapper to fix the number of tasks in an existing multitask environment to *n_tasks*.

Each task is sampled in this fixed set of *n_tasks*.

Parameters

- **env** (MTEnv) – Multitask environment to wrap over.
- **n_tasks** (*int*) – Number of tasks to sample.

reset_task_state() → None
Sample a new task_state from the set of *n_tasks* tasks and set the environment to that *task_state*.

For more information on *task_state*, refer *Task State*.

sample_task_state() → Any
Sample a *task_state* from the set of *n_tasks* tasks.

task_state contains all the information that the environment needs to switch to any other task.

The subclasses, extending this class, should ensure that the task seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_task_seed_is_set()*.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

mtenv.wrappers.ntasks_id module

Wrapper to fix the number of tasks in an existing multitask environment and return the id of the task as part of the observation.

class mtenv.wrappers.ntasks_id.**NTasksId**(*env:* mtenv.core.MTEnv, *n_tasks: int*)
Bases: *mtenv.wrappers.ntasks.NTasks*

Wrapper to fix the number of tasks in an existing multitask environment to *n_tasks*.

Each task is sampled in this fixed set of *n_tasks*. The agent observes the id of the task.

Parameters

- **env** (MTEnv) – Multitask environment to wrap over.
- **n_tasks** (*int*) – Number of tasks to sample.

get_task_obs() → Any
Get the current value of task observation.

Environment returns task observation everytime we call *step* or *reset*. This function is useful when the user wants to access the task observation without acting in (or resetting) the environment.

Returns

Return type TaskObsType

get_task_state() → Any
Return all the information needed to execute the current task again.

This function is useful when we want to set the environment to a previous task.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

reset() → Dict[str, Union[numpy.ndarray, str, int, float]]
Reset the environment to some initial state and return the observation in the new state.

The subclasses, extending this class, should ensure that the environment seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_env_seed_is_set()*.

Returns For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type ObsType

sample_task_state() → Any
Sample a *task_state* from the set of *n_tasks* tasks.

task_state contains all the information that the environment needs to switch to any other task.

The subclasses, extending this class, should ensure that the task seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_task_seed_is_set()*.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

set_task_state(*task_state: Any*) → None
Reset the environment to a particular task.

task_state contains all the information that the environment needs to switch to any other task.

Parameters **task_state** (*TaskStateType*) – For more information on *task_state*, refer *Task State*.

step(*action: Union[str, int, float, numpy.ndarray]*) → Tuple[Dict[str, Union[numpy.ndarray, str, int, float]], float, bool, Dict[str, Any]]
Execute the action in the environment.

Parameters **action** (*ActionType*) –

Returns Tuple of *multitask observation*, *reward*, *done*, and *info*. For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type StepReturnType

mtenv.wrappers.sample_random_task module

Wrapper that samples a new task everytime the environment is reset.

class mtenv.wrappers.sample_random_task.**SampleRandomTask**(*env:* mtenv.core.MTEnv)
Bases: *mtenv.wrappers.multitask.MultiTask*

Wrapper that samples a new task everytime the environment is reset.

Parameters **env** (MTEnv) – Multitask environment to wrap over.

reset() → Dict[str, Union[numpy.ndarray, str, int, float]]
Reset the environment to some initial state and return the observation in the new state.

The subclasses, extending this class, should ensure that the environment seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_env_seed_is_set()*.

Returns For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type ObsType

Module contents

4.1.2 Submodules

4.1.3 mtenv.core module

Core API of MultiTask Environments for Reinforcement Learning.

class mtenv.core.**MTEnv**(*action_space: gym.spaces.space.Space*, *env_observation_space: gym.spaces.space.Space*, *task_observation_space: gym.spaces.space.Space*)
Bases: gym.core.Env, abc.ABC

Main class for multitask RL Environments.

This abstract class extends the OpenAI Gym environment and adds support for return the task-specific information from the environment. The observation returned from the single task environments is encoded as *env_obs* (environment observation) while the task specific observation is encoded as the *task_obs* (task observation). The observation returned by *mtenv* is a dictionary of *env_obs* and *task_obs*. Since this class extends the OpenAI gym, the *mtenv* API looks similar to the gym API.

```
import mtenv
env = mtenv.make('xxx')
env.reset()
```

Any multitask RL environment class should extend/implement this class.

Parameters

- **action_space** (*Space*) –
- **env_observation_space** (*Space*) –
- **task_observation_space** (*Space*) –

assert_env_seed_is_set() → None
Check that seed (for the environment) is set.

reset function should invoke this function before resetting the environment (for reproducibility).

assert_task_seed_is_set() → None
Check that seed (for the task) is set.

sample_task_state function should invoke this function before sampling a new task state (for reproducibility).

get_task_obs() → Union[str, int, float, numpy.ndarray]
Get the current value of task observation.

Environment returns task observation everytime we call *step* or *reset*. This function is useful when the user wants to access the task observation without acting in (or resetting) the environment.

Returns

Return type TaskObsType

abstract get_task_state() → Any
Return all the information needed to execute the current task again.

This function is useful when we want to set the environment to a previous task.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

abstract reset() → Dict[str, Union[numpy.ndarray, str, int, float]]
Reset the environment to some initial state and return the observation in the new state.

The subclasses, extending this class, should ensure that the environment seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_env_seed_is_set()*.

Returns For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type ObsType

reset_task_state() → None
Sample a new task_state and set the environment to that *task_state*.

