# Mushroom Documentation

***Release 1.0.0***

**Carlo D'Eramo, Davide Tateo**

**Dec 21, 2019**

# Contents:

# CHAPTER 1

## Reinforcement Learning python library

Mushroom is a Reinforcement Learning (RL) library that aims to be a simple, yet powerful way to make **RL** and **deep RL** experiments. The idea behind Mushroom consists in offering the majority of RL algorithms providing a common interface in order to run them without excessive effort. Moreover, it is designed in such a way that new algorithms and other stuff can generally be added transparently without the need of editing other parts of the code. Mushroom makes a large use of the environments provided by OpenAI Gym library and of the regression models provided by Scikit-Learn library giving also the possibility to build and run neural networks using Tensorflow library.

With Mushroom you can:

- solve RL problems simply writing a single small script;
- add custom algorithms and other stuff transparently;
- use all RL environments offered by OpenAI Gym and build customized environments as well;
- exploit regression models offered by Scikit-Learn or build a customized one with Tensorflow;
- run experiments with CPU or GPU.

# CHAPTER 2

## Basic run example

Solve a discrete MDP in few a lines. Firstly, create a **MDP**:

```
from mushroom.environments import GridWorld

mdp = GridWorld(width=3, height=3, goal=(2, 2), start=(0, 0))
```

Then, an epsilon-greedy **policy** with:

```
from mushroom.policy import EpsGreedy
from mushroom.utils.parameters import Parameter

epsilon = Parameter(value=1.)
policy = EpsGreedy(epsilon=epsilon)
```

Eventually, the **agent** is:

```
from mushroom.algorithms.value import QLearning

learning_rate = Parameter(value=.6)
agent = QLearning(policy, mdp.info, learning_rate)
```

Learn:

```
from mushroom.core.core import Core

core = Core(agent, mdp)
core.learn(n_steps=10000, n_steps_per_fit=1)
```

Print final Q-table:

```
import numpy as np

shape = agent.approximator.shape
q = np.zeros(shape)
```

(continues on next page)

(continued from previous page)

```
for i in range(shape[0]):
    for j in range(shape[1]):
        state = np.array([i])
        action = np.array([j])
        q[i, j] = agent.approximator.predict(state, action)
print(q)
```

Results in:

```
[[  6.561   7.29    6.561   7.29 ]
 [  7.29    8.1     6.561   8.1  ]
 [  8.1     9.      7.29    8.1  ]
 [  6.561   8.1     7.29    8.1  ]
 [  7.29    9.      7.29    9.   ]
 [  8.1    10.      8.1     9.   ]
 [  7.29    8.1     8.1     9.   ]
 [  8.1     9.      8.1    10.   ]
 [  0.      0.      0.      0.   ]]
```

where the Q-values of each action of the MDP are stored for each rows representing a state of the MDP.

# CHAPTER 3

# Download and installation

Mushroom can be downloaded from the GitHub repository. Installation can be done running

```
pip3 install -e .
```

and

```
pip3 install -r requirements.txt
```

to install all its dependencies.

To compile the documentation:

```
cd mushroom/docs
make html
```

or to compile the pdf version:

```
cd mushroom/docs
make latexpdf
```

To launch mushroom test suite:

```
cd mushroom/tests
python3 -m pytest
```

## 3.1 Mushroom

List of the Mushroom modules:

### 3.1.1 Core

**class** mushroom.core.core.**Core**(*agent*, *mdp*, *callbacks=None*)
Bases: object

Implements the functions to run a generic algorithm.

**__init__**(*agent*, *mdp*, *callbacks=None*)
Constructor.

**Parameters**

- **agent** (Agent) – the agent moving according to a policy;
- **mdp** (*Environment*) – the environment in which the agent moves;
- **callbacks** (*list*) – list of callbacks to execute at the end of each learn iteration.

**learn**(*n_steps=None*, *n_episodes=None*, *n_steps_per_fit=None*, *n_episodes_per_fit=None*, *render=False*, *quiet=False*)
This function moves the agent in the environment and fits the policy using the collected samples. The agent can be moved for a given number of steps or a given number of episodes and, independently from this choice, the policy can be fitted after a given number of steps or a given number of episodes. By default, the environment is reset.

**Parameters**

- **n_steps** (*int, None*) – number of steps to move the agent;
- **n_episodes** (*int, None*) – number of episodes to move the agent;
- **n_steps_per_fit** (*int, None*) – number of steps between each fit of the policy;
- **n_episodes_per_fit** (*int, None*) – number of episodes between each fit of the policy;
- **render** (*bool, False*) – whether to render the environment or not;
- **quiet** (*bool, False*) – whether to show the progress bar or not.

**evaluate**(*initial_states=None*, *n_steps=None*, *n_episodes=None*, *render=False*, *quiet=False*)
This function moves the agent in the environment using its policy. The agent is moved for a provided number of steps, episodes, or from a set of initial states for the whole episode. By default, the environment is reset.

**Parameters**

- **initial_states** (*np.ndarray, None*) – the starting states of each episode;
- **n_steps** (*int, None*) – number of steps to move the agent;
- **n_episodes** (*int, None*) – number of episodes to move the agent;
- **render** (*bool, False*) – whether to render the environment or not;
- **quiet** (*bool, False*) – whether to show the progress bar or not.

**_step**(*render*)
Single step.

**Parameters** **render** (*bool*) – whether to render or not.

**Returns** A tuple containing the previous state, the action sampled by the agent, the reward obtained, the reached state, the absorbing flag of the reached state and the last step flag.

**reset**(*initial_states=None*)
Reset the state of the agent.

### 3.1.2 Environments

#### Environments

**class** mushroom.environments.environment.**MDPInfo**(*observation_space*, *action_space*, *gamma*, *horizon*)

Bases: object

This class is used to store the information of the environment.

**__init__**(*observation_space*, *action_space*, *gamma*, *horizon*)

Constructor.

**Parameters**

- **observation_space** ([Box, Discrete]) – the state space;
- **action_space** ([Box, Discrete]) – the action space;
- **gamma** (*float*) – the discount factor;
- **horizon** (*int*) – the horizon.

**size**

The sum of the number of discrete states and discrete actions. Only works for discrete spaces.

**Type** Returns

**shape**

The concatenation of the shape tuple of the state and action spaces.

**Type** Returns

#### Atari

**class** mushroom.environments.atari.**MaxAndSkip**(*env*, *skip*, *max_pooling=True*)

Bases: gym.core.Wrapper

**__init__**(*env*, *skip*, *max_pooling=True*)

Initialize self. See help(type(self)) for accurate signature.

**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)

**reset**(***kwargs*)

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

**Returns** the initial observation.

**Return type** observation (object)

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

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

**render**(*mode='human'*, ***kwargs*)
Renders the environment.

The set of supported modes varies per environment. (And some environments do not support rendering at all.) By convention, if mode is:

- human: render to the current display or terminal and return nothing. Usually for human consumption.
- rgb_array: Return an numpy.ndarray with shape (x, y, 3), representing RGB values for an x-by-y pixel image, suitable for turning into a video.
- ansi: Return a string (str) or StringIO.StringIO containing a terminal-style text representation. The text can include newlines and ANSI escape sequences (e.g. for colors).

---

**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.

---

**Parameters** **mode** (*str*) – the mode to render with

Example:

**class MyEnv(Env):** metadata = {'render.modes': ['human', 'rgb_array']}

**def render(self, mode='human'):**

**if mode == 'rgb_array':** return np.array(...) # return RGB frame suitable for video

**elif mode == 'human':** ... # pop up a window and render

**else:** super(MyEnv, self).render(mode=mode) # just raise an exception

**seed**(*seed=None*)
Sets the seed for this env's random number generator(s).

---

**Note:** Some environments use multiple pseudorandom number generators. We want to capture all such seeds used in order to ensure that there aren't accidental correlations between multiple generators.

---

**Returns**

**Returns the list of seeds used in this env's random** number generators. The first value in the list should be the "main" seed, or the value which a reproducer should pass to 'seed'. Often, the main seed equals the provided 'seed', but this won't be true if seed=None, for example.

**Return type** list<bigint>

**unwrapped**
Completely unwrap this env.

**Returns** The base non-wrapped gym.Env instance

**Return type** gym.Env

**class** mushroom.environments.atari.**LazyFrames**(*frames*, *history_length*)
Bases: object

From OpenAI Baseline. https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py

**__init__**(*frames*, *history_length*)
Initialize self. See help(type(self)) for accurate signature.

**class** mushroom.environments.atari.**Atari**(*name*, *width=84*, *height=84*, *ends_at_life=False*, *max_pooling=True*, *history_length=4*, *max_no_op_actions=30*)
Bases: mushroom.environments.environment.Environment

The Atari environment as presented in: "Human-level control through deep reinforcement learning". Mnih et. al.. 2015.

**__init__**(*name*, *width=84*, *height=84*, *ends_at_life=False*, *max_pooling=True*, *history_length=4*, *max_no_op_actions=30*)
Constructor.

**Parameters**

- **name** (*str*) – id name of the Atari game in Gym;
- **width** (*int, 84*) – width of the screen;
- **height** (*int, 84*) – height of the screen;
- **ends_at_life** (*bool, False*) – whether the episode ends when a life is lost or not;
- **max_pooling** (*bool, True*) – whether to do max-pooling or average-pooling of the last two frames when using NoFrameskip;
- **history_length** (*int, 4*) – number of frames to form a state;
- **max_no_op_actions** (*int, 30*) – maximum number of no-op action to execute at the beginning of an episode.

**reset**(*state=None*)
Reset the current state.

**Parameters** **state** (*np.ndarray, None*) – the state to set to the current state.

**Returns** The current state.

**step**(*action*)
Move the agent from its current state according to the action.

**Parameters** **action** (*np.ndarray*) – the action to execute.

**Returns** The state reached by the agent executing `action` in its current state, the reward obtained in the transition and a flag to signal if the next state is absorbing. Also an additional dictionary is returned (possibly empty).

**stop**()
Method used to stop an mdp. Useful when dealing with real world environments, simulators, or when using openai-gym rendering

**static _bound**(*x*, *min_value*, *max_value*)
Method used to bound state and action variables.

**Parameters**

- **x** – the variable to bound;
- **min_value** – the minimum value;

- **max_value** – the maximum value;

**Returns** The bounded variable.

**info**
An object containing the info of the environment.

**Type** Returns

**seed**(*seed*)
Set the seed of the environment.

**Parameters** **seed** (*float*) – the value of the seed.

**set_episode_end**(*ends_at_life*)
Setter.

**Parameters** **ends_at_life** (*bool*) – whether the episode ends when a life is lost or not.

### Car on hill

**class** mushroom.environments.car_on_hill.**CarOnHill**(*horizon=100*, *gamma=0.95*)
Bases: mushroom.environments.environment.Environment

The Car On Hill environment as presented in: "Tree-Based Batch Mode Reinforcement Learning". Ernst D. et al.. 2005.

**__init__**(*horizon=100*, *gamma=0.95*)
Constructor.

**reset**(*state=None*)
Reset the current state.

**Parameters** **state** (*np.ndarray, None*) – the state to set to the current state.

**Returns** The current state.

**step**(*action*)
Move the agent from its current state according to the action.

**Parameters** **action** (*np.ndarray*) – the action to execute.

**Returns** The state reached by the agent executing action in its current state, the reward obtained in the transition and a flag to signal if the next state is absorbing. Also an additional dictionary is returned (possibly empty).

**static _bound**(*x*, *min_value*, *max_value*)
Method used to bound state and action variables.

**Parameters**

- **x** – the variable to bound;
- **min_value** – the minimum value;
- **max_value** – the maximum value;

**Returns** The bounded variable.

**info**
An object containing the info of the environment.

**Type** Returns

**seed**(*seed*)
Set the seed of the environment.

**Parameters** **seed** (*float*) – the value of the seed.

**stop**()
Method used to stop an mdp. Useful when dealing with real world environments, simulators, or when using openai-gym rendering

### Finite MDP

**class** mushroom.environments.finite_mdp.**FiniteMDP**(*p*, *rew*, *mu=None*, *gamma=0.9*, *horizon=inf*)
Bases: mushroom.environments.environment.Environment

Finite Markov Decision Process.

**__init__**(*p*, *rew*, *mu=None*, *gamma=0.9*, *horizon=inf*)
Constructor.

**Parameters**

- **p** (*np.ndarray*) – transition probability matrix;
- **rew** (*np.ndarray*) – reward matrix;
- **mu** (*np.ndarray, None*) – initial state probability distribution;
- **gamma** (*float, 9*) – discount factor;
- **horizon** (*int, np.inf*) – the horizon.

**reset**(*state=None*)
Reset the current state.

**Parameters** **state** (*np.ndarray, None*) – the state to set to the current state.

**Returns** The current state.

**step**(*action*)
Move the agent from its current state according to the action.

**Parameters** **action** (*np.ndarray*) – the action to execute.

**Returns** The state reached by the agent executing action in its current state, the reward obtained in the transition and a flag to signal if the next state is absorbing. Also an additional dictionary is returned (possibly empty).

**static _bound**(*x*, *min_value*, *max_value*)
Method used to bound state and action variables.

**Parameters**

- **x** – the variable to bound;
- **min_value** – the minimum value;
- **max_value** – the maximum value;

**Returns** The bounded variable.

**info**
An object containing the info of the environment.

**Type** Returns

**seed**(*seed*)
Set the seed of the environment.

**Parameters** **seed** (*float*) – the value of the seed.

**stop**()
Method used to stop an mdp. Useful when dealing with real world environments, simulators, or when using openai-gym rendering

### Grid World

**class** mushroom.environments.grid_world.**AbstractGridWorld**(*mdp_info*, *height*, *width*, *start*, *goal*)
Bases: mushroom.environments.environment.Environment

Abstract class to build a grid world.

**__init__**(*mdp_info*, *height*, *width*, *start*, *goal*)
Constructor.

**Parameters**

- **height** (*int*) – height of the grid;
- **width** (*int*) – width of the grid;
- **start** (*tuple*) – x-y coordinates of the goal;
- **goal** (*tuple*) – x-y coordinates of the goal.

**reset**(*state=None*)
Reset the current state.

**Parameters** **state** (*np.ndarray, None*) – the state to set to the current state.

**Returns** The current state.

**step**(*action*)
Move the agent from its current state according to the action.

**Parameters** **action** (*np.ndarray*) – the action to execute.

**Returns** The state reached by the agent executing `action` in its current state, the reward obtained in the transition and a flag to signal if the next state is absorbing. Also an additional dictionary is returned (possibly empty).

**static _bound**(*x*, *min_value*, *max_value*)
Method used to bound state and action variables.

**Parameters**

- **x** – the variable to bound;
- **min_value** – the minimum value;
- **max_value** – the maximum value;

**Returns** The bounded variable.

**info**
An object containing the info of the environment.

**Type** Returns

**seed**(*seed*)
Set the seed of the environment.

**Parameters** **seed** (*float*) – the value of the seed.

**stop**()
Method used to stop an mdp. Useful when dealing with real world environments, simulators, or when using openai-gym rendering

**class** mushroom.environments.grid_world.**GridWorld**(*height*, *width*, *goal*, *start=(0, 0)*)
Bases: *mushroom.environments.grid_world.AbstractGridWorld*

Standard grid world.

**__init__**(*height*, *width*, *goal*, *start=(0, 0)*)
Constructor.

**Parameters**

- **height** (*int*) – height of the grid;
- **width** (*int*) – width of the grid;
- **start** (*tuple*) – x-y coordinates of the goal;
- **goal** (*tuple*) – x-y coordinates of the goal.

**static _bound**(*x*, *min_value*, *max_value*)
Method used to bound state and action variables.

**Parameters**

- **x** – the variable to bound;
- **min_value** – the minimum value;
- **max_value** – the maximum value;

**Returns** The bounded variable.

**info**
An object containing the info of the environment.

**Type** Returns

**reset**(*state=None*)
Reset the current state.

**Parameters** **state** (*np.ndarray, None*) – the state to set to the current state.

**Returns** The current state.

**seed**(*seed*)
Set the seed of the environment.

**Parameters** **seed** (*float*) – the value of the seed.

**step**(*action*)
Move the agent from its current state according to the action.

**Parameters** **action** (*np.ndarray*) – the action to execute.

**Returns** The state reached by the agent executing `action` in its current state, the reward obtained in the transition and a flag to signal if the next state is absorbing. Also an additional dictionary is returned (possibly empty).

**stop**()
Method used to stop an mdp. Useful when dealing with real world environments, simulators, or when using openai-gym rendering

**class** mushroom.environments.grid_world.**GridWorldVanHasselt**(*height=3*, *width=3*, *goal=(0, 2)*, *start=(2, 0)*)
Bases: *mushroom.environments.grid_world.AbstractGridWorld*

A variant of the grid world as presented in: "Double Q-Learning". Hasselt H. V.. 2010.

**__init__**(*height=3*, *width=3*, *goal=(0, 2)*, *start=(2, 0)*)
Constructor.

**Parameters**

- **height** (*int*) – height of the grid;
- **width** (*int*) – width of the grid;
- **start** (*tuple*) – x-y coordinates of the goal;
- **goal** (*tuple*) – x-y coordinates of the goal.

**static _bound**(*x*, *min_value*, *max_value*)
Method used to bound state and action variables.

**Parameters**

- **x** – the variable to bound;
- **min_value** – the minimum value;
- **max_value** – the maximum value;

**Returns** The bounded variable.

**info**
An object containing the info of the environment.

**Type** Returns

**reset**(*state=None*)
Reset the current state.

**Parameters** **state** (*np.ndarray, None*) – the state to set to the current state.

**Returns** The current state.

**seed**(*seed*)
Set the seed of the environment.

**Parameters** **seed** (*float*) – the value of the seed.

**step**(*action*)
Move the agent from its current state according to the action.

**Parameters** **action** (*np.ndarray*) – the action to execute.

**Returns** The state reached by the agent executing `action` in its current state, the reward obtained in the transition and a flag to signal if the next state is absorbing. Also an additional dictionary is returned (possibly empty).

**stop**()
Method used to stop an mdp. Useful when dealing with real world environments, simulators, or when using openai-gym rendering

### Gym

**class** mushroom.environments.gym_env.**Gym**(*name*, *horizon*, *gamma*)
Bases: mushroom.environments.environment.Environment

Interface for OpenAI Gym environments. It makes it possible to use every Gym environment just providing the id, except for the Atari games that are managed in a separate class.

**__init__**(*name*, *horizon*, *gamma*)
Constructor.

**Parameters**

- **name** (*str*) – gym id of the environment;
- **horizon** (*int*) – the horizon;
- **gamma** (*float*) – the discount factor.

**reset**(*state=None*)
Reset the current state.

**Parameters** **state** (*np.ndarray, None*) – the state to set to the current state.

**Returns** The current state.

**step**(*action*)
Move the agent from its current state according to the action.

**Parameters** **action** (*np.ndarray*) – the action to execute.

**Returns** The state reached by the agent executing `action` in its current state, the reward obtained in the transition and a flag to signal if the next state is absorbing. Also an additional dictionary is returned (possibly empty).

**stop**()
Method used to stop an mdp. Useful when dealing with real world environments, simulators, or when using openai-gym rendering

**static _bound**(*x*, *min_value*, *max_value*)
Method used to bound state and action variables.

**Parameters**

- **x** – the variable to bound;
- **min_value** – the minimum value;
- **max_value** – the maximum value;

**Returns** The bounded variable.

**info**
An object containing the info of the environment.

**Type** Returns

**seed**(*seed*)
Set the seed of the environment.

**Parameters** **seed** (*float*) – the value of the seed.

### Inverted pendulum

**class** mushroom.environments.inverted_pendulum.**InvertedPendulum**(*random_start=False*, *m=1.0*, *l=1.0*, *g=9.8*, *mu=0.01*, *max_u=5.0*, *horizon=5000*, *gamma=0.99*)

Bases: mushroom.environments.environment.Environment

The Inverted Pendulum environment (continuous version) as presented in: "Reinforcement Learning In Continuous Time and Space". Doya K.. 2000. "Off-Policy Actor-Critic". Degris T. et al.. 2012. "Deterministic Policy Gradient Algorithms". Silver D. et al. 2014.

**__init__**(*random_start=False*, *m=1.0*, *l=1.0*, *g=9.8*, *mu=0.01*, *max_u=5.0*, *horizon=5000*, *gamma=0.99*)

Constructor.

**Parameters**

- **random_start** (*bool, False*) – whether to start from a random position or from the horizontal one;
- **m** (*float, 1.0*) – mass of the pendulum;
- **l** (*float, 1.0*) – length of the pendulum;
- **g** (*float, 9.8*) – gravity acceleration constant;
- **mu** (*float, 1e-2*) – friction constant of the pendulum;
- **max_u** (*float, 5.0*) – maximum allowed input torque;
- **horizon** (*int, 5000*) – horizon of the problem;
- **gamma** (*int, 99*) – discount factor.

**reset**(*state=None*)

Reset the current state.

**Parameters** **state** (*np.ndarray, None*) – the state to set to the current state.

**Returns** The current state.

**step**(*action*)

Move the agent from its current state according to the action.

**Parameters** **action** (*np.ndarray*) – the action to execute.

**Returns** The state reached by the agent executing `action` in its current state, the reward obtained in the transition and a flag to signal if the next state is absorbing. Also an additional dictionary is returned (possibly empty).

**stop**()

Method used to stop an mdp. Useful when dealing with real world environments, simulators, or when using openai-gym rendering

**static _bound**(*x*, *min_value*, *max_value*)

Method used to bound state and action variables.

**Parameters**

- **x** – the variable to bound;

- **min_value** – the minimum value;
- **max_value** – the maximum value;

**Returns** The bounded variable.

**info**
An object containing the info of the environment.

**Type** Returns

**seed**(*seed*)
Set the seed of the environment.

**Parameters** **seed** (*float*) – the value of the seed.

**class** mushroom.environments.inverted_pendulum.**InvertedPendulumDiscrete**(*m=2.0*, *M=8.0*, *l=0.5*, *g=9.8*, *mu=0.01*, *max_u=50.0*, *noise_u=10.0*, *horizon=3000*, *gamma=0.95*)

Bases: mushroom.environments.environment.Environment

The Inverted Pendulum environment as presented in: "Least-Squares Policy Iteration". Lagoudakis M. G. and Parr R.. 2003.

**__init__**(*m=2.0*, *M=8.0*, *l=0.5*, *g=9.8*, *mu=0.01*, *max_u=50.0*, *noise_u=10.0*, *horizon=3000*, *gamma=0.95*)
Constructor.

**Parameters**

- **m** (*float, 2.0*) – mass of the pendulum;
- **M** (*float, 8.0*) – mass of the cart;
- **l** (*float, 5*) – length of the pendulum;
- **g** (*float, 9.8*) – gravity acceleration constant;
- **mu** (*float, 1e-2*) – friction constant of the pendulum;
- **max_u** (*float, 50.*) – maximum allowed input torque;
- **noise_u** (*float, 10.*) – maximum noise on the action;
- **horizon** (*int, 3000*) – horizon of the problem;
- **gamma** (*int, 95*) – discount factor.

**reset**(*state=None*)
Reset the current state.

**Parameters** **state** (*np.ndarray, None*) – the state to set to the current state.

**Returns** The current state.

**step**(*action*)
Move the agent from its current state according to the action.

**Parameters** **action** (*np.ndarray*) – the action to execute.

**Returns** The state reached by the agent executing `action` in its current state, the reward obtained in the transition and a flag to signal if the next state is absorbing. Also an additional dictionary is returned (possibly empty).

**stop**()
Method used to stop an mdp. Useful when dealing with real world environments, simulators, or when using openai-gym rendering

**static _bound**(*x*, *min_value*, *max_value*)
Method used to bound state and action variables.

**Parameters**

- **x** – the variable to bound;
- **min_value** – the minimum value;
- **max_value** – the maximum value;

**Returns** The bounded variable.

**info**
An object containing the info of the environment.

**Type** Returns

**seed**(*seed*)
Set the seed of the environment.

**Parameters** **seed** (*float*) – the value of the seed.

### LQR

**class** mushroom.environments.lqr.**LQR**(*A*, *B*, *Q*, *R*, *random_init=False*, *gamma=0.9*, *horizon=50*)
Bases: mushroom.environments.environment.Environment

This class implements a Linear-Quadratic Regulator. This task aims to minimize the undesired deviations from nominal values of some controller settings in control problems. The system equations in this task are:

$$x_{t+1} = Ax_t + Bu_t$$

where x is the state and u is the control signal.

The reward function is given by:

$$r_t = -\left(x_t^T Q x_t + u_t^T R u_t\right)$$

"Policy gradient approaches for multi-objective sequential decision making". Parisi S., Pirotta M., Smacchia N., Bascetta L., Restelli M.. 2014

**__init__**(*A*, *B*, *Q*, *R*, *random_init=False*, *gamma=0.9*, *horizon=50*)
Constructor.

**Args:** A (np.ndarray): the state dynamics matrix; B (np.ndarray): the action dynamics matrix; Q (np.ndarray): reward weight matrix for state; R (np.ndarray): reward weight matrix for action; random_init (bool, False): start from a random state; gamma (float, 0.9): discount factor; horizon (int, 50): horizon of the mdp.

**static generate**(*dimensions*, *eps=0.1*, *index=0*, *random_init=False*, *gamma=0.9*, *horizon=50*)
Factory method that generates an lqr with identity dynamics and symmetric reward matrices.

**Parameters**

- **dimensions** (*int*) – number of state-action dimensions;
- **eps** (*double, 0.1*) – reward matrix weights specifier;
- **index** (*int, 0*) – selector for the principal state;
- **random_init** (*bool, False*) – start from a random state;
- **gamma** (*float, 0.9*) – discount factor;
- **horizon** (*int, 50*) – horizon of the mdp.

**reset**(*state=None*)
Reset the current state.

**Parameters** **state** (*np.ndarray, None*) – the state to set to the current state.

**Returns** The current state.

**step**(*action*)
Move the agent from its current state according to the action.

**Parameters** **action** (*np.ndarray*) – the action to execute.

**Returns** The state reached by the agent executing `action` in its current state, the reward obtained in the transition and a flag to signal if the next state is absorbing. Also an additional dictionary is returned (possibly empty).

**static _bound**(*x*, *min_value*, *max_value*)
Method used to bound state and action variables.

**Parameters**

- **x** – the variable to bound;
- **min_value** – the minimum value;
- **max_value** – the maximum value;

**Returns** The bounded variable.

**info**
An object containing the info of the environment.

**Type** Returns

**seed**(*seed*)
Set the seed of the environment.

**Parameters** **seed** (*float*) – the value of the seed.

**stop**()
Method used to stop an mdp. Useful when dealing with real world environments, simulators, or when using openai-gym rendering

### Mujoco

### Segway

**class** mushroom.environments.segway.**Segway**(*random_start=False*)
Bases: mushroom.environments.environment.Environment

The Segway environment (continuous version) as presented in: "Deep Learning for Actor-Critic Reinforcement Learning". Xueli Jia. 2015.

**__init__**(*random_start=False*)
Constructor.

**Parameters** **random_start** (*bool, False*) – whether to start from a random position or from the horizontal one.

**reset**(*state=None*)
Reset the current state.

**Parameters** **state** (*np.ndarray, None*) – the state to set to the current state.

**Returns** The current state.

**step**(*action*)
Move the agent from its current state according to the action.

**Parameters** **action** (*np.ndarray*) – the action to execute.

**Returns** The state reached by the agent executing `action` in its current state, the reward obtained in the transition and a flag to signal if the next state is absorbing. Also an additional dictionary is returned (possibly empty).

**static _bound**(*x*, *min_value*, *max_value*)
Method used to bound state and action variables.

**Parameters**

- **x** – the variable to bound;
- **min_value** – the minimum value;
- **max_value** – the maximum value;

**Returns** The bounded variable.

**info**
An object containing the info of the environment.

**Type** Returns

**seed**(*seed*)
Set the seed of the environment.

**Parameters** **seed** (*float*) – the value of the seed.

**stop**()
Method used to stop an mdp. Useful when dealing with real world environments, simulators, or when using openai-gym rendering

### Ship steering

**class** mushroom.environments.ship_steering.**ShipSteering**(*small=True*, *n_steps_action=3*)
Bases: `mushroom.environments.environment.Environment`

The Ship Steering environment as presented in: "Hierarchical Policy Gradient Algorithms". Ghavamzadeh M. and Mahadevan S.. 2013.

**__init__**(*small=True*, *n_steps_action=3*)
Constructor.

**Parameters**

- **small** (*bool, True*) – whether to use a small state space or not.
- **n_steps_action** (*int, 3*) – number of integration intervals for each step of the mdp.

**reset**(*state=None*)
Reset the current state.

**Parameters** **state** (*np.ndarray, None*) – the state to set to the current state.

**Returns** The current state.

**step**(*action*)
Move the agent from its current state according to the action.

**Parameters** **action** (*np.ndarray*) – the action to execute.

**Returns** The state reached by the agent executing `action` in its current state, the reward obtained in the transition and a flag to signal if the next state is absorbing. Also an additional dictionary is returned (possibly empty).

**stop**()
Method used to stop an mdp. Useful when dealing with real world environments, simulators, or when using openai-gym rendering

**static _bound**(*x*, *min_value*, *max_value*)
Method used to bound state and action variables.

**Parameters**

- **x** – the variable to bound;
- **min_value** – the minimum value;
- **max_value** – the maximum value;

**Returns** The bounded variable.

**info**
An object containing the info of the environment.

**Type** Returns

**seed**(*seed*)
Set the seed of the environment.

**Parameters** **seed** (*float*) – the value of the seed.

### Generators

### Grid world

mushroom.environments.generators.grid_world.**generate_grid_world**(*grid*, *prob*, *pos_rew*, *neg_rew*, *gamma=0.9*, *horizon=100*)
This Grid World generator requires a .txt file to specify the shape of the grid world and the cells. There are five types of cells: 'S' is the starting position where the agent is; 'G' is the goal state; '.' is a normal cell; '*' is a hole, when the agent steps on a hole, it receives a negative reward and the episode ends; '#' is a wall, when the agent is supposed to step on a wall, it actually remains in its current state. The initial states distribution is uniform among all the initial states provided.

The grid is expected to be rectangular.

**Parameters**

- **grid** (*str*) – the path of the file containing the grid structure;
- **prob** (*float*) – probability of success of an action;
- **pos_rew** (*float*) – reward obtained in goal states;
- **neg_rew** (*float*) – reward obtained in "hole" states;
- **gamma** (*float, 9*) – discount factor;
- **horizon** (*int, 100*) – the horizon.

**Returns** A FiniteMDP object built with the provided parameters.

mushroom.environments.generators.grid_world.**parse_grid**(*grid*)

Parse the grid file:

**Parameters** **grid** (*str*) – the path of the file containing the grid structure;

**Returns** A list containing the grid structure.

mushroom.environments.generators.grid_world.**compute_probabilities**(*grid_map*, *cell_list*, *prob*)

Compute the transition probability matrix.

**Parameters**

- **grid_map** (*list*) – list containing the grid structure;
- **cell_list** (*list*) – list of non-wall cells;
- **prob** (*float*) – probability of success of an action.

**Returns** The transition probability matrix;

mushroom.environments.generators.grid_world.**compute_reward**(*grid_map*, *cell_list*, *pos_rew*, *neg_rew*)

Compute the reward matrix.

**Parameters**

- **grid_map** (*list*) – list containing the grid structure;
- **cell_list** (*list*) – list of non-wall cells;
- **pos_rew** (*float*) – reward obtained in goal states;
- **neg_rew** (*float*) – reward obtained in "hole" states;

**Returns** The reward matrix.

mushroom.environments.generators.grid_world.**compute_mu**(*grid_map*, *cell_list*)

Compute the initial states distribution.

**Parameters**

- **grid_map** (*list*) – list containing the grid structure;
- **cell_list** (*list*) – list of non-wall cells.

**Returns** The initial states distribution.

### Simple chain

mushroom.environments.generators.simple_chain.**generate_simple_chain**(*state_n*, *goal_states*, *prob*, *rew*, *mu=None*, *gamma=0.9*, *horizon=100*)

Simple chain generator.

**Parameters**

- **state_n** (*int*) – number of states;
- **goal_states** (*list*) – list of goal states;
- **prob** (*float*) – probability of success of an action;
- **rew** (*float*) – reward obtained in goal states;
- **mu** (*np.ndarray*) – initial state probability distribution;
- **gamma** (*float, 9*) – discount factor;
- **horizon** (*int, 100*) – the horizon.

**Returns** A FiniteMDP object built with the provided parameters.

mushroom.environments.generators.simple_chain.**compute_probabilities**(*state_n*, *prob*)

Compute the transition probability matrix.

**Parameters**

- **state_n** (*int*) – number of states;
- **prob** (*float*) – probability of success of an action.

**Returns** The transition probability matrix;

mushroom.environments.generators.simple_chain.**compute_reward**(*state_n*, *goal_states*, *rew*)

Compute the reward matrix.

**Parameters**

- **state_n** (*int*) – number of states;
- **goal_states** (*list*) – list of goal states;
- **rew** (*float*) – reward obtained in goal states.

**Returns** The reward matrix.

### Taxi

mushroom.environments.generators.taxi.**generate_taxi**(*grid*, *prob=0.9*, *rew=(0*, *1*, *3*, *15)*, *gamma=0.99*, *horizon=inf*)

This Taxi generator requires a .txt file to specify the shape of the grid world and the cells. There are five types of cells: 'S' is the starting where the agent is; 'G' is the goal state; '.' is a normal cell; 'F' is a passenger, when the agent steps on a hole, it picks up it. '#' is a wall, when the agent is supposed to step on a wall, it actually

remains in its current state. The initial states distribution is uniform among all the initial states provided. The episode terminates when the agent reaches the goal state. The reward is always 0, except for the goal state where it depends on the number of collected passengers. Each action has a certain probability of success and, if it fails, the agent goes in a perpendicular direction from the supposed one.

The grid is expected to be rectangular.

This problem is inspired from: "Bayesian Q-Learning". Dearden R. et al.. 1998.

**Parameters**

- **grid** (*str*) – the path of the file containing the grid structure;
- **prob** (*float, 9*) – probability of success of an action;
- **rew** (*tuple, (0, 1, 3, 15)*) – rewards obtained in goal states;
- **gamma** (*float, 99*) – discount factor;
- **horizon** (*int, np.inf*) – the horizon.

**Returns** A FiniteMDP object built with the provided parameters.

mushroom.environments.generators.taxi.**parse_grid**(*grid*)

Parse the grid file:

**Parameters** **grid** (*str*) – the path of the file containing the grid structure.

**Returns** A list containing the grid structure.

mushroom.environments.generators.taxi.**compute_probabilities**(*grid_map*, *cell_list*, *passenger_list*, *prob*)

Compute the transition probability matrix.

**Parameters**

- **grid_map** (*list*) – list containing the grid structure;
- **cell_list** (*list*) – list of non-wall cells;
- **passenger_list** (*list*) – list of passenger cells;
- **prob** (*float*) – probability of success of an action.

**Returns** The transition probability matrix;

mushroom.environments.generators.taxi.**compute_reward**(*grid_map*, *cell_list*, *passenger_list*, *rew*)

Compute the reward matrix.

**Parameters**

- **grid_map** (*list*) – list containing the grid structure;
- **cell_list** (*list*) – list of non-wall cells;
- **passenger_list** (*list*) – list of passenger cells;
- **rew** (*tuple*) – rewards obtained in goal states.

**Returns** The reward matrix.

mushroom.environments.generators.taxi.**compute_mu**(*grid_map*, *cell_list*, *passenger_list*)

Compute the initial states distribution.

**Parameters**

- **grid_map** (*list*) – list containing the grid structure;

- **cell_list** (*list*) – list of non-wall cells;
- **passenger_list** (*list*) – list of passenger cells.

**Returns** The initial states distribution.

### 3.1.3 Algorithms

Mushroom provides the implementations of several algorithms belonging to all categories of RL:

- value-based;
- policy-search;
- actor-critic.

One can easily implement customized algorithms following the structure of the already available ones.

#### Agent

**class** mushroom.algorithms.agent.**Agent**(*policy*, *mdp_info*, *features=None*)

Bases: object

This class implements the functions to manage the agent (e.g. move the agent following its policy).

**__init__**(*policy*, *mdp_info*, *features=None*)

Constructor.

**Parameters**

- **policy** (Policy) – the policy followed by the agent;
- **mdp_info** (MDPInfo) – information about the MDP;
- **features** (*object, None*) – features to extract from the state.

**fit**(*dataset*)

Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**draw_action**(*state*)

Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()

Called by the agent when a new episode starts.

**stop**()

Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

#### Subpackages

#### Value

### TD

**class** mushroom.algorithms.value.td.**TD**(*approximator*, *policy*, *mdp_info*, *learning_rate*, *features=None*)

Bases: *mushroom.algorithms.agent.Agent*

Implements functions to run TD algorithms.

**__init__**(*approximator*, *policy*, *mdp_info*, *learning_rate*, *features=None*)

Constructor.