For more information on *task_state*, refer *Task State*.

abstract sample_task_state() → Any
Sample a *task_state*.

task_state contains all the information that the environment needs to switch to any other task.

The subclasses, extending this class, should ensure that the task seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_task_seed_is_set()*.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

seed(*seed: Optional[int] = None*) → List[int]
Set the seed for the environment's random number generator.

Invoke *seed_task* to set the seed for the task's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the environment's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

seed_task(*seed: Optional[int] = None*) → List[int]
Set the seed for the task's random number generator.

Invoke *seed* to set the seed for the environment's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the task's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

abstract set_task_state(*task_state: Any*) → None
Reset the environment to a particular task.

task_state contains all the information that the environment needs to switch to any other task.

Parameters **task_state** (*TaskStateType*) – For more information on *task_state*, refer *Task State*.

abstract step(*action: Union[str, int, float, numpy.ndarray]*) → Tuple[Dict[str, Union[numpy.ndarray, str, int, float]], float, bool, Dict[str, Any]]
Execute the action in the environment.

Parameters **action** (*ActionType*) –

Returns Tuple of *multitask observation*, *reward*, *done*, and *info*. For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type StepReturnType

4.1.4 Module contents

class mtenv.**MTEnv**(*action_space: gym.spaces.space.Space, env_observation_space: gym.spaces.space.Space, task_observation_space: gym.spaces.space.Space*)
Bases: gym.core.Env, abc.ABC

Main class for multitask RL Environments.

This abstract class extends the OpenAI Gym environment and adds support for return the task-specific information from the environment. The observation returned from the single task environments is encoded as *env_obs* (environment observation) while the task specific observation is encoded as the *task_obs* (task observation). The observation returned by *mtenv* is a dictionary of *env_obs* and *task_obs*. Since this class extends the OpenAI gym, the *mtenv* API looks similar to the gym API.

```
import mtenv
env = mtenv.make('xxx')
env.reset()
```

Any multitask RL environment class should extend/implement this class.

Parameters

- **action_space** (*Space*) –
- **env_observation_space** (*Space*) –
- **task_observation_space** (*Space*) –

assert_env_seed_is_set() → None
Check that seed (for the environment) is set.

reset function should invoke this function before resetting the environment (for reproducibility).

assert_task_seed_is_set() → None
Check that seed (for the task) is set.

sample_task_state function should invoke this function before sampling a new task state (for reproducibility).

get_task_obs() → Union[str, int, float, numpy.ndarray]
Get the current value of task observation.

Environment returns task observation everytime we call *step* or *reset*. This function is useful when the user wants to access the task observation without acting in (or resetting) the environment.

Returns

Return type TaskObsType

abstract get_task_state() → Any
Return all the information needed to execute the current task again.

This function is useful when we want to set the environment to a previous task.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

abstract reset() → Dict[str, Union[numpy.ndarray, str, int, float]]
Reset the environment to some initial state and return the observation in the new state.

The subclasses, extending this class, should ensure that the environment seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_env_seed_is_set()*.

Returns For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type ObsType

reset_task_state() → None
Sample a new task_state and set the environment to that *task_state*.

For more information on *task_state*, refer *Task State*.

abstract sample_task_state() → Any
Sample a *task_state*.

task_state contains all the information that the environment needs to switch to any other task.

The subclasses, extending this class, should ensure that the task seed is set (by calling *seed(int)*) before invoking this method (for reproducibility). It can be done by invoking *self.assert_task_seed_is_set()*.

Returns For more information on *task_state*, refer *Task State*.

Return type TaskStateType

seed(*seed: Optional[int] = None*) → List[int]
Set the seed for the environment's random number generator.

Invoke *seed_task* to set the seed for the task's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the environment's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

seed_task (*seed: Optional[int] = None*) → List[int]
Set the seed for the task's random number generator.

Invoke *seed* to set the seed for the environment's random number generator.

Parameters **seed** (*Optional[int], optional*) – Defaults to None.

Returns Returns the list of seeds used in the task's random number generator. The first value in the list should be the seed that should be passed to this method for reproducibility.

Return type List[int]

abstract set_task_state (*task_state: Any*) → None
Reset the environment to a particular task.

task_state contains all the information that the environment needs to switch to any other task.

Parameters **task_state** (*TaskStateType*) – For more information on *task_state*, refer *Task State*.

abstract step (*action: Union[str, int, float, numpy.ndarray]*) → Tuple[Dict[str, Union[numpy.ndarray, str, int, float]], float, bool, Dict[str, Any]]
Execute the action in the environment.

Parameters **action** (*ActionType*) –

Returns Tuple of *multitask observation*, *reward*, *done*, and *info*. For more information on *multitask observation* returned by the environment, refer *MultiTask Observation*.

Return type StepReturnType

mtenv.**make** (*id: str*, ***kwargs: Any*) → gym.core.Env

CHAPTER

FIVE

COMMUNITY

Ask questions in the chat or GitHub issues.

To contribute, open a Pull Request (PR)

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[TTM+20] Yuval Tassa, Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, and Nicolas Heess. Dm_control: software and tasks for continuous control. 2020. arXiv:2006.12983.

[YQH+20] Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: a benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on Robot Learning*, 1094–1100. PMLR, 2020.

[ZSKP20] Amy Zhang, Shagun Sodhani, Khimya Khetarpal, and Joelle Pineau. Multi-task reinforcement learning as a hidden-parameter block mdp. *arXiv preprint arXiv:2007.07206*, 2020.

PYTHON MODULE INDEX

m

INDEX

K

M

N

P

R

S

T

U

W