**Parameters**

- **approximator** (*object*) – the approximator to use to fit the Q-function;
- **learning_rate** (*Parameter*) – the learning rate.

**fit**(*dataset*)

Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**static _parse**(*dataset*)

Utility to parse the dataset that is supposed to contain only a sample.

**Parameters** **dataset** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag.

**_update**(*state*, *action*, *reward*, *next_state*, *absorbing*)

Update the Q-table.

**Parameters**

- **state** (*np.ndarray*) – state;
- **action** (*np.ndarray*) – action;
- **reward** (*np.ndarray*) – reward;
- **next_state** (*np.ndarray*) – next state;
- **absorbing** (*np.ndarray*) – absorbing flag.

**draw_action**(*state*)

Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()

Called by the agent when a new episode starts.

**stop**()

Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.td.**QLearning**(*policy*, *mdp_info*, *learning_rate*)

Bases: *mushroom.algorithms.value.td.TD*

Q-Learning algorithm. “Learning from Delayed Rewards”. Watkins C.J.C.H.. 1989.

**__init__**(*policy*, *mdp_info*, *learning_rate*)

Constructor.

**Parameters**

- **approximator** (*object*) – the approximator to use to fit the Q-function;
- **learning_rate** (Parameter) – the learning rate.

**_update**(*state*, *action*, *reward*, *next_state*, *absorbing*)
Update the Q-table.

**Parameters**

- **state** (*np.ndarray*) – state;
- **action** (*np.ndarray*) – action;
- **reward** (*np.ndarray*) – reward;
- **next_state** (*np.ndarray*) – next state;
- **absorbing** (*np.ndarray*) – absorbing flag.

**static _parse**(*dataset*)
Utility to parse the dataset that is supposed to contain only a sample.

**Parameters** **dataset** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.td.**DoubleQLearning**(*policy*, *mdp_info*, *learning_rate*)
Bases: *mushroom.algorithms.value.td.TD*

Double Q-Learning algorithm. “Double Q-Learning”. Hasselt H. V.. 2010.

**__init__**(*policy*, *mdp_info*, *learning_rate*)
Constructor.

**Parameters**

- **approximator** (*object*) – the approximator to use to fit the Q-function;
- **learning_rate** (Parameter) – the learning rate.

**_update**(*state*, *action*, *reward*, *next_state*, *absorbing*)
Update the Q-table.

**Parameters**

- **state** (*np.ndarray*) – state;

- **action** (*np.ndarray*) – action;
- **reward** (*np.ndarray*) – reward;
- **next_state** (*np.ndarray*) – next state;
- **absorbing** (*np.ndarray*) – absorbing flag.

**static _parse**(*dataset*)
Utility to parse the dataset that is supposed to contain only a sample.

**Parameters** **dataset** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.td.**WeightedQLearning**(*policy*, *mdp_info*, *learning_rate*, *sampling=True*, *precision=1000*, *weighted_policy=False*)
Bases: *mushroom.algorithms.value.td.TD*

Weighted Q-Learning algorithm. "Estimating the Maximum Expected Value through Gaussian Approximation". D'Eramo C. et. al.. 2016.

**__init__**(*policy*, *mdp_info*, *learning_rate*, *sampling=True*, *precision=1000*, *weighted_policy=False*)
Constructor.

**Parameters**

- **sampling** (*bool, True*) – use the approximated version to speed up the computation;
- **precision** (*int, 1000*) – number of samples to use in the approximated version;
- **weighted_policy** (*bool, False*) – whether to use the weighted policy or not.

**_update**(*state*, *action*, *reward*, *next_state*, *absorbing*)
Update the Q-table.

**Parameters**

- **state** (*np.ndarray*) – state;
- **action** (*np.ndarray*) – action;
- **reward** (*np.ndarray*) – reward;
- **next_state** (*np.ndarray*) – next state;

- **absorbing** (*np.ndarray*) – absorbing flag.

**_next_q**(*next_state*)

**Parameters** **next_state** (*np.ndarray*) – the state where next action has to be evaluated.

**Returns** The weighted estimator value in `next_state`.

**static _parse**(*dataset*)
Utility to parse the dataset that is supposed to contain only a sample.

**Parameters** **dataset** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.td.**SpeedyQLearning**(*policy*, *mdp_info*, *learning_rate*)
Bases: *mushroom.algorithms.value.td.TD*

Speedy Q-Learning algorithm. “Speedy Q-Learning”. Ghavamzadeh et. al.. 2011.

**__init__**(*policy*, *mdp_info*, *learning_rate*)
Constructor.

**Parameters**

- **approximator** (*object*) – the approximator to use to fit the Q-function;
- **learning_rate** (Parameter) – the learning rate.

**_update**(*state*, *action*, *reward*, *next_state*, *absorbing*)
Update the Q-table.

**Parameters**

- **state** (*np.ndarray*) – state;
- **action** (*np.ndarray*) – action;
- **reward** (*np.ndarray*) – reward;
- **next_state** (*np.ndarray*) – next state;
- **absorbing** (*np.ndarray*) – absorbing flag.

**static _parse**(*dataset*)
Utility to parse the dataset that is supposed to contain only a sample.

**Parameters** **dataset** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.td.**SARSA**(*policy*, *mdp_info*, *learning_rate*)
Bases: *mushroom.algorithms.value.td.TD*

SARSA algorithm.

**__init__**(*policy*, *mdp_info*, *learning_rate*)
Constructor.

**Parameters**

- **approximator** (*object*) – the approximator to use to fit the Q-function;
- **learning_rate** (Parameter) – the learning rate.

**_update**(*state*, *action*, *reward*, *next_state*, *absorbing*)
Update the Q-table.

**Parameters**

- **state** (*np.ndarray*) – state;
- **action** (*np.ndarray*) – action;
- **reward** (*np.ndarray*) – reward;
- **next_state** (*np.ndarray*) – next state;
- **absorbing** (*np.ndarray*) – absorbing flag.

**static _parse**(*dataset*)
Utility to parse the dataset that is supposed to contain only a sample.

**Parameters** **dataset** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.td.**SARSALambdaDiscrete**(*policy*, *mdp_info*, *learning_rate*, *lambda_coeff*, *trace='replacing'*)

Bases: *mushroom.algorithms.value.td.TD*

Discrete version of SARSA(lambda) algorithm.

**__init__**(*policy*, *mdp_info*, *learning_rate*, *lambda_coeff*, *trace='replacing'*)
Constructor.

**Parameters**

- **lambda_coeff** (*float*) – eligibility trace coefficient;
- **trace** (*str, 'replacing'*) – type of eligibility trace to use.

**_update**(*state*, *action*, *reward*, *next_state*, *absorbing*)
Update the Q-table.

**Parameters**

- **state** (*np.ndarray*) – state;
- **action** (*np.ndarray*) – action;
- **reward** (*np.ndarray*) – reward;
- **next_state** (*np.ndarray*) – next state;
- **absorbing** (*np.ndarray*) – absorbing flag.

**episode_start**()
Called by the agent when a new episode starts.

**static _parse**(*dataset*)
Utility to parse the dataset that is supposed to contain only a sample.

**Parameters** **dataset** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.td.**SARSALambdaContinuous**(*approximator*, *policy*, *mdp_info*, *learning_rate*, *lambda_coeff*, *features*, *approximator_params=None*)
Bases: *mushroom.algorithms.value.td.TD*

Continuous version of SARSA(lambda) algorithm.

**__init__**(*approximator*, *policy*, *mdp_info*, *learning_rate*, *lambda_coeff*, *features*, *approximator_params=None*)
Constructor.

**Parameters** **lambda_coeff** (*float*) – eligibility trace coefficient.

**_update**(*state*, *action*, *reward*, *next_state*, *absorbing*)
Update the Q-table.

**Parameters**

- **state** (*np.ndarray*) – state;
- **action** (*np.ndarray*) – action;
- **reward** (*np.ndarray*) – reward;
- **next_state** (*np.ndarray*) – next state;
- **absorbing** (*np.ndarray*) – absorbing flag.

**episode_start**()
Called by the agent when a new episode starts.

**static _parse**(*dataset*)
Utility to parse the dataset that is supposed to contain only a sample.

**Parameters** **dataset** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.td.**ExpectedSARSA**(*policy*, *mdp_info*, *learning_rate*)
Bases: *mushroom.algorithms.value.td.TD*

Expected SARSA algorithm. "A theoretical and empirical analysis of Expected Sarsa". Seijen H. V. et al.. 2009.

**__init__**(*policy*, *mdp_info*, *learning_rate*)
Constructor.

**Parameters**

- **approximator** (*object*) – the approximator to use to fit the Q-function;
- **learning_rate** (Parameter) – the learning rate.

**_update**(*state*, *action*, *reward*, *next_state*, *absorbing*)
Update the Q-table.

**Parameters**

- **state** (*np.ndarray*) – state;
- **action** (*np.ndarray*) – action;
- **reward** (*np.ndarray*) – reward;
- **next_state** (*np.ndarray*) – next state;
- **absorbing** (*np.ndarray*) – absorbing flag.

**static _parse**(*dataset*)
Utility to parse the dataset that is supposed to contain only a sample.

**Parameters** **dataset** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.td.**TrueOnlineSARSALambda**(*policy*, *mdp_info*, *learning_rate*, *lambda_coeff*, *features*, *approximator_params=None*)
Bases: *mushroom.algorithms.value.td.TD*

True Online SARSA(lambda) with linear function approximation. “True Online TD(lambda)”. Seijen H. V. et al.. 2014.

**__init__**(*policy*, *mdp_info*, *learning_rate*, *lambda_coeff*, *features*, *approximator_params=None*)
Constructor.

**Parameters** **lambda_coeff** (*float*) – eligibility trace coefficient.

**_update**(*state*, *action*, *reward*, *next_state*, *absorbing*)
Update the Q-table.

**Parameters**

- **state** (*np.ndarray*) – state;
- **action** (*np.ndarray*) – action;
- **reward** (*np.ndarray*) – reward;
- **next_state** (*np.ndarray*) – next state;
- **absorbing** (*np.ndarray*) – absorbing flag.

**episode_start**()
Called by the agent when a new episode starts.

**static _parse**(*dataset*)
Utility to parse the dataset that is supposed to contain only a sample.

**Parameters** **dataset** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.td.**RLearning**(*policy*, *mdp_info*, *learning_rate*, *beta*)
Bases: *mushroom.algorithms.value.td.TD*

R-Learning algorithm. "A Reinforcement Learning Method for Maximizing Undiscounted Rewards". Schwartz A.. 1993.

**__init__**(*policy*, *mdp_info*, *learning_rate*, *beta*)
Constructor.

**Parameters** **beta** (Parameter) – beta coefficient.

**_update**(*state*, *action*, *reward*, *next_state*, *absorbing*)
Update the Q-table.

**Parameters**

- **state** (*np.ndarray*) – state;
- **action** (*np.ndarray*) – action;
- **reward** (*np.ndarray*) – reward;
- **next_state** (*np.ndarray*) – next state;
- **absorbing** (*np.ndarray*) – absorbing flag.

**static _parse**(*dataset*)
Utility to parse the dataset that is supposed to contain only a sample.

**Parameters** **dataset** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.td.**RQLearning**(*policy*, *mdp_info*, *learning_rate*, *off_policy=False*, *beta=None*, *delta=None*)
Bases: *mushroom.algorithms.value.td.TD*

RQ-Learning algorithm. "Exploiting Structure and Uncertainty of Bellman Updates in Markov Decision Processes". Tateo D. et al.. 2017.

**__init__**(*policy*, *mdp_info*, *learning_rate*, *off_policy=False*, *beta=None*, *delta=None*)
Constructor.

**Parameters**

- **off_policy** (*bool, False*) – whether to use the off policy setting or the online one;
- **beta** (*Parameter, None*) – beta coefficient;
- **delta** (*Parameter, None*) – delta coefficient.

**static _parse**(*dataset*)
Utility to parse the dataset that is supposed to contain only a sample.

**Parameters** **dataset** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag.

**_update**(*state*, *action*, *reward*, *next_state*, *absorbing*)
Update the Q-table.

**Parameters**

- **state** (*np.ndarray*) – state;
- **action** (*np.ndarray*) – action;
- **reward** (*np.ndarray*) – reward;
- **next_state** (*np.ndarray*) – next state;
- **absorbing** (*np.ndarray*) – absorbing flag.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**_next_q**(*next_state*)

**Parameters** **next_state** (*np.ndarray*) – the state where next action has to be evaluated.

**Returns** The weighted estimator value in 'next_state'.

### Batch TD

**class** mushroom.algorithms.value.batch_td.**BatchTD**(*approximator*, *policy*, *mdp_info*, *fit_params=None*, *approximator_params=None*, *features=None*)
Bases: *mushroom.algorithms.agent.Agent*

Abstract class to implement a generic Batch TD algorithm.

**__init__**(*approximator*, *policy*, *mdp_info*, *fit_params=None*, *approximator_params=None*, *features=None*)
Constructor.

**Parameters**

- **approximator** (*object*) – approximator used by the algorithm and the policy.
- **fit_params** (*dict, None*) – parameters of the fitting algorithm of the approximator;
- **approximator_params** (*dict, None*) – parameters of the approximator to build;

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.batch_td.**FQI**(*approximator*, *policy*, *mdp_info*, *n_iterations*, *fit_params=None*, *approximator_params=None*, *quiet=False*, *boosted=False*)
Bases: *mushroom.algorithms.value.batch_td.BatchTD*

Fitted Q-Iteration algorithm. "Tree-Based Batch Mode Reinforcement Learning", Ernst D. et al.. 2005.

**__init__**(*approximator*, *policy*, *mdp_info*, *n_iterations*, *fit_params=None*, *approximator_params=None*, *quiet=False*, *boosted=False*)
Constructor.

**Parameters**

- **n_iterations** (*int*) – number of iterations to perform for training;
- **quiet** (*bool, False*) – whether to show the progress bar or not;
- **boosted** (*bool, False*) – whether to use boosted FQI or not.

**fit**(*dataset*)
Fit loop.

**_fit**(*x*)
Single fit iteration.

**Parameters** **x** (*list*) – the dataset.

**_fit_boosted**(*x*)
Single fit iteration for boosted FQI.

**Parameters** **x** (*list*) – the dataset.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.batch_td.**DoubleFQI**(*approximator*, *policy*, *mdp_info*, *n_iterations*, *fit_params=None*, *approximator_params=None*, *quiet=False*)
Bases: *mushroom.algorithms.value.batch_td.FQI*

Double Fitted Q-Iteration algorithm. "Estimating the Maximum Expected Value in Continuous Reinforcement Learning Problems". D'Eramo C. et al.. 2017.

**__init__**(*approximator*, *policy*, *mdp_info*, *n_iterations*, *fit_params=None*, *approximator_params=None*, *quiet=False*)
Constructor.

**Parameters**

- **n_iterations** (*int*) – number of iterations to perform for training;
- **quiet** (*bool, False*) – whether to show the progress bar or not;
- **boosted** (*bool, False*) – whether to use boosted FQI or not.

**_fit**(*x*)
Single fit iteration.

**Parameters** **x** (*list*) – the dataset.

**_fit_boosted**(*x*)
Single fit iteration for boosted FQI.

**Parameters** **x** (*list*) – the dataset.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit loop.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.batch_td.**LSPI**(*policy*, *mdp_info*, *epsilon=0.01*, *fit_params=None*, *approximator_params=None*, *features=None*)
Bases: *mushroom.algorithms.value.batch_td.BatchTD*

Least-Squares Policy Iteration algorithm. "Least-Squares Policy Iteration". Lagoudakis M. G. and Parr R.. 2003.

**__init__**(*policy*, *mdp_info*, *epsilon=0.01*, *fit_params=None*, *approximator_params=None*, *features=None*)
Constructor.

**Parameters**

- **approximator** (*object*) – approximator used by the algorithm and the policy.
- **fit_params** (*dict, None*) – parameters of the fitting algorithm of the approximator;
- **approximator_params** (*dict, None*) – parameters of the approximator to build;

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

### DQN

**class** mushroom.algorithms.value.dqn.**DQN**(*approximator*, *policy*, *mdp_info*, *batch_size*, *approximator_params*, *target_update_frequency*, *replay_memory=None*, *initial_replay_size=500*, *max_replay_size=5000*, *fit_params=None*, *n_approximators=1*, *clip_reward=True*)

Bases: *mushroom.algorithms.agent.Agent*

Deep Q-Network algorithm. "Human-Level Control Through Deep Reinforcement Learning". Mnih V. et al.. 2015.

**__init__**(*approximator*, *policy*, *mdp_info*, *batch_size*, *approximator_params*, *target_update_frequency*, *replay_memory=None*, *initial_replay_size=500*, *max_replay_size=5000*, *fit_params=None*, *n_approximators=1*, *clip_reward=True*)
Constructor.

**Parameters**

- **approximator** (*object*) – the approximator to use to fit the Q-function;
- **batch_size** (*int*) – the number of samples in a batch;
- **approximator_params** (*dict*) – parameters of the approximator to build;
- **target_update_frequency** (*int*) – the number of samples collected between each update of the target network;
- **replay_memory** (*[ReplayMemory, PrioritizedReplayMemory], None*) – the object of the replay memory to use; if None, a default replay memory is created;
- **initial_replay_size** (*int*) – the number of samples to collect before starting the learning;
- **max_replay_size** (*int*) – the maximum number of samples in the replay memory;
- **fit_params** (*dict, None*) – parameters of the fitting algorithm of the approximator;
- **n_approximators** (*int, 1*) – the number of approximator to use in AverageDQN;
- **clip_reward** (*bool, True*) – whether to clip the reward or not.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**_update_target**()
Update the target network.

**_next_q**(*next_state*, *absorbing*)

**Parameters**

- **next_state** (*np.ndarray*) – the states where next action has to be evaluated;

- **absorbing** (*np.ndarray*) – the absorbing flag for the states in `next_state`.

**Returns** Maximum action-value for each state in `next_state`.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.dqn.**DoubleDQN**(*approximator*, *policy*, *mdp_info*, *batch_size*, *approximator_params*, *target_update_frequency*, *replay_memory=None*, *initial_replay_size=500*, *max_replay_size=5000*, *fit_params=None*, *n_approximators=1*, *clip_reward=True*)
Bases: *mushroom.algorithms.value.dqn.DQN*

Double DQN algorithm. "Deep Reinforcement Learning with Double Q-Learning". Hasselt H. V. et al.. 2016.

**_next_q**(*next_state*, *absorbing*)

**Parameters**

- **next_state** (*np.ndarray*) – the states where next action has to be evaluated;
- **absorbing** (*np.ndarray*) – the absorbing flag for the states in `next_state`.

**Returns** Maximum action-value for each state in `next_state`.

**__init__**(*approximator*, *policy*, *mdp_info*, *batch_size*, *approximator_params*, *target_update_frequency*, *replay_memory=None*, *initial_replay_size=500*, *max_replay_size=5000*, *fit_params=None*, *n_approximators=1*, *clip_reward=True*)
Constructor.

**Parameters**

- **approximator** (*object*) – the approximator to use to fit the Q-function;
- **batch_size** (*int*) – the number of samples in a batch;
- **approximator_params** (*dict*) – parameters of the approximator to build;
- **target_update_frequency** (*int*) – the number of samples collected between each update of the target network;
- **replay_memory** (*[ReplayMemory, PrioritizedReplayMemory], None*) – the object of the replay memory to use; if None, a default replay memory is created;
- **initial_replay_size** (*int*) – the number of samples to collect before starting the learning;
- **max_replay_size** (*int*) – the maximum number of samples in the replay memory;
- **fit_params** (*dict, None*) – parameters of the fitting algorithm of the approximator;

- **n_approximators** (*int, 1*) – the number of approximator to use in AverageDQN;
- **clip_reward** (*bool, True*) – whether to clip the reward or not.

**_update_target**()
Update the target network.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.dqn.**AveragedDQN**(*approximator*, *policy*, *mdp_info*, ***params*)
Bases: *mushroom.algorithms.value.dqn.DQN*

Averaged-DQN algorithm. "Averaged-DQN: Variance Reduction and Stabilization for Deep Reinforcement Learning". Anschel O. et al.. 2017.

**__init__**(*approximator*, *policy*, *mdp_info*, ***params*)
Constructor.

**Parameters**

- **approximator** (*object*) – the approximator to use to fit the Q-function;
- **batch_size** (*int*) – the number of samples in a batch;
- **approximator_params** (*dict*) – parameters of the approximator to build;
- **target_update_frequency** (*int*) – the number of samples collected between each update of the target network;
- **replay_memory** (*[ReplayMemory, PrioritizedReplayMemory], None*) – the object of the replay memory to use; if None, a default replay memory is created;
- **initial_replay_size** (*int*) – the number of samples to collect before starting the learning;
- **max_replay_size** (*int*) – the maximum number of samples in the replay memory;
- **fit_params** (*dict, None*) – parameters of the fitting algorithm of the approximator;
- **n_approximators** (*int, 1*) – the number of approximator to use in AverageDQN;
- **clip_reward** (*bool, True*) – whether to clip the reward or not.

**_update_target**()
Update the target network.

**_next_q**(*next_state*, *absorbing*)

**Parameters**

- **next_state** (*np.ndarray*) – the states where next action has to be evaluated;
- **absorbing** (*np.ndarray*) – the absorbing flag for the states in `next_state`.

**Returns** Maximum action-value for each state in `next_state`.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.value.dqn.**CategoricalNetwork**(*input_shape*, *output_shape*, *features_network*, *n_atoms*, *v_min*, *v_max*, *n_features*, *use_cuda*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

**__init__**(*input_shape*, *output_shape*, *features_network*, *n_atoms*, *v_min*, *v_max*, *n_features*, *use_cuda*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

**__call__**(**args*, ***kw*)
Call self as a function.

**class** mushroom.algorithms.value.dqn.**CategoricalDQN**(*policy*, *mdp_info*, *n_atoms*, *v_min*, *v_max*, *approximator_params*, ***params*)
Bases: *mushroom.algorithms.value.dqn.DQN*

Categorical DQN algorithm. "A Distributional Perspective on Reinforcement Learning". Bellemare M. et al.. 2017.

**__init__**(*policy*, *mdp_info*, *n_atoms*, *v_min*, *v_max*, *approximator_params*, ***params*)
Constructor.

**Parameters**

- **n_atoms** (*int*) – number of atoms;
- **v_min** (*float*) – minimum value of value-function;
- **v_max** (*float*) – maximum value of value-function.

**_next_q**(*next_state*, *absorbing*)

**Parameters**

- **next_state** (*np.ndarray*) – the states where next action has to be evaluated;

- **absorbing** (*np.ndarray*) – the absorbing flag for the states in `next_state`.

**Returns** Maximum action-value for each state in `next_state`.

**_update_target**()
Update the target network.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

### Policy search

### Policy gradient

**class** mushroom.algorithms.policy_search.policy_gradient.**PolicyGradient**(*policy*, *mdp_info*, *learning_rate*, *features*)

Bases: *mushroom.algorithms.agent.Agent*

Abstract class to implement a generic Policy Search algorithm using the gradient of the policy to update its parameters. "A survey on Policy Gradient algorithms for Robotics". Deisenroth M. P. et al.. 2011.

**__init__**(*policy*, *mdp_info*, *learning_rate*, *features*)
Constructor.

**Parameters** **learning_rate** (*float*) – the learning rate.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**_update_parameters**(*J*)
Update the parameters of the policy.

**Parameters** **J** (*list*) – list of the cumulative discounted rewards for each episode in the dataset.

**_init_update**()
This function is called, when parsing the dataset, at the beginning of each episode. The implementation is dependent on the algorithm (e.g. REINFORCE resets some data structure).

**_step_update**(*x*, *u*, *r*)
This function is called, when parsing the dataset, at each episode step.

**Parameters**

- **x** (*np.ndarray*) – the state at the current step;
- **u** (*np.ndarray*) – the action at the current step;
- **r** (*np.ndarray*) – the reward at the current step.

**_episode_end_update**()
This function is called, when parsing the dataset, at the beginning of each episode. The implementation is dependent on the algorithm (e.g. REINFORCE updates some data structures).

**_compute_gradient**(*J*)
Return the gradient computed by the algorithm.

**Parameters** **J** (*list*) – list of the cumulative discounted rewards for each episode in the dataset.

**_parse**(*sample*)
Utility to parse the sample.

**Parameters** **sample** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag. If provided, `state` is preprocessed with the features.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.policy_search.policy_gradient.**REINFORCE**(*policy*, *mdp_info*, *learning_rate*, *features=None*)
Bases: *mushroom.algorithms.policy_search.policy_gradient.PolicyGradient*

REINFORCE algorithm. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning", Williams R. J.. 1992.

**__init__**(*policy*, *mdp_info*, *learning_rate*, *features=None*)
Constructor.

**Parameters** **learning_rate** (*float*) – the learning rate.

**_compute_gradient**(*J*)
Return the gradient computed by the algorithm.

**Parameters** **J** (*list*) – list of the cumulative discounted rewards for each episode in the dataset.

**_step_update**(*x*, *u*, *r*)
This function is called, when parsing the dataset, at each episode step.

**Parameters**

- **x** (*np.ndarray*) – the state at the current step;
- **u** (*np.ndarray*) – the action at the current step;
- **r** (*np.ndarray*) – the reward at the current step.

**_episode_end_update**()
This function is called, when parsing the dataset, at the beginning of each episode. The implementation is dependent on the algorithm (e.g. REINFORCE updates some data structures).

**_init_update**()
This function is called, when parsing the dataset, at the beginning of each episode. The implementation is dependent on the algorithm (e.g. REINFORCE resets some data structure).

**_parse**(*sample*)
Utility to parse the sample.

**Parameters** **sample** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag. If provided, `state` is preprocessed with the features.

**_update_parameters**(*J*)
Update the parameters of the policy.

**Parameters** **J** (*list*) – list of the cumulative discounted rewards for each episode in the dataset.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.policy_search.policy_gradient.**GPOMDP**(*policy*, *mdp_info*, *learning_rate*, *features=None*)
Bases: *mushroom.algorithms.policy_search.policy_gradient.PolicyGradient*

GPOMDP algorithm. "Infinite-Horizon Policy-Gradient Estimation". Baxter J. and Bartlett P. L.. 2001.

**__init__**(*policy*, *mdp_info*, *learning_rate*, *features=None*)
Constructor.

**Parameters** **learning_rate** (*float*) – the learning rate.

**_compute_gradient**(*J*)
Return the gradient computed by the algorithm.

**Parameters** **J** (*list*) – list of the cumulative discounted rewards for each episode in the dataset.

**_step_update**(*x*, *u*, *r*)
This function is called, when parsing the dataset, at each episode step.

**Parameters**

- **x** (*np.ndarray*) – the state at the current step;
- **u** (*np.ndarray*) – the action at the current step;
- **r** (*np.ndarray*) – the reward at the current step.

**_episode_end_update**()
This function is called, when parsing the dataset, at the beginning of each episode. The implementation is dependent on the algorithm (e.g. REINFORCE updates some data structures).

**_init_update**()
This function is called, when parsing the dataset, at the beginning of each episode. The implementation is dependent on the algorithm (e.g. REINFORCE resets some data structure).

**_parse**(*sample*)
Utility to parse the sample.

**Parameters** **sample** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag. If provided, `state` is preprocessed with the features.

**_update_parameters**(*J*)
Update the parameters of the policy.

**Parameters** **J** (*list*) – list of the cumulative discounted rewards for each episode in the dataset.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.policy_search.policy_gradient.**eNAC**(*policy*, *mdp_info*, *learning_rate*, *features=None*, *critic_features=None*)
Bases: *mushroom.algorithms.policy_search.policy_gradient.PolicyGradient*

Episodic Natural Actor Critic algorithm. “A Survey on Policy Search for Robotics”, Deisenroth M. P., Neumann G., Peters J. 2013.

**__init__**(*policy*, *mdp_info*, *learning_rate*, *features=None*, *critic_features=None*)
Constructor.

**Parameters** **critic_features** (*Features, None*) – features used by the critic.

**_compute_gradient**(*J*)
Return the gradient computed by the algorithm.

**Parameters** **J** (*list*) – list of the cumulative discounted rewards for each episode in the dataset.

**_step_update**(*x*, *u*, *r*)
This function is called, when parsing the dataset, at each episode step.

**Parameters**

- **x** (*np.ndarray*) – the state at the current step;
- **u** (*np.ndarray*) – the action at the current step;
- **r** (*np.ndarray*) – the reward at the current step.

**_episode_end_update**()
This function is called, when parsing the dataset, at the beginning of each episode. The implementation is dependent on the algorithm (e.g. REINFORCE updates some data structures).

**_init_update**()
This function is called, when parsing the dataset, at the beginning of each episode. The implementation is dependent on the algorithm (e.g. REINFORCE resets some data structure).

**_parse**(*sample*)
Utility to parse the sample.

**Parameters** **sample** (*list*) – the current episode step.

**Returns** A tuple containing state, action, reward, next state, absorbing and last flag. If provided, `state` is preprocessed with the features.

**_update_parameters**(*J*)
Update the parameters of the policy.

**Parameters** **J** (*list*) – list of the cumulative discounted rewards for each episode in the dataset.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

### Black-Box optimization

**class** mushroom.algorithms.policy_search.black_box_optimization.**BlackBoxOptimization**(*distributi*
*pol-*
*icy*,
*mdp_info*
*fea-*
*tures=No*

Bases: *mushroom.algorithms.agent.Agent*

Base class for black box optimization algorithms. These algorithms work on a distribution of policy parameters and often they do not rely on stochastic and differentiable policies.

**__init__**(*distribution*, *policy*, *mdp_info*, *features=None*)
Constructor.

**Parameters**

- **distribution** (Distribution) – the distribution of policy parameters;
- **policy** (ParametricPolicy) – the policy to use.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**_update**(*Jep*, *theta*)
Function that implements the update routine of distribution parameters. Every black box algorithms should implement this function with the proper update.

**Parameters**

- **Jep** (*np.ndarray*) – a vector containing the J of the considered trajectories;
- **theta** (*np.ndarray*) – a matrix of policy parameters of the considered trajectories.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**class** mushroom.algorithms.policy_search.black_box_optimization.**RWR**(*distribution*, *policy*, *mdp_info*, *beta*, *features=None*)

Bases: *mushroom.algorithms.policy_search.black_box_optimization.BlackBoxOptimization*

Reward-Weighted Regression algorithm. “A Survey on Policy Search for Robotics”, Deisenroth M. P., Neumann G., Peters J.. 2013.

**__init__**(*distribution*, *policy*, *mdp_info*, *beta*, *features=None*)
Constructor.

**Parameters** **beta** (*float*) – the temperature for the exponential reward transformation.

**_update**(*Jep*, *theta*)
Function that implements the update routine of distribution parameters. Every black box algorithms should implement this function with the proper update.

**Parameters**

- **Jep** (*np.ndarray*) – a vector containing the J of the considered trajectories;
- **theta** (*np.ndarray*) – a matrix of policy parameters of the considered trajectories.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.policy_search.black_box_optimization.**PGPE**(*distribution*, *policy*, *mdp_info*, *learning_rate*, *features=None*)
Bases: *mushroom.algorithms.policy_search.black_box_optimization.BlackBoxOptimization*

Policy Gradient with Parameter Exploration algorithm. "A Survey on Policy Search for Robotics", Deisenroth M. P., Neumann G., Peters J.. 2013.

**__init__**(*distribution*, *policy*, *mdp_info*, *learning_rate*, *features=None*)
Constructor.

**Parameters** **learning_rate** (*Parameter*) – the learning rate for the gradient step.

**_update**(*Jep*, *theta*)
Function that implements the update routine of distribution parameters. Every black box algorithms should implement this function with the proper update.

**Parameters**

- **Jep** (*np.ndarray*) – a vector containing the J of the considered trajectories;
- **theta** (*np.ndarray*) – a matrix of policy parameters of the considered trajectories.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.policy_search.black_box_optimization.**REPS**(*distribution*, *policy*, *mdp_info*, *eps*, *features=None*)
Bases: *mushroom.algorithms.policy_search.black_box_optimization.BlackBoxOptimization*

Episodic Relative Entropy Policy Search algorithm. "A Survey on Policy Search for Robotics", Deisenroth M. P., Neumann G., Peters J.. 2013.

**__init__**(*distribution*, *policy*, *mdp_info*, *eps*, *features=None*)
Constructor.

**Parameters** **eps** (*float*) – the maximum admissible value for the Kullback-Leibler divergence between the new distribution and the previous one at each update step.

**_update**(*Jep*, *theta*)
Function that implements the update routine of distribution parameters. Every black box algorithms should implement this function with the proper update.

**Parameters**

- **Jep** (*np.ndarray*) – a vector containing the J of the considered trajectories;
- **theta** (*np.ndarray*) – a matrix of policy parameters of the considered trajectories.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

### Actor-Critic

#### Deterministic Policy Gradient

**class** mushroom.algorithms.actor_critic.dpg.**COPDAC_Q**(*policy*, *mu*, *mdp_info*, *alpha_theta*, *alpha_omega*, *alpha_v*, *value_function_features=None*, *policy_features=None*)
Bases: *mushroom.algorithms.agent.Agent*

Compatible off-policy deterministic actor-critic algorithm. "Deterministic Policy Gradient Algorithms". Silver D. et al.. 2014.

**__init__**(*policy*, *mu*, *mdp_info*, *alpha_theta*, *alpha_omega*, *alpha_v*, *value_function_features=None*, *policy_features=None*)
Constructor.

**Parameters**

- **policy** (Policy) – the policy followed by the agent;
- **mdp_info** (MDPInfo) – information about the MDP;
- **features** (*object, None*) – features to extract from the state.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

#### Deep Deterministic Policy Gradient

**class** mushroom.algorithms.actor_critic.ddpg.**ActorLoss**(*critic*)
Bases: sphinx.ext.autodoc.importer._MockObject

Class used to implement the loss function of the actor.

**__init__**(*critic*)
Initialize self. See help(type(self)) for accurate signature.

**__call__**(**args*, ***kw*)
Call self as a function.

**class** mushroom.algorithms.actor_critic.ddpg.**ActorLossTD3**(*critic*)
Bases: sphinx.ext.autodoc.importer._MockObject

Class used to implement the loss function of the actor.

**__init__**(*critic*)
Initialize self. See help(type(self)) for accurate signature.

**__call__**(**args*, ***kw*)
Call self as a function.

**class** mushroom.algorithms.actor_critic.ddpg.**DDPG**(*actor_approximator*, *critic_approximator*, *policy_class*, *mdp_info*, *batch_size*, *initial_replay_size*, *max_replay_size*, *tau*, *actor_params*, *critic_params*, *policy_params*, *policy_delay=1*, *actor_fit_params=None*, *critic_fit_params=None*)
Bases: *mushroom.algorithms.agent.Agent*

Deep Deterministic Policy Gradient algorithm. "Continuous Control with Deep Reinforcement Learning". Lillicrap T. P. et al.. 2016.

**__init__**(*actor_approximator*, *critic_approximator*, *policy_class*, *mdp_info*, *batch_size*, *initial_replay_size*, *max_replay_size*, *tau*, *actor_params*, *critic_params*, *policy_params*, *policy_delay=1*, *actor_fit_params=None*, *critic_fit_params=None*)
Constructor.

**Parameters**

- **actor_approximator** (*object*) – the approximator to use for the actor;
- **critic_approximator** (*object*) – the approximator to use for the critic;
- **policy_class** (Policy) – class of the policy;
- **batch_size** (*int*) – the number of samples in a batch;
- **initial_replay_size** (*int*) – the number of samples to collect before starting the learning;
- **max_replay_size** (*int*) – the maximum number of samples in the replay memory;
- **tau** (*float*) – value of coefficient for soft updates;
- **actor_params** (*dict*) – parameters of the actor approximator to build;
- **critic_params** (*dict*) – parameters of the critic approximator to build;
- **policy_params** (*dict*) – parameters of the policy to build;
- **policy_delay** (*int, 1*) – the number of updates of the critic after which an actor update is implemented;
- **actor_fit_params** (*dict, None*) – parameters of the fitting algorithm of the actor approximator;
- **critic_fit_params** (*dict, None*) – parameters of the fitting algorithm of the critic approximator;

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**_init_target**()
Init weights for target approximators

**_update_target**()
Update the target networks.

**_next_q**(*next_state*, *absorbing*)

**Parameters**

- **next_state** (*np.ndarray*) – the states where next action has to be evaluated;
- **absorbing** (*np.ndarray*) – the absorbing flag for the states in `next_state`.

**Returns** Action-values returned by the critic for `next_state` and the action returned by the actor.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.actor_critic.ddpg.**TD3**(*actor_approximator*, *critic_approximator*, *policy_class*, *mdp_info*, *batch_size*, *initial_replay_size*, *max_replay_size*, *tau*, *actor_params*, *critic_params*, *policy_params*, *policy_delay=2*, *noise_std=0.2*, *noise_clip=0.5*, *actor_fit_params=None*, *critic_fit_params=None*)

Bases: *mushroom.algorithms.actor_critic.ddpg.DDPG*

Twin Delayed DDPG algorithm. "Addressing Function Approximation Error in Actor-Critic Methods". Fujimoto S. et al.. 2018.

**__init__**(*actor_approximator*, *critic_approximator*, *policy_class*, *mdp_info*, *batch_size*, *initial_replay_size*, *max_replay_size*, *tau*, *actor_params*, *critic_params*, *policy_params*, *policy_delay=2*, *noise_std=0.2*, *noise_clip=0.5*, *actor_fit_params=None*, *critic_fit_params=None*)
Constructor.

**Parameters**

- **actor_approximator** (*object*) – the approximator to use for the actor;
- **critic_approximator** (*object*) – the approximator to use for the critic;
- **policy_class** (*Policy*) – class of the policy;

- **batch_size** (*int*) – the number of samples in a batch;
- **initial_replay_size** (*int*) – the number of samples to collect before starting the learning;
- **max_replay_size** (*int*) – the maximum number of samples in the replay memory;
- **tau** (*float*) – value of coefficient for soft updates;
- **actor_params** (*dict*) – parameters of the actor approximator to build;
- **critic_params** (*dict*) – parameters of the critic approximator to build;
- **policy_params** (*dict*) – parameters of the policy to build;
- **policy_delay** (*int, 2*) – the number of updates of the critic after which an actor update is implemented;
- **noise_std** (*float, 0.2*) – standard deviation of the noise used for policy smoothing;
- **noise_clip** (*float, 0.5*) – maximum absolute value for policy smoothing noise;
- **actor_fit_params** (*dict, None*) – parameters of the fitting algorithm of the actor approximator;
- **critic_fit_params** (*dict, None*) – parameters of the fitting algorithm of the critic approximator;

**_init_target**()
Init weights for target approximators

**_update_target**()
Update the target networks.

**_next_q**(*next_state*, *absorbing*)

**Parameters**

- **next_state** (*np.ndarray*) – the states where next action has to be evaluated;
- **absorbing** (*np.ndarray*) – the absorbing flag for the states in `next_state`.

**Returns** Action-values returned by the critic for `next_state` and the action returned by the actor.

**draw_action**(*state*)
Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**episode_start**()
Called by the agent when a new episode starts.

**fit**(*dataset*)
Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**stop**()
Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

### Stochastic Actor-Critic

**class** mushroom.algorithms.actor_critic.stochastic_actor_critic.**SAC**(*policy*, *mdp_info*, *alpha_theta*, *alpha_v*, *lambda_par=0.9*, *value_function_features=None*, *policy_features=None*)

Bases: *mushroom.algorithms.agent.Agent*

Stochastic Actor critic in the episodic setting as presented in: "Model-Free Reinforcement Learning with Continuous Action in Practice". Degris T. et al.. 2012.

**__init__**(*policy*, *mdp_info*, *alpha_theta*, *alpha_v*, *lambda_par=0.9*, *value_function_features=None*, *policy_features=None*)

Constructor.

**Parameters**

- **policy** (ParametricPolicy) – a differentiable stochastic policy;
- **mdp_info** – information about the MDP;
- **alpha_theta** (Parameter) – learning rate for policy update;
- **alpha_v** (Parameter) – learning rate for the value function;
- **lambda_par** (*float, 0.9*) – trace decay parameter;
- **value_function_features** (*Features, None*) – features used by the value function approximator;
- **policy_features** (*Features, None*) – features used by the policy.

**episode_start**()

Called by the agent when a new episode starts.

**fit**(*dataset*)

Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**draw_action**(*state*)

Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**stop**()

Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

**class** mushroom.algorithms.actor_critic.stochastic_actor_critic.**SAC_AVG**(*policy*, *mdp_info*, *alpha_theta*, *alpha_v*, *alpha_r*, *lambda_par=0.9*, *value_function_features=No*
*policy_features=None*)

Bases: *mushroom.algorithms.agent.Agent*

Stochastic Actor critic in the average reward setting as presented in: "Model-Free Reinforcement Learning with Continuous Action in Practice". Degris T. et al.. 2012.

**__init__**(*policy*, *mdp_info*, *alpha_theta*, *alpha_v*, *alpha_r*, *lambda_par=0.9*, *value_function_features=None*, *policy_features=None*)

Constructor.

**Parameters**

- **policy** (ParametricPolicy) – a differentiable stochastic policy;
- **mdp_info** – information about the MDP;
- **alpha_theta** (Parameter) – learning rate for policy update;
- **alpha_v** (Parameter) – learning rate for the value function;
- **alpha_r** (Parameter) – learning rate for the reward trace;
- **lambda_par** (*float, 0.9*) – trace decay parameter;
- **value_function_features** (*Features, None*) – features used by the value function approximator;
- **policy_features** (*Features, None*) – features used by the policy.

**episode_start**()

Called by the agent when a new episode starts.

**fit**(*dataset*)

Fit step.

**Parameters** **dataset** (*list*) – the dataset.

**draw_action**(*state*)

Return the action to execute in the given state. It is the action returned by the policy or the action set by the algorithm (e.g. in the case of SARSA).

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action to be executed.

**stop**()

Method used to stop an agent. Useful when dealing with real world environments, simulators, or to cleanup environments internals after a core learn/evaluate to enforce consistency.

### 3.1.4 Approximators

Mushroom exposes the high-level class `Regressor` that can manage any type of function regressor. This class is a wrapper for any kind of function approximator, e.g. a scikit-learn approximator or a pytorch neural network.

#### Regressor

**class** mushroom.approximators.regressor.**Regressor**(*approximator*, *input_shape*, *output_shape=(1, )*, *n_actions=None*, *n_models=1*, ***params*)

Bases: `object`

This class implements the function to manage a function approximator. This class selects the appropriate kind of regressor to implement according to the parameters provided by the user; this makes this class the only one to use for each kind of task that has to be performed. The inference of the implementation to choose is done checking the provided values of parameters `n_actions`. If `n_actions` is provided, it means that the user wants to implement an approximator of the Q-function: if the value of `n_actions` is equal to the `output_shape` then a `QRegressor` is created, else (`output_shape` should be (1,)) an `ActionRegressor` is created. Else a `GenericRegressor` is created. An `Ensemble` model can be used for all the previous implementations listed before simply providing a `n_models` parameter greater than 1.

**__init__**(*approximator*, *input_shape*, *output_shape=(1, )*, *n_actions=None*, *n_models=1*, ***params*)
Constructor.

**Parameters**

- **approximator** (*object*) – the approximator class to use to create the model;
- **input_shape** (*tuple*) – the shape of the input of the model;
- **output_shape** (*tuple, (1,)*) – the shape of the output of the model;
- **n_actions** (*int, None*) – number of actions considered to create a `QRegressor` or an `ActionRegressor`;
- **n_models** (*int, 1*) – number of models to create;
- ****params** (*dict*) – other parameters to create each model.

**__call__**(**z*, ***predict_params*)
Call self as a function.

**fit**(**z*, ***fit_params*)
Fit the model.

**Parameters**

- ***z** (*list*) – list of input of the model;
- ****fit_params** (*dict*) – parameters to use to fit the model.

**predict**(**z*, ***predict_params*)
Predict the output of the model given an input.

**Parameters**

- ***z** (*list*) – list of input of the model;
- ****predict_params** (*dict*) – parameters to use to predict with the model.

**Returns** The model prediction.

**model**
The model object.

**Type** Returns

**reset**()
Reset the model parameters.

**input_shape**
The shape of the input of the model.

**Type** Returns

**output_shape**
The shape of the output of the model.

**Type** Returns

**weights_size**
The shape of the weights of the model.

**Type** Returns

**get_weights**()

**Returns** The weights of the model.

**set_weights**(*w*)

**Parameters** **w** (*list*) – list of weights to be set in the model.

**diff**(**z*)

**Parameters** ***z** (*list*) – the input of the model.

**Returns** The derivative of the model.

### Approximator

### Linear

**class** mushroom.approximators.parametric.linear.**LinearApproximator**(*weights=None*, *input_shape=None*, *output_shape=1*, ***kwargs*)

Bases: object

This class implements a linear approximator.

**__init__**(*weights=None*, *input_shape=None*, *output_shape=1*, ***kwargs*)
Constructor.

**Parameters**

- **weights** (*np.ndarray*) – array of weights to initialize the weights of the approximator;
- **input_shape** (*np.ndarray*) – the shape of the input of the model;
- **output_shape** (*np.ndarray*) – the shape of the output of the model;
- ****kwargs** (*dict*) – other params of the approximator.

**fit** (*x*, *y*, ***fit_params*)
Fit the model.

**Parameters**

- **x** (*np.ndarray*) – input;
- **y** (*np.ndarray*) – target;
- ****fit_params** (*dict*) – other parameters used by the fit method of the regressor.

**predict** (*x*, ***predict_params*)
Predict.

**Parameters**

- **x** (*np.ndarray*) – input;
- ****predict_params** (*dict*) – other parameters used by the predict method the regressor.

**Returns** The predictions of the model.

### Pytorch Neural Network

**class** mushroom.approximators.parametric.pytorch_network.**PyTorchApproximator**(*input_shape*, *output_shape*, *network*, *optimizer=None*, *loss=None*, *batch_size=0*, *n_fit_targets=1*, *use_cuda=False*, *reinitialize=False*, *dropout=False*, *quiet=True*, ***params*)

Bases: object

Class to interface a pytorch model to the mushroom Regressor interface. This class implements all is needed to use a generic pytorch model and train it using a specified optimizer and objective function. This class supports also minibatches.

**__init__** (*input_shape*, *output_shape*, *network*, *optimizer=None*, *loss=None*, *batch_size=0*, *n_fit_targets=1*, *use_cuda=False*, *reinitialize=False*, *dropout=False*, *quiet=True*, ***params*)
Constructor.

**Parameters**

- **input_shape** (*tuple*) – shape of the input of the network;
- **output_shape** (*tuple*) – shape of the output of the network;
- **network** (*torch.nn.Module*) – the network class to use;

- **optimizer** (*dict*) – the optimizer used for every fit step;
- **loss** (*torch.nn.functional*) – the loss function to optimize in the fit method;
- **batch_size** (*int, 0*) – the size of each minibatch. If 0, the whole dataset is fed to the optimizer at each epoch;
- **n_fit_targets** (*int, 1*) – the number of fit targets used by the fit method of the network;
- **use_cuda** (*bool, False*) – if True, runs the network on the GPU;
- **reinitialize** (*bool, False*) – if True, the approximator is re
- **at every fit call. To perform the initialization, the** (*initialized*) –
- **method must be defined properly for the selected** (*weights_init*) –
- **network.** (*model*) –
- **dropout** (*bool, False*) – if True, dropout is applied only during train;
- **quiet** (*bool, True*) – if False, shows two progress bars, one for epochs and one for the minibatches;
- **params** (*dict*) – dictionary of parameters needed to construct the network.

### 3.1.5 Features

The features in Mushroom are 1-D arrays computed applying a specified function to a raw input, e.g. polynomial features of the state of an MDP. Mushroom supports three types of features:

- basis functions;
- tensor basis functions;
- tiles.

The GPU-accelerated basis functions are a Pytorch implementation of the standard basis functions. They are less straightforward than the standard ones, but they are faster to compute as they can exploit parallel computing, e.g. GPU-acceleration and multi-core systems.

All the types of features are exposed by a single factory method `Features` that builds the one requested by the user.

`mushroom.features.features.`**Features**(*basis_list=None*, *tilings=None*, *tensor_list=None*, *device=None*)

Factory method to build the requested type of features. The types are mutually exclusive.

The difference between `basis_list` and `tensor_list` is that the former is a list of python classes each one evaluating a single element of the feature vector, while the latter consists in a list of PyTorch modules that can be used to build a PyTorch network. The use of `tensor_list` is a faster way to compute features than *basis_list* and is suggested when the computation of the requested features is slow (see the Gaussian radial basis function implementation as an example).

**Parameters**

- **basis_list** (*list, None*) – list of basis functions;
- **tilings** (*[object, list], None*) – single object or list of tilings;
- **tensor_list** (*list, None*) – list of dictionaries containing the instructions to build the requested tensors;

- **device** (*int, None*) – where to run the group of tensors. Only needed when using a list of tensors;

**Returns** The class implementing the requested type of features.

mushroom.features.features.**get_action_features**(*phi_state*, *action*, *n_actions*)
Compute an array of size `len(phi_state) * n_actions` filled with zeros, except for elements from `len(phi_state) * action` to `len(phi_state) * (action + 1)` that are filled with *phi_state*. This is used to compute state-action features.

**Parameters**

- **phi_state** (*np.ndarray*) – the feature of the state;
- **action** (*np.ndarray*) – the action whose features have to be computed;
- **n_actions** (*int*) – the number of actions.

**Returns** The state-action features.

The factory method returns a class that extends the abstract class `FeatureImplementation`.

### Components

### Basis

### Fourier

**class** mushroom.features.basis.fourier.**FourierBasis**(*low*, *delta*, *c*, *dimensions=None*)
Bases: `object`

Class implementing Fourier basis functions. The value of the feature is computed using the formula:

$$\sum \cos \pi (X - m)/\Delta c$$

where X is the input, m is the vector of the minumum input values (for each dimensions) , Delta is the vector of maximum

**__init__**(*low*, *delta*, *c*, *dimensions=None*)
Constructor.

**Parameters**

- **low** (*np.ndarray*) – vector of minimum values of the input variables;
- **delta** (*np.ndarray*) – vector of the maximum difference between two values of the input variables, i.e. delta = high - low;
- **c** (*np.ndarray*) – vector of weights for the state variables;
- **dimensions** (*list, None*) – list of the dimensions of the input to be considered by the feature.

**__call__**(*x*)
Call self as a function.

**static generate**(*low*, *high*, *n*, *dimensions=None*)
Factory method to build a set of fourier basis.

**Parameters**

- **low** (*np.ndarray*) – vector of minimum values of the input variables;

- **high** (*np.ndarray*) – vector of maximum values of the input variables;
- **n** (*int*) – number of harmonics to consider for each state variable
- **dimensions** (*list, None*) – list of the dimensions of the input to be considered by the features.

**Returns** The list of the generated fourier basis functions.

### Gaussian RBF

**class** mushroom.features.basis.gaussian_rbf.**GaussianRBF**(*mean*, *scale*, *dimensions=None*)

Bases: object

Class implementing Gaussian radial basis functions. The value of the feature is computed using the formula:

$$\sum \frac{(X_i - \mu_i)^2}{\sigma_i}$$

where X is the input, mu is the mean vector and sigma is the scale parameter vector.

**__init__**(*mean*, *scale*, *dimensions=None*)
Constructor.

**Parameters**

- **mean** (*np.ndarray*) – the mean vector of the feature;
- **scale** (*np.ndarray*) – the scale vector of the feature;
- **dimensions** (*list, None*) – list of the dimensions of the input to be considered by the feature. The number of dimensions must match the dimensionality of `mean` and `scale`.

**__call__**(*x*)
Call self as a function.

**static generate**(*n_centers*, *low*, *high*, *dimensions=None*)
Factory method to build uniformly spaced gaussian radial basis functions with a 25% overlap.

**Parameters**

- **n_centers** (*list*) – list of the number of radial basis functions to be used for each dimension.
- **low** (*np.ndarray*) – lowest value for each dimension;
- **high** (*np.ndarray*) – highest value for each dimension;
- **dimensions** (*list, None*) – list of the dimensions of the input to be considered by the feature. The number of dimensions must match the number of elements in `n_centers` and `low`.

**Returns** The list of the generated radial basis functions.

### Polynomial

**class** mushroom.features.basis.polynomial.**PolynomialBasis**(*dimensions=None*, *degrees=None*)

Bases: object

Class implementing polynomial basis functions. The value of the feature is computed using the formula:

$$\prod X_i^{d_i}$$

where X is the input and d is the vector of the exponents of the polynomial.

**__init__**(*dimensions=None*, *degrees=None*)
Constructor. If both parameters are None, the constant feature is built.

**Parameters**

- **dimensions** (*list, None*) – list of the dimensions of the input to be considered by the feature;
- **degrees** (*list, None*) – list of the degrees of each dimension to be considered by the feature. It must match the number of elements of `dimensions`.

**__call__**(*x*)
Call self as a function.

**static _compute_exponents**(*order*, *n_variables*)
Find the exponents of a multivariate polynomial expression of order `order` and `n_variables` number of variables.

**Parameters**

- **order** (*int*) – the maximum order of the polynomial;
- **n_variables** (*int*) – the number of elements of the input vector.

**Yields** The current exponent of the polynomial.

**static generate**(*max_degree*, *input_size*)
Factory method to build a polynomial of order `max_degree` based on the first `input_size` dimensions of the input.

**Parameters**

- **max_degree** (*int*) – maximum degree of the polynomial;
- **input_size** (*int*) – size of the input.

**Returns** The list of the generated polynomial basis functions.

### Tensors

### Gaussian tensor

**class** mushroom.features.tensors.gaussian_tensor.**PyTorchGaussianRBF**(*mu*, *scale*, *dim*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Pytorch module to implement a gaussian radial basis function.

**__init__**(*mu*, *scale*, *dim*)
Initialize self. See help(type(self)) for accurate signature.

**static generate**(*n_centers*, *low*, *high*, *dimensions=None*)
Factory method that generates the list of dictionaries to build the tensors representing a set of uniformly spaced Gaussian radial basis functions with a 25% overlap.

**Parameters**

- **n_centers** (*list*) – list of the number of radial basis functions to be used for each dimension;
- **low** (*np.ndarray*) – lowest value for each dimension;
- **high** (*np.ndarray*) – highest value for each dimension;
- **dimensions** (*list, None*) – list of the dimensions of the input to be considered by the feature. The number of dimensions must match the number of elements in `n_centers` and `low`.

**Returns** The list of dictionaries as described above.

#### Tiles

**class** mushroom.features.tiles.tiles.**Tiles**(*x_range*, *n_tiles*, *state_components=None*)
Bases: `object`

Class implementing rectangular tiling. For each point in the state space, this class can be used to compute the index of the corresponding tile.

**__init__**(*x_range*, *n_tiles*, *state_components=None*)
Constructor.

**Parameters**

- **x_range** (*list*) – list of two-elements lists specifying the range of each state variable;
- **n_tiles** (*list*) – list of the number of tiles to be used for each dimension.
- **state_components** (*list, None*) – list of the dimensions of the input to be considered by the tiling. The number of elements must match the number of elements in `x_range` and `n_tiles`.

**__call__**(*x*)
Call self as a function.

**static generate**(*n_tilings*, *n_tiles*, *low*, *high*, *uniform=False*)
Factory method to build `n_tilings` tilings of `n_tiles` tiles with a range between `low` and `high` for each dimension.

**Parameters**

- **n_tilings** (*int*) – number of tilings;
- **n_tiles** (*list*) – number of tiles for each tilings for each dimension;
- **low** (*np.ndarray*) – lowest value for each dimension;
- **high** (*np.ndarray*) – highest value for each dimension.
- **uniform** (*bool, False*) – if True the displacement for each tiling will be w/n_tilings, where w is the tile width. Otherwise, the displacement will be k*w/n_tilings, where k=2i+1, where i is the dimension index.

**Returns** The list of the generated tiles.

### 3.1.6 Policy

**class** mushroom.policy.policy.**Policy**
Bases: `object`

Interface representing a generic policy. A policy is a probability distribution that gives the probability of taking an action given a specified state. A policy is used by mushroom agents to interact with the environment.

**__call__**(**args*)
Compute the probability of taking action in a certain state following the policy.

**Parameters** ***args** (*list*) – list containing a state or a state and an action.

**Returns** The probability of all actions following the policy in the given state if the list contains only the state, else the probability of the given action in the given state following the policy. If the action space is continuous, state and action must be provided

**draw_action**(*state*)
Sample an action in `state` using the policy.

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action sampled from the policy.

**reset**()
Useful when the policy needs a special initialization at the beginning of an episode.

**__init__**
Initialize self. See help(type(self)) for accurate signature.

**class** mushroom.policy.policy.**ParametricPolicy**
Bases: *mushroom.policy.policy.Policy*

Interface for a generic parametric policy. A parametric policy is a policy that depends on set of parameters, called the policy weights. If the policy is differentiable, the derivative of the probability for a specified state-action pair can be provided.

**diff_log**(*state*, *action*)
Compute the gradient of the logarithm of the probability density function, in the specified state and action pair, i.e.:

$$\nabla_\theta \log p(s, a)$$

**Parameters**

- **state** (*np.ndarray*) – the state where the gradient is computed
- **action** (*np.ndarray*) – the action where the gradient is computed

**Returns** The gradient of the logarithm of the pdf w.r.t. the policy weights

**diff**(*state*, *action*)
Compute the derivative of the probability density function, in the specified state and action pair. Normally it is computed w.r.t. the derivative of the logarithm of the probability density function, exploiting the likelihood ratio trick, i.e.:

$$\nabla_\theta p(s, a) = p(s, a) \nabla_\theta \log p(s, a)$$

**Parameters**

- **state** (*np.ndarray*) – the state where the derivative is computed
- **action** (*np.ndarray*) – the action where the derivative is computed

**Returns** The derivative w.r.t. the policy weights

**set_weights**(*weights*)
Setter.

**Parameters** **weights** (*np.ndarray*) – the vector of the new weights to be used by the policy

**get_weights**()
Getter.

**Returns** The current policy weights

**weights_size**
Property.

**Returns** The size of the policy weights

**__call__**(**args*)
Compute the probability of taking action in a certain state following the policy.

**Parameters** ***args** (*list*) – list containing a state or a state and an action.

**Returns** The probability of all actions following the policy in the given state if the list contains only the state, else the probability of the given action in the given state following the policy. If the action space is continuous, state and action must be provided

**__init__**
Initialize self. See help(type(self)) for accurate signature.

**draw_action**(*state*)
Sample an action in `state` using the policy.

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action sampled from the policy.

**reset**()
Useful when the policy needs a special initialization at the beginning of an episode.

### Gaussian policy

**class** mushroom.policy.gaussian_policy.**GaussianPolicy**(*mu*, *sigma*)
Bases: *mushroom.policy.policy.ParametricPolicy*

Gaussian policy. This is a differentiable policy for continuous action spaces. The policy samples an action in every state following a gaussian distribution, where the mean is computed in the state and the covariance matrix is fixed.

**__init__**(*mu*, *sigma*)
Constructor.

**Parameters**

- **mu** (Regressor) – the regressor representing the mean w.r.t. the state;
- **sigma** (*np.ndarray*) – a square positive definite matrix representing the covariance matrix. The size of this matrix must be n x n, where n is the action dimensionality.

**set_sigma**(*sigma*)
Setter.

**Parameters** **sigma** (*np.ndarray*) – the new covariance matrix. Must be a square positive definite matrix.

**__call__**(*state*, *action*)
Compute the probability of taking action in a certain state following the policy.

**Parameters** ***args** (*list*) – list containing a state or a state and an action.

**Returns** The probability of all actions following the policy in the given state if the list contains only the state, else the probability of the given action in the given state following the policy. If the action space is continuous, state and action must be provided

**draw_action**(*state*)
Sample an action in `state` using the policy.

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action sampled from the policy.

**diff_log**(*state*, *action*)
Compute the gradient of the logarithm of the probability density function, in the specified state and action pair, i.e.:

$$\nabla_\theta \log p(s, a)$$

**Parameters**

- **state** (*np.ndarray*) – the state where the gradient is computed
- **action** (*np.ndarray*) – the action where the gradient is computed

**Returns** The gradient of the logarithm of the pdf w.r.t. the policy weights

**set_weights**(*weights*)
Setter.

**Parameters** **weights** (*np.ndarray*) – the vector of the new weights to be used by the policy

**get_weights**()
Getter.

**Returns** The current policy weights

**weights_size**
Property.

**Returns** The size of the policy weights

**diff**(*state*, *action*)
Compute the derivative of the probability density function, in the specified state and action pair. Normally it is computed w.r.t. the derivative of the logarithm of the probability density function, exploiting the likelihood ratio trick, i.e.:

$$\nabla_\theta p(s, a) = p(s, a) \nabla_\theta \log p(s, a)$$

**Parameters**

- **state** (*np.ndarray*) – the state where the derivative is computed
- **action** (*np.ndarray*) – the action where the derivative is computed

**Returns** The derivative w.r.t. the policy weights

**reset**()
Useful when the policy needs a special initialization at the beginning of an episode.

**class** mushroom.policy.gaussian_policy.**DiagonalGaussianPolicy**(*mu*, *std*)
Bases: *mushroom.policy.policy.ParametricPolicy*

Gaussian policy with learnable standard deviation. The Covariance matrix is constrained to be a diagonal matrix, where the diagonal is the squared standard deviation vector. This is a differentiable policy for continuous action spaces. This policy is similar to the gaussian policy, but the weights includes also the standard deviation.

**__init__**(*mu*, *std*)
Constructor.

**Parameters**

- **mu** (Regressor) – the regressor representing the mean w.r.t. the state;
- **std** (*np.ndarray*) – a vector of standard deviations. The length of this vector must be equal to the action dimensionality.

**set_std**(*std*)
Setter.

**Parameters** **std** (*np.ndarray*) – the new standard deviation. Must be a square positive definite matrix.

**__call__**(*state*, *action*)
Compute the probability of taking action in a certain state following the policy.

**Parameters** ***args** (*list*) – list containing a state or a state and an action.

**Returns** The probability of all actions following the policy in the given state if the list contains only the state, else the probability of the given action in the given state following the policy. If the action space is continuous, state and action must be provided

**draw_action**(*state*)
Sample an action in `state` using the policy.

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action sampled from the policy.

**diff_log**(*state*, *action*)
Compute the gradient of the logarithm of the probability density function, in the specified state and action pair, i.e.:

$$\nabla_\theta \log p(s, a)$$

**Parameters**

- **state** (*np.ndarray*) – the state where the gradient is computed
- **action** (*np.ndarray*) – the action where the gradient is computed

**Returns** The gradient of the logarithm of the pdf w.r.t. the policy weights

**set_weights**(*weights*)
Setter.

**Parameters** **weights** (*np.ndarray*) – the vector of the new weights to be used by the policy

**get_weights**()
Getter.

**Returns** The current policy weights

**weights_size**
Property.

**Returns** The size of the policy weights

**diff**(*state*, *action*)
Compute the derivative of the probability density function, in the specified state and action pair. Normally

it is computed w.r.t. the derivative of the logarithm of the probability density function, exploiting the likelihood ratio trick, i.e.:

$$\nabla_\theta p(s,a) = p(s,a)\nabla_\theta \log p(s,a)$$

**Parameters**

- **state** (*np.ndarray*) – the state where the derivative is computed
- **action** (*np.ndarray*) – the action where the derivative is computed

**Returns** The derivative w.r.t. the policy weights

**reset**()
Useful when the policy needs a special initialization at the beginning of an episode.

**class** mushroom.policy.gaussian_policy.**StateStdGaussianPolicy**(*mu*, *std*, *eps=1e-06*)
Bases: *mushroom.policy.policy.ParametricPolicy*

Gaussian policy with learnable standard deviation. The Covariance matrix is constrained to be a diagonal matrix, where the diagonal is the squared standard deviation, which is computed for each state. This is a differentiable policy for continuous action spaces. This policy is similar to the diagonal gaussian policy, but a parametric regressor is used to compute the standard deviation, so the standard deviation depends on the current state.

**__init__**(*mu*, *std*, *eps=1e-06*)
Constructor.

**Parameters**

- **mu** (Regressor) – the regressor representing the mean w.r.t. the state;
- **std** (Regressor) – the regressor representing the standard deviations w.r.t. the state. The output dimensionality of the regressor must be equal to the action dimensionality;
- **eps** (*float, 1e-6*) – A positive constant added to the variance to ensure that is always greater than zero.

**__call__**(*state*, *action*)
Compute the probability of taking action in a certain state following the policy.

**Parameters** ***args** (*list*) – list containing a state or a state and an action.

**Returns** The probability of all actions following the policy in the given state if the list contains only the state, else the probability of the given action in the given state following the policy. If the action space is continuous, state and action must be provided

**draw_action**(*state*)
Sample an action in state using the policy.

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action sampled from the policy.

**diff_log**(*state*, *action*)
Compute the gradient of the logarithm of the probability density function, in the specified state and action pair, i.e.:

$$\nabla_\theta \log p(s,a)$$

**Parameters**

- **state** (*np.ndarray*) – the state where the gradient is computed
- **action** (*np.ndarray*) – the action where the gradient is computed

**Returns** The gradient of the logarithm of the pdf w.r.t. the policy weights

**set_weights**(*weights*)
Setter.

**Parameters** **weights** (*np.ndarray*) – the vector of the new weights to be used by the policy

**get_weights**()
Getter.

**Returns** The current policy weights

**weights_size**
Property.

**Returns** The size of the policy weights

**diff**(*state*, *action*)
Compute the derivative of the probability density function, in the specified state and action pair. Normally it is computed w.r.t. the derivative of the logarithm of the probability density function, exploiting the likelihood ratio trick, i.e.:

$$\nabla_\theta p(s,a) = p(s,a)\nabla_\theta \log p(s,a)$$

**Parameters**

- **state** (*np.ndarray*) – the state where the derivative is computed
- **action** (*np.ndarray*) – the action where the derivative is computed

**Returns** The derivative w.r.t. the policy weights

**reset**()
Useful when the policy needs a special initialization at the beginning of an episode.

**class** mushroom.policy.gaussian_policy.**StateLogStdGaussianPolicy**(*mu*, *log_std*)
Bases: *mushroom.policy.policy.ParametricPolicy*

Gaussian policy with learnable standard deviation. The Covariance matrix is constrained to be a diagonal matrix, the diagonal is computed by an exponential transformation of the logarithm of the standard deviation computed in each state. This is a differentiable policy for continuous action spaces. This policy is similar to the State std gaussian policy, but here the regressor represents the logarithm of the standard deviation.

**__init__**(*mu*, *log_std*)
Constructor.

**Parameters**

- **mu** (Regressor) – the regressor representing the mean w.r.t. the state;
- **log_std** (Regressor) – a regressor representing the logarithm of the variance w.r.t. the state. The output dimensionality of the regressor must be equal to the action dimensionality.

**__call__**(*state*, *action*)
Compute the probability of taking action in a certain state following the policy.

**Parameters** ***args** (*list*) – list containing a state or a state and an action.

**Returns** The probability of all actions following the policy in the given state if the list contains only the state, else the probability of the given action in the given state following the policy. If the action space is continuous, state and action must be provided

**draw_action**(*state*)
Sample an action in state using the policy.

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action sampled from the policy.

**diff_log**(*state*, *action*)
Compute the gradient of the logarithm of the probability density function, in the specified state and action pair, i.e.:

$$\nabla_\theta \log p(s, a)$$

**Parameters**

- **state** (*np.ndarray*) – the state where the gradient is computed
- **action** (*np.ndarray*) – the action where the gradient is computed

**Returns** The gradient of the logarithm of the pdf w.r.t. the policy weights

**set_weights**(*weights*)
Setter.

**Parameters** **weights** (*np.ndarray*) – the vector of the new weights to be used by the policy

**get_weights**()
Getter.

**Returns** The current policy weights

**weights_size**
Property.

**Returns** The size of the policy weights

**diff**(*state*, *action*)
Compute the derivative of the probability density function, in the specified state and action pair. Normally it is computed w.r.t. the derivative of the logarithm of the probability density function, exploiting the likelihood ratio trick, i.e.:

$$\nabla_\theta p(s, a) = p(s, a) \nabla_\theta \log p(s, a)$$

**Parameters**

- **state** (*np.ndarray*) – the state where the derivative is computed
- **action** (*np.ndarray*) – the action where the derivative is computed

**Returns** The derivative w.r.t. the policy weights

**reset**()
Useful when the policy needs a special initialization at the beginning of an episode.

### TD policy

**class** mushroom.policy.td_policy.**TDPolicy**
Bases: *mushroom.policy.policy.Policy*

**__init__**()
Constructor.

**set_q**(*approximator*)

**Parameters** **approximator** (*object*) – the approximator to use.

**get_q**()

**Returns** The approximator used by the policy.

**__call__**(**args*)
Compute the probability of taking action in a certain state following the policy.

**Parameters** ***args** (*list*) – list containing a state or a state and an action.

**Returns** The probability of all actions following the policy in the given state if the list contains only the state, else the probability of the given action in the given state following the policy. If the action space is continuous, state and action must be provided

**draw_action**(*state*)
Sample an action in `state` using the policy.

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action sampled from the policy.

**reset**()
Useful when the policy needs a special initialization at the beginning of an episode.

**class** mushroom.policy.td_policy.**EpsGreedy**(*epsilon*)
Bases: *mushroom.policy.td_policy.TDPolicy*

Epsilon greedy policy.

**__init__**(*epsilon*)
Constructor.

**Parameters** **epsilon** (Parameter) – the exploration coefficient. It indicates the probability of performing a random actions in the current step.

**__call__**(**args*)
Compute the probability of taking action in a certain state following the policy.

**Parameters** ***args** (*list*) – list containing a state or a state and an action.

**Returns** The probability of all actions following the policy in the given state if the list contains only the state, else the probability of the given action in the given state following the policy. If the action space is continuous, state and action must be provided

**draw_action**(*state*)
Sample an action in `state` using the policy.

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action sampled from the policy.

**set_epsilon**(*epsilon*)
Setter.

**Parameters**

- **epsilon** (Parameter) – the exploration coefficient. It indicates the
- **of performing a random actions in the current step.** (*probability*) –

**update**(**idx*)
Update the value of the epsilon parameter at the provided index (e.g. in case of different values of epsilon for each visited state according to the number of visits).

**Parameters** ***idx** (*list*) – index of the parameter to be updated.

**get_q**()

**Returns** The approximator used by the policy.

**reset**()
Useful when the policy needs a special initialization at the beginning of an episode.

**set_q**(*approximator*)

**Parameters** **approximator** (*object*) – the approximator to use.

**class** mushroom.policy.td_policy.**Boltzmann**(*beta*)
Bases: *mushroom.policy.td_policy.TDPolicy*

Boltzmann softmax policy.

**__init__**(*beta*)
Constructor.

**Parameters**

- **beta** (Parameter) – the inverse of the temperature distribution. As
- **temperature approaches infinity, the policy becomes more and** (*the*) –
- **random. As the temperature approaches 0.0, the policy becomes** (*more*) –
- **and more greedy.** (*more*) –

**__call__**(**args*)
Compute the probability of taking action in a certain state following the policy.

**Parameters** ***args** (*list*) – list containing a state or a state and an action.

**Returns** The probability of all actions following the policy in the given state if the list contains only the state, else the probability of the given action in the given state following the policy. If the action space is continuous, state and action must be provided

**draw_action**(*state*)
Sample an action in state using the policy.

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action sampled from the policy.

**get_q**()

**Returns** The approximator used by the policy.

**reset**()
Useful when the policy needs a special initialization at the beginning of an episode.

**set_q**(*approximator*)

**Parameters** **approximator** (*object*) – the approximator to use.

**class** mushroom.policy.td_policy.**Mellowmax**(*omega*, *beta_min=-10.0*, *beta_max=10.0*)
Bases: *mushroom.policy.td_policy.Boltzmann*

Mellowmax policy. "An Alternative Softmax Operator for Reinforcement Learning". Asadi K. and Littman M.L.. 2017.

**__init__**(*omega*, *beta_min=-10.0*, *beta_max=10.0*)
Constructor.

**Parameters**

- **omega** (Parameter) – the omega parameter of the policy from which beta of the Boltzmann policy is computed;
- **beta_min** (*float, -10.*) – one end of the bracketing interval for minimization with Brent's method;
- **beta_max** (*float, 10.*) – the other end of the bracketing interval for minimization with Brent's method.

**__call__**(**args*)
Compute the probability of taking action in a certain state following the policy.

**Parameters** ***args** (*list*) – list containing a state or a state and an action.

**Returns** The probability of all actions following the policy in the given state if the list contains only the state, else the probability of the given action in the given state following the policy. If the action space is continuous, state and action must be provided

**draw_action**(*state*)
Sample an action in `state` using the policy.

**Parameters** **state** (*np.ndarray*) – the state where the agent is.

**Returns** The action sampled from the policy.

**get_q**()

**Returns** The approximator used by the policy.

**reset**()
Useful when the policy needs a special initialization at the beginning of an episode.

**set_q**(*approximator*)

**Parameters** **approximator** (*object*) – the approximator to use.

### 3.1.7 Distributions

**class** mushroom.distributions.distribution.**Distribution**
Bases: `object`

Interface for Distributions to represent a generic probability distribution. Probability distributions are often used by black box optimization algorithms in order to perform exploration in parameter space. In literature, they are also known as high level policies.

**sample**()
Draw a sample from the distribution.

**Returns** A random vector sampled from the distribution.

**log_pdf**(*theta*)
Compute the logarithm of the probability density function in the specified point

**Parameters** **theta** (*np.ndarray*) – the point where the log pdf is calculated

**Returns** The value of the log pdf in the specified point.

**__call__**(*theta*)
Compute the probability density function in the specified point

**Parameters** **theta** (*np.ndarray*) – the point where the pdf is calculated

**Returns** The value of the pdf in the specified point.

**mle**(*theta*, *weights=None*)
Compute the (weighted) maximum likelihood estimate of the points, and update the distribution accordingly.

**Parameters**

- **theta** (*np.ndarray*) – a set of points, every row is a sample
- **weights** (*np.ndarray, None*) – a vector of weights. If specified the weighted maximum likelihood estimate is computed instead of the plain maximum likelihood. The number of elements of this vector must be equal to the number of rows of the theta matrix.

**diff_log**(*theta*)
Compute the derivative of the gradient of the probability denstity function in the specified point.

**Parameters**

- **theta** (*np.ndarray*) – the point where the gradient of the log pdf is
- **calculated** –

**Returns** The gradient of the log pdf in the specified point.

**diff**(*theta*)
Compute the derivative of the probability density function, in the specified point. Normally it is computed w.r.t. the derivative of the logarithm of the probability density function, exploiting the likelihood ratio trick, i.e.:

$$\nabla_\rho p(\theta) = p(\theta)\nabla_\rho \log p(\theta)$$

**Parameters**

- **theta** (*np.ndarray*) – the point where the gradient of the pdf is
- **calculated.** –

**Returns** The gradient of the pdf in the specified point.

**get_parameters**()
Getter.

**Returns** The current distribution parameters.

**set_parameters**(*rho*)
Setter.

**Parameters** **rho** (*np.ndarray*) – the vector of the new parameters to be used by the distribution

**parameters_size**
Property.

**Returns** The size of the distribution parameters.

**__init__**
Initialize self. See help(type(self)) for accurate signature.

### Gaussian

**class** mushroom.distributions.gaussian.**GaussianDistribution**(*mu*, *sigma*)
Bases: *mushroom.distributions.distribution.Distribution*

Gaussian distribution with fixed covariance matrix. The parameters vector represents only the mean.

**__init__**(*mu*, *sigma*)
Initialize self. See help(type(self)) for accurate signature.

**sample**()
Draw a sample from the distribution.

**Returns** A random vector sampled from the distribution.

**log_pdf**(*theta*)
Compute the logarithm of the probability density function in the specified point

**Parameters** **theta** (*np.ndarray*) – the point where the log pdf is calculated

**Returns** The value of the log pdf in the specified point.

**__call__**(*theta*)
Compute the probability density function in the specified point

**Parameters** **theta** (*np.ndarray*) – the point where the pdf is calculated

**Returns** The value of the pdf in the specified point.

**mle**(*theta*, *weights=None*)
Compute the (weighted) maximum likelihood estimate of the points, and update the distribution accordingly.

**Parameters**

- **theta** (*np.ndarray*) – a set of points, every row is a sample
- **weights** (*np.ndarray, None*) – a vector of weights. If specified the weighted maximum likelihood estimate is computed instead of the plain maximum likelihood. The number of elements of this vector must be equal to the number of rows of the theta matrix.

**diff_log**(*theta*)
Compute the derivative of the gradient of the probability denstity function in the specified point.

**Parameters**

- **theta** (*np.ndarray*) – the point where the gradient of the log pdf is
- **calculated** –

**Returns** The gradient of the log pdf in the specified point.

**get_parameters**()
Getter.

**Returns** The current distribution parameters.

**set_parameters**(*rho*)
Setter.

**Parameters** **rho** (*np.ndarray*) – the vector of the new parameters to be used by the distribution

**parameters_size**
Property.

**Returns** The size of the distribution parameters.

**diff**(*theta*)
Compute the derivative of the probability density function, in the specified point. Normally it is computed

w.r.t. the derivative of the logarithm of the probability density function, exploiting the likelihood ratio trick, i.e.:

$$\nabla_\rho p(\theta) = p(\theta)\nabla_\rho \log p(\theta)$$

**Parameters**

- **theta** (*np.ndarray*) – the point where the gradient of the pdf is
- **calculated.** –

**Returns** The gradient of the pdf in the specified point.

**class** mushroom.distributions.gaussian.**GaussianDiagonalDistribution**(*mu*, *std*)
Bases: *mushroom.distributions.distribution.Distribution*

Gaussian distribution with diagonal covariance matrix. The parameters vector represents the mean and the standard deviation for each dimension.

**__init__**(*mu*, *std*)
Initialize self. See help(type(self)) for accurate signature.

**sample**()
Draw a sample from the distribution.

**Returns** A random vector sampled from the distribution.

**log_pdf**(*theta*)
Compute the logarithm of the probability density function in the specified point

**Parameters** **theta** (*np.ndarray*) – the point where the log pdf is calculated

**Returns** The value of the log pdf in the specified point.

**__call__**(*theta*)
Compute the probability density function in the specified point

**Parameters** **theta** (*np.ndarray*) – the point where the pdf is calculated

**Returns** The value of the pdf in the specified point.

**mle**(*theta*, *weights=None*)
Compute the (weighted) maximum likelihood estimate of the points, and update the distribution accordingly.

**Parameters**

- **theta** (*np.ndarray*) – a set of points, every row is a sample
- **weights** (*np.ndarray, None*) – a vector of weights. If specified the weighted maximum likelihood estimate is computed instead of the plain maximum likelihood. The number of elements of this vector must be equal to the number of rows of the theta matrix.

**diff_log**(*theta*)
Compute the derivative of the gradient of the probability denstity function in the specified point.

**Parameters**

- **theta** (*np.ndarray*) – the point where the gradient of the log pdf is
- **calculated** –

**Returns** The gradient of the log pdf in the specified point.

**get_parameters**()
Getter.

**Returns** The current distribution parameters.

**set_parameters**(*rho*)
Setter.

**Parameters** **rho** (*np.ndarray*) – the vector of the new parameters to be used by the distribution

**parameters_size**
Property.

**Returns** The size of the distribution parameters.

**diff**(*theta*)
Compute the derivative of the probability density function, in the specified point. Normally it is computed w.r.t. the derivative of the logarithm of the probability density function, exploiting the likelihood ratio trick, i.e.:

$$\nabla_\rho p(\theta) = p(\theta)\nabla_\rho \log p(\theta)$$

**Parameters**

- **theta** (*np.ndarray*) – the point where the gradient of the pdf is
- **calculated.** –

**Returns** The gradient of the pdf in the specified point.

**class** mushroom.distributions.gaussian.**GaussianCholeskyDistribution**(*mu*, *sigma*)
Bases: *mushroom.distributions.distribution.Distribution*

Gaussian distribution with full covariance matrix. The parameters vector represents the mean and the Cholesky decomposition of the covariance matrix. This parametrization enforce the covariance matrix to be positive definite.

**__init__**(*mu*, *sigma*)
Initialize self. See help(type(self)) for accurate signature.

**sample**()
Draw a sample from the distribution.

**Returns** A random vector sampled from the distribution.

**log_pdf**(*theta*)
Compute the logarithm of the probability density function in the specified point

**Parameters** **theta** (*np.ndarray*) – the point where the log pdf is calculated

**Returns** The value of the log pdf in the specified point.

**__call__**(*theta*)
Compute the probability density function in the specified point

**Parameters** **theta** (*np.ndarray*) – the point where the pdf is calculated

**Returns** The value of the pdf in the specified point.

**mle**(*theta*, *weights=None*)
Compute the (weighted) maximum likelihood estimate of the points, and update the distribution accordingly.

**Parameters**

- **theta** (*np.ndarray*) – a set of points, every row is a sample

- **weights** (*np.ndarray, None*) – a vector of weights. If specified the weighted maximum likelihood estimate is computed instead of the plain maximum likelihood. The number of elements of this vector must be equal to the number of rows of the theta matrix.

**diff_log**(*theta*)
Compute the derivative of the gradient of the probability denstity function in the specified point.

**Parameters**

- **theta** (*np.ndarray*) – the point where the gradient of the log pdf is
- **calculated** –

**Returns** The gradient of the log pdf in the specified point.

**get_parameters**()
Getter.

**Returns** The current distribution parameters.

**set_parameters**(*rho*)
Setter.

**Parameters** **rho** (*np.ndarray*) – the vector of the new parameters to be used by the distribution

**parameters_size**
Property.

**Returns** The size of the distribution parameters.

**diff**(*theta*)
Compute the derivative of the probability density function, in the specified point. Normally it is computed w.r.t. the derivative of the logarithm of the probability density function, exploiting the likelihood ratio trick, i.e.:

$$\nabla_\rho p(\theta) = p(\theta)\nabla_\rho \log p(\theta)$$

**Parameters**

- **theta** (*np.ndarray*) – the point where the gradient of the pdf is
- **calculated.** –

**Returns** The gradient of the pdf in the specified point.

### 3.1.8 Solvers

#### Dynamic programming

mushroom.solvers.dynamic_programming.**value_iteration**(*prob*, *reward*, *gamma*, *eps*)
Value iteration algorithm to solve a dynamic programming problem.

**Parameters**

- **prob** (*np.ndarray*) – transition probability matrix;
- **reward** (*np.ndarray*) – reward matrix;
- **gamma** (*float*) – discount factor;
- **eps** (*float*) – accuracy threshold.

**Returns** The optimal value of each state.

mushroom.solvers.dynamic_programming.**policy_iteration**(*prob*, *reward*, *gamma*)
Policy iteration algorithm to solve a dynamic programming problem.

**Parameters**

- **prob** (*np.ndarray*) – transition probability matrix;
- **reward** (*np.ndarray*) – reward matrix;
- **gamma** (*float*) – discount factor.

**Returns** The optimal value of each state and the optimal policy.

### 3.1.9 Utils

#### Angles

mushroom.utils.angles.**normalize_angle_positive**(*angle*)
Wrap the angle between 0 and 2 * pi.

**Parameters** **angle** (*float*) – angle to wrap.

**Returns** The wrapped angle.

mushroom.utils.angles.**normalize_angle**(*angle*)
Wrap the angle between -pi and pi.

**Parameters** **angle** (*float*) – angle to wrap.

**Returns** The wrapped angle.

#### Callbacks

**class** mushroom.utils.callbacks.**CollectDataset**
Bases: object

This callback can be used to collect samples during the learning of the agent.

**__init__**()
Constructor.

**__call__**(*dataset*)
Add samples to the samples list.

**Parameters** **dataset** (*list*) – the samples to collect.

**get**()

**Returns** The current samples list.

**clean**()
Deletes the current dataset

**class** mushroom.utils.callbacks.**CollectQ**(*approximator*)
Bases: object

This callback can be used to collect the action values in all states at the current time step.

**__init__**(*approximator*)
Constructor.

**Parameters** **approximator** (*[Table, EnsembleTable]*) – the approximator to use to predict the action values.

**__call__**(***kwargs*)
Add action values to the action-values list.

**Parameters** ****kwargs** (*dict*) – empty dictionary.

**get_values**()

**Returns** The current action-values list.

**class** mushroom.utils.callbacks.**CollectMaxQ**(*approximator*, *state*)
Bases: object

This callback can be used to collect the maximum action value in a given state at each call.

**__init__**(*approximator*, *state*)
Constructor.

**Parameters**

- **approximator** ([Table, EnsembleTable]) – the approximator to use;
- **state** (*np.ndarray*) – the state to consider.

**__call__**(***kwargs*)
Add maximum action values to the maximum action-values list.

**Parameters** ****kwargs** (*dict*) – empty dictionary.

**get_values**()

**Returns** The current maximum action-values list.

**class** mushroom.utils.callbacks.**CollectParameters**(*parameter*, **idx*)
Bases: object

This callback can be used to collect the values of a parameter (e.g. learning rate) during a run of the agent.

**__init__**(*parameter*, **idx*)
Constructor.

**Parameters**

- **parameter** (Parameter) – the parameter whose values have to be collected;
- ***idx** (*list*) – index of the parameter when the parameter is tabular.

**__call__**(***kwargs*)
Add the parameter value to the parameter values list.

**Parameters** ****kwargs** (*dict*) – empty dictionary.

**get_values**()

**Returns** The current parameter values list.

### Dataset

mushroom.utils.dataset.**parse_dataset**(*dataset*, *features=None*)
Split the dataset in its different components and return them.

**Parameters**

- **dataset** (*list*) – the dataset to parse;
- **features** (*object, None*) – features to apply to the states.

**Returns** The np.ndarray of state, action, reward, next_state, absorbing flag and last step flag. Features are applied to `state` and `next_state`, when provided.

`mushroom.utils.dataset.`**`episodes_length`**(*dataset*)
Compute the length of each episode in the dataset.

**Parameters** **`dataset`** (*`list`*) – the dataset to consider.

**Returns** A list of length of each episode in the dataset.

`mushroom.utils.dataset.`**`select_episodes`**(*dataset*, *n_episodes*, *parse=False*)
Return the first `n_episodes` episodes in the provided dataset.

**Parameters**

- **`dataset`** (*`list`*) – the dataset to consider;
- **`n_episodes`** (*`int`*) – the number of episodes to pick from the dataset;
- **`parse`** (*`bool, False`*) – whether to parse the dataset to return.

**Returns** A subset of the dataset containing the first `n_episodes` episodes.

`mushroom.utils.dataset.`**`select_samples`**(*dataset*, *n_samples*, *parse=False*)
Return the randomly picked desired number of samples in the provided dataset.

**Parameters**

- **`dataset`** (*`list`*) – the dataset to consider;
- **`n_samples`** (*`int`*) – the number of samples to pick from the dataset;
- **`parse`** (*`bool, False`*) – whether to parse the dataset to return.

**Returns** A subset of the dataset containing randomly picked `n_samples` samples.

`mushroom.utils.dataset.`**`compute_J`**(*dataset*, *gamma=1.0*)
Compute the cumulative discounted reward of each episode in the dataset.

**Parameters**

- **`dataset`** (*`list`*) – the dataset to consider;
- **`gamma`** (*`float, 1.`*) – discount factor.

**Returns** The cumulative discounted reward of each episode in the dataset.

`mushroom.utils.dataset.`**`compute_scores`**(*dataset*)
Compute the scores of each episode in the dataset. This is meant to be used for the Atari environments.

**Parameters** **`dataset`** (*`list`*) – the dataset to consider.

**Returns**

The minimum score reached in an episode, the maximum score reached in an episode, the mean score reached, the number of completed games.

If no game has been completed, it returns 0 for all values.

### Eligibility trace

`mushroom.utils.eligibility_trace.`**`EligibilityTrace`**(*shape*, *name='replacing'*)
Factory method to create an eligibility trace of the provided type.

**Parameters**

- **`shape`** (*`list`*) – shape of the eligibility trace table;

- **name** (*str, 'replacing'*) – type of the eligibility trace.

**Returns** The eligibility trace table of the provided shape and type.

**class** mushroom.utils.eligibility_trace.**ReplacingTrace**(*shape*, *initial_value=0.0*, *dtype=None*)

Bases: *mushroom.utils.table.Table*

Replacing trace.

**reset**()

**update**(*state*, *action*)

**__init__**(*shape*, *initial_value=0.0*, *dtype=None*)
Constructor.

**Parameters**

- **shape** (*tuple*) – the shape of the tabular regressor.
- **initial_value** (*float, 0.*) – the initial value for each entry of the tabular regressor.
- **dtype** (*[int, float], None*) – the dtype of the table array.

**fit**(*x*, *y*)

**Parameters**

- **x** (*int*) – index of the table to be filled;
- **y** (*float*) – value to fill in the table.

**n_actions**
The number of actions considered by the table.

**Type** Returns

**predict**(**z*)
Predict the output of the table given an input.

**Parameters**

- ***z** (*list*) – list of input of the model. If the table is a Q-table,
- **list may contain states or states and actions depending** (*this*) – on whether the call requires to predict all q-values or only one q-value corresponding to the provided action;

**Returns** The table prediction.

**shape**
The shape of the table.

**Type** Returns

**class** mushroom.utils.eligibility_trace.**AccumulatingTrace**(*shape*, *initial_value=0.0*, *dtype=None*)

Bases: *mushroom.utils.table.Table*

Accumulating trace.

**reset**()

**update**(*state*, *action*)

**__init__**(*shape*, *initial_value=0.0*, *dtype=None*)
Constructor.

**Parameters**

- **shape** (*tuple*) – the shape of the tabular regressor.
- **initial_value** (*float, 0.*) – the initial value for each entry of the tabular regressor.
- **dtype** (*[int, float], None*) – the dtype of the table array.

**fit**(*x*, *y*)

**Parameters**

- **x** (*int*) – index of the table to be filled;
- **y** (*float*) – value to fill in the table.

**n_actions**
The number of actions considered by the table.

**Type** Returns

**predict**(**z*)
Predict the output of the table given an input.

**Parameters**

- ***z** (*list*) – list of input of the model. If the table is a Q-table,
- **list may contain states or states and actions depending** (*this*) – on whether the call requires to predict all q-values or only one q-value corresponding to the provided action;

**Returns** The table prediction.

**shape**
The shape of the table.

**Type** Returns

### Features

mushroom.utils.features.**uniform_grid**(*n_centers*, *low*, *high*)
This function is used to create the parameters of uniformly spaced radial basis functions with 25% of overlap. It creates a uniformly spaced grid of `n_centers[i]` points in each `ranges[i]`. Also returns a vector containing the appropriate scales of the radial basis functions.

**Parameters**

- **n_centers** (*list*) – number of centers of each dimension;
- **low** (*np.ndarray*) – lowest value for each dimension;
- **high** (*np.ndarray*) – highest value for each dimension.

**Returns** The uniformly spaced grid and the scale vector.

### Folder

mushroom.utils.folder.**mk_dir_recursive**(*dir_path*)
Create a directory and, if needed, all the directory tree. Differently from os.mkdir, this function does not raise exception when the directory already exists.

**Parameters** **dir_path** (*str*) – the path of the directory to create.

mushroom.utils.folder.**force_symlink**(*src*, *dst*)
Create a symlink deleting the previous one, if it already exists.

**Parameters**

- **src** (*str*) – source;
- **dst** (*str*) – destination.

### Minibatches

mushroom.utils.minibatches.**minibatch_number**(*size*, *batch_size*)
Function to retrieve the number of batches, given a batch sizes.

**Parameters**

- **size** (*int*) – size of the dataset;
- **batch_size** (*int*) – size of the batches.

**Returns** The number of minibatches in the dataset.

mushroom.utils.minibatches.**minibatch_generator**(*batch_size*, **dataset*)
Generator that creates a minibatch from the full dataset.

**Parameters**

- **batch_size** (*int*) – the maximum size of each minibatch;
- **dataset** – the dataset to be splitted.

**Returns** The current minibatch.

### Numerical gradient

mushroom.utils.numerical_gradient.**numerical_diff_policy**(*policy*, *state*, *action*, *eps=1e-06*)
Compute the gradient of a policy in (state, action) numerically.

**Parameters**

- **policy** (Policy) – the policy whose gradient has to be returned;
- **state** (*np.ndarray*) – the state;
- **action** (*np.ndarray*) – the action;
- **eps** (*float, 1e-6*) – the value of the perturbation.

**Returns** The gradient of the provided policy in (state, action) computed numerically.

mushroom.utils.numerical_gradient.**numerical_diff_dist**(*dist*, *theta*, *eps=1e-06*)
Compute the gradient of a distribution in theta numerically.

**Parameters**

- **dist** (Distribution) – the distribution whose gradient has to be returned;
- **theta** (*np.ndarray*) – the parametrization where to compute the gradient;
- **eps** (*float, 1e-6*) – the value of the perturbation.

**Returns** The gradient of the provided distribution `theta` computed numerically.

### Parameters

**class** mushroom.utils.parameters.**Parameter**(*value*, *min_value=None*, *max_value=None*, *size=(1, )*)

Bases: `object`

This class implements function to manage parameters, such as learning rate. It also allows to have a single parameter for each state of state-action tuple.

**__init__**(*value*, *min_value=None*, *max_value=None*, *size=(1, )*)

Constructor.

**Parameters**

- **value** (*float*) – initial value of the parameter;
- **min_value** (*float, None*) – minimum value that the parameter can reach when decreasing;
- **max_value** (*float, None*) – maximum value that the parameter can reach when increasing;
- **size** (*tuple, (1,)*) – shape of the matrix of parameters; this shape can be used to have a single parameter for each state or state-action tuple.

**__call__**(**idx*, ***kwargs*)

Update and return the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The updated parameter in the provided index.

**get_value**(**idx*, ***kwargs*)

Return the current value of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The current value of the parameter in the provided index.

**_compute**(**idx*, ***kwargs*)

**Returns** The value of the parameter in the provided index.

**update**(**idx*, ***kwargs*)

Updates the number of visit of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter whose number of visits has to be updated.

**shape**

The shape of the table of parameters.

**Type** Returns

**class** mushroom.utils.parameters.**LinearParameter**(*value*, *threshold_value*, *n*, *size=(1, )*)

Bases: *mushroom.utils.parameters.Parameter*

This class implements a linearly changing parameter according to the number of times it has been used.

**__init__**(*value*, *threshold_value*, *n*, *size=(1, )*)
Constructor.

**Parameters**

- **value** (*float*) – initial value of the parameter;
- **min_value** (*float, None*) – minimum value that the parameter can reach when decreasing;
- **max_value** (*float, None*) – maximum value that the parameter can reach when increasing;
- **size** (*tuple, (1,)*) – shape of the matrix of parameters; this shape can be used to have a single parameter for each state or state-action tuple.

**_compute**(**idx*, ***kwargs*)
Returns: The value of the parameter in the provided index.

**__call__**(**idx*, ***kwargs*)
Update and return the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The updated parameter in the provided index.

**get_value**(**idx*, ***kwargs*)
Return the current value of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The current value of the parameter in the provided index.

**shape**
The shape of the table of parameters.

**Type** Returns

**update**(**idx*, ***kwargs*)
Updates the number of visit of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter whose number of visits has to be updated.

**class** mushroom.utils.parameters.**ExponentialParameter**(*value*, *exp=1.0*, *min_value=None*, *max_value=None*, *size=(1, )*)
Bases: *mushroom.utils.parameters.Parameter*

This class implements a exponentially changing parameter according to the number of times it has been used.

**__init__**(*value*, *exp=1.0*, *min_value=None*, *max_value=None*, *size=(1, )*)
Constructor.

**Parameters**

- **value** (*float*) – initial value of the parameter;
- **min_value** (*float, None*) – minimum value that the parameter can reach when decreasing;
- **max_value** (*float, None*) – maximum value that the parameter can reach when increasing;
- **size** (*tuple, (1,)*) – shape of the matrix of parameters; this shape can be used to have a single parameter for each state or state-action tuple.

**_compute**(*idx, **kwargs)
Returns: The value of the parameter in the provided index.

**__call__**(*idx, **kwargs)
Update and return the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The updated parameter in the provided index.

**get_value**(*idx, **kwargs)
Return the current value of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The current value of the parameter in the provided index.

**shape**
The shape of the table of parameters.

**Type** Returns

**update**(*idx, **kwargs)
Updates the number of visit of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter whose number of visits has to be updated.

**class** mushroom.utils.parameters.**AdaptiveParameter**(*value*)
Bases: object

This class implements a basic adaptive gradient step. Instead of moving of a step proportional to the gradient, takes a step limited by a given metric. To specify the metric, the natural gradient has to be provided. If natural gradient is not provided, the identity matrix is used.

The step rule is:

$$\Delta\theta = \underset{\Delta\vartheta}{argmax}\Delta\vartheta^t\nabla_\theta J$$
$$s.t. : \Delta\vartheta^T M\Delta\vartheta \leq \varepsilon$$

Lecture notes, Neumann G. http://www.ias.informatik.tu-darmstadt.de/uploads/Geri/lecture-notes-constraint.pdf

**__init__**(*value*)
Initialize self. See help(type(self)) for accurate signature.

**__call__**(*args, **kwargs)
Call self as a function.

### Preprocessor

**class** mushroom.utils.preprocessor.**Preprocessor**
Bases: object

This is the interface class of the preprocessors.

**__call__**(*x*)
Compute the preprocessing of the given input according to the type of preprocessor.

**Parameters** **x** (*np.ndarray*) – the array to preprocess.

**Returns** The preprocessed input data array.

**class** mushroom.utils.preprocessor.**Scaler**(*coeff*)
Bases: *mushroom.utils.preprocessor.Preprocessor*

This class implements the function to scale the input data by a given coefficient.

**__init__**(*coeff*)
Constructor.

**Parameters** **coeff** (*float*) – the coefficient to use to scale input data.

**class** mushroom.utils.preprocessor.**Binarizer**(*threshold*, *geq=True*)
Bases: *mushroom.utils.preprocessor.Preprocessor*

This class implements the function to binarize the values of an array according to a provided threshold value.

**__init__**(*threshold*, *geq=True*)
Constructor.

**Parameters**

- **threshold** (*float*) – the coefficient to use to scale input data.
- **geq** (*bool, True*) – whether the threshold include equal elements or not.

**class** mushroom.utils.preprocessor.**Filter**(*idxs*)
Bases: *mushroom.utils.preprocessor.Preprocessor*

This class implements the function to filter the values of an array according to a provided array of indexes.

**__init__**(*idxs*)
Constructor.

**Parameters** **idxs** (*float*) – the array of idxs to use to filter input data.

### Replay memory

**class** mushroom.utils.replay_memory.**ReplayMemory**(*initial_size*, *max_size*)
Bases: object

This class implements function to manage a replay memory as the one used in "Human-Level Control Through Deep Reinforcement Learning" by Mnih V. et al..

**__init__**(*initial_size*, *max_size*)
Constructor.

**Parameters**

- **initial_size** (*int*) – initial number of elements in the replay memory;
- **max_size** (*int*) – maximum number of elements that the replay memory can contain.

**add**(*dataset*)
Add elements to the replay memory.

**Parameters** **dataset** (*list*) – list of elements to add to the replay memory.

**get**(*n_samples*)
Returns the provided number of states from the replay memory.

**Parameters** **n_samples** (*int*) – the number of samples to return.

**Returns** The requested number of samples.

**reset**()
Reset the replay memory.

**initialized**
Whether the replay memory has reached the number of elements that allows it to be used.

**Type** Returns

**size**
The number of elements contained in the replay memory.

**Type** Returns

### Spaces

**class** mushroom.utils.spaces.**Box**(*low*, *high*, *shape=None*)
Bases: object

This class implements functions to manage continuous states and action spaces. It is similar to the Box class in gym.spaces.box.

**__init__**(*low*, *high*, *shape=None*)
Constructor.

**Parameters**

- **low** (*[float, np.ndarray]*) – the minimum value of each dimension of the space. If a scalar value is provided, this value is considered as the minimum one for each dimension. If a np.ndarray is provided, each i-th element is considered the minimum value of the i-th dimension;
- **high** (*[float, np.ndarray]*) – the maximum value of dimensions of the space. If a scalar value is provided, this value is considered as the maximum one for each dimension. If a np.ndarray is provided, each i-th element is considered the maximum value of the i-th dimension;
- **shape** (*np.ndarray, None*) – the dimension of the space. Must match the shape of low and high, if they are np.ndarray.

**low**
The minimum value of each dimension of the space.

**Type** Returns

**high**
The maximum value of each dimension of the space.

**Type** Returns

**shape**
The dimensions of the space.

**Type** Returns

**class** mushroom.utils.spaces.**Discrete**(*n*)
Bases: object

This class implements functions to manage discrete states and action spaces. It is similar to the Discrete class in gym.spaces.discrete.

**__init__**(*n*)
Constructor.

**Parameters** **n** (*int*) – the number of values of the space.

**size**
The number of elements of the space.

**Type** Returns

**shape**
The shape of the space that is always (1,).

**Type** Returns

### Table

**class** mushroom.utils.table.**Table**(*shape*, *initial_value=0.0*, *dtype=None*)
Bases: object

Table regressor. Used for discrete state and action spaces.

**__init__**(*shape*, *initial_value=0.0*, *dtype=None*)
Constructor.

**Parameters**

- **shape** (*tuple*) – the shape of the tabular regressor.
- **initial_value** (*float, 0.*) – the initial value for each entry of the tabular regressor.
- **dtype** (*[int, float], None*) – the dtype of the table array.

**fit**(*x*, *y*)

**Parameters**

- **x** (*int*) – index of the table to be filled;
- **y** (*float*) – value to fill in the table.

**predict**(**z*)
Predict the output of the table given an input.

**Parameters**

- ***z** (*list*) – list of input of the model. If the table is a Q-table,
- **list may contain states or states and actions depending** (*this*) – on whether the call requires to predict all q-values or only one q-value corresponding to the provided action;

**Returns** The table prediction.

**n_actions**
The number of actions considered by the table.

**Type** Returns

**shape**
The shape of the table.

**Type** Returns

**class** mushroom.utils.table.**EnsembleTable**(*n_models*, *shape*, *prediction='mean'*)
Bases: mushroom.approximators._implementations.ensemble.Ensemble

This class implements functions to manage table ensembles.

**__init__** (*n_models*, *shape*, *prediction='mean'*)
Constructor.

**Parameters**

- **n_models** (*int*) – number of models in the ensemble;
- **shape** (*np.ndarray*) – shape of each table in the ensemble;
- **prediction** (*str, 'mean'*) – type of prediction to return.

**fit** (**z*, ***fit_params*)
Fit the `idx`-th model of the ensemble if `idx` is provided, every model otherwise.

**Parameters**

- ***z** (*list*) – a list containing the inputs to use to predict with each regressor of the ensemble;
- ****fit_params** (*dict*) – other params.

**model**
The list of the models in the ensemble.

**Type** Returns

**predict** (**z*, ***predict_params*)
Predict.

**Parameters**

- ***z** (*list*) – a list containing the inputs to use to predict with each regressor of the ensemble;
- ****predict_params** (*dict*) – other params.

**Returns** The predictions of the model.

**reset** ()
Reset the model parameters.

### Variance parameters

**class** mushroom.utils.variance_parameters.**VarianceParameter**(*value*, *exponential=False*, *min_value=None*, *tol=1.0*, *size=(1, )*)

Bases: *mushroom.utils.parameters.Parameter*

Abstract class to implement variance-dependent parameters. A `target` parameter is expected.

**__init__** (*value*, *exponential=False*, *min_value=None*, *tol=1.0*, *size=(1, )*)
Constructor.

**Parameters**

- **value** (*float*) – initial value of the parameter;
- **min_value** (*float, None*) – minimum value that the parameter can reach when decreasing;
- **max_value** (*float, None*) – maximum value that the parameter can reach when increasing;

- **size** (*tuple, (1,)*) – shape of the matrix of parameters; this shape can be used to have a single parameter for each state or state-action tuple.

**_compute**(**idx*, ***kwargs*)
Returns: The value of the parameter in the provided index.

**update**(**idx*, ***kwargs*)
Updates the number of visit of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter whose number of visits has to be updated.

**__call__**(**idx*, ***kwargs*)
Update and return the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The updated parameter in the provided index.

**get_value**(**idx*, ***kwargs*)
Return the current value of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The current value of the parameter in the provided index.

**shape**
The shape of the table of parameters.

**Type** Returns

**class** mushroom.utils.variance_parameters.**VarianceIncreasingParameter**(*value*, *exponential=False*, *min_value=None*, *tol=1.0*, *size=(1, )*)
Bases: *mushroom.utils.variance_parameters.VarianceParameter*

**__call__**(**idx*, ***kwargs*)
Update and return the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The updated parameter in the provided index.

**__init__**(*value*, *exponential=False*, *min_value=None*, *tol=1.0*, *size=(1, )*)
Constructor.

**Parameters**

- **value** (*float*) – initial value of the parameter;
- **min_value** (*float, None*) – minimum value that the parameter can reach when decreasing;
- **max_value** (*float, None*) – maximum value that the parameter can reach when increasing;
- **size** (*tuple, (1,)*) – shape of the matrix of parameters; this shape can be used to have a single parameter for each state or state-action tuple.

**_compute**(**idx*, ***kwargs*)
Returns: The value of the parameter in the provided index.

**get_value**(**idx*, ***kwargs*)
Return the current value of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The current value of the parameter in the provided index.

**shape**
The shape of the table of parameters.

**Type** Returns

**update**(**idx*, ***kwargs*)
Updates the number of visit of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter whose number of visits has to be updated.

**class** mushroom.utils.variance_parameters.**VarianceDecreasingParameter**(*value*, *exponential=False*, *min_value=None*, *tol=1.0*, *size=(1, )*)
Bases: *mushroom.utils.variance_parameters.VarianceParameter*

**__call__**(**idx*, ***kwargs*)
Update and return the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The updated parameter in the provided index.

**__init__**(*value*, *exponential=False*, *min_value=None*, *tol=1.0*, *size=(1, )*)
Constructor.

**Parameters**

- **value** (*float*) – initial value of the parameter;
- **min_value** (*float, None*) – minimum value that the parameter can reach when decreasing;
- **max_value** (*float, None*) – maximum value that the parameter can reach when increasing;
- **size** (*tuple, (1,)*) – shape of the matrix of parameters; this shape can be used to have a single parameter for each state or state-action tuple.

**_compute**(**idx*, ***kwargs*)
Returns: The value of the parameter in the provided index.

**get_value**(**idx*, ***kwargs*)
Return the current value of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The current value of the parameter in the provided index.

**shape**
The shape of the table of parameters.

**Type** Returns

**update**(**idx*, ***kwargs*)
Updates the number of visit of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter whose number of visits has to be updated.

**class** mushroom.utils.variance_parameters.**WindowedVarianceParameter**(*value*, *exponential=False*, *min_value=None*, *tol=1.0*, *window=100*, *size=(1, )*)
Bases: *mushroom.utils.parameters.Parameter*

**__init__**(*value*, *exponential=False*, *min_value=None*, *tol=1.0*, *window=100*, *size=(1, )*)
Constructor.

**Parameters**

- **value** (*float*) – initial value of the parameter;
- **min_value** (*float, None*) – minimum value that the parameter can reach when decreasing;
- **max_value** (*float, None*) – maximum value that the parameter can reach when increasing;
- **size** (*tuple, (1,)*) – shape of the matrix of parameters; this shape can be used to have a single parameter for each state or state-action tuple.

**_compute**(**idx*, ***kwargs*)
Returns: The value of the parameter in the provided index.

**update**(**idx*, ***kwargs*)
Updates the number of visit of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter whose number of visits has to be updated.

**__call__**(**idx*, ***kwargs*)
Update and return the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The updated parameter in the provided index.

**get_value**(**idx*, ***kwargs*)
Return the current value of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The current value of the parameter in the provided index.

**shape**
The shape of the table of parameters.

**Type** Returns

**class** mushroom.utils.variance_parameters.**WindowedVarianceIncreasingParameter**(*value*, *exponential=False*, *min_value=None*, *tol=1.0*, *window=100*, *size=(1, )*)

Bases: *mushroom.utils.variance_parameters.WindowedVarianceParameter*

**__call__**(**idx*, ***kwargs*)

Update and return the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The updated parameter in the provided index.

**__init__**(*value*, *exponential=False*, *min_value=None*, *tol=1.0*, *window=100*, *size=(1, )*)

Constructor.

**Parameters**

- **value** (*float*) – initial value of the parameter;
- **min_value** (*float, None*) – minimum value that the parameter can reach when decreasing;
- **max_value** (*float, None*) – maximum value that the parameter can reach when increasing;
- **size** (*tuple, (1,)*) – shape of the matrix of parameters; this shape can be used to have a single parameter for each state or state-action tuple.

**_compute**(**idx*, ***kwargs*)

Returns: The value of the parameter in the provided index.

**get_value**(**idx*, ***kwargs*)

Return the current value of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter to return.

**Returns** The current value of the parameter in the provided index.

**shape**

The shape of the table of parameters.

**Type** Returns

**update**(**idx*, ***kwargs*)

Updates the number of visit of the parameter in the provided index.

**Parameters** ***idx** (*list*) – index of the parameter whose number of visits has to be updated.

### Viewer

**class** mushroom.utils.viewer.**ImageViewer**(*size*, *dt*)

Bases: object

Interface to pygame for visualizing plain images. Used in mujoco.py.

**__init__**(*size*, *dt*)
Constructor.

**Parameters**

- **size** (*[list, tuple]*) – size of the displayed image;
- **dt** (*float*) – duration of a control step.

**display**(*img*)
Display given frame.

**Parameters** **img** – image to display.

**class** mushroom.utils.viewer.**Viewer**(*env_width*, *env_height*, *width=500*, *height=500*, *background=(0, 0, 0)*)
Bases: object

Interface to pygame for visualizing mushroom native environments.

**__init__**(*env_width*, *env_height*, *width=500*, *height=500*, *background=(0, 0, 0)*)
Constructor.

**Parameters**

- **env_width** (*int*) – The x dimension limit of the desired environment;
- **env_height** (*int*) – The y dimension limit of the desired environment;
- **width** (*int, 500*) – width of the environment window;
- **height** (*int, 500*) – height of the environment window;
- **background** (*tuple, (0, 0, 0)*) – background color of the screen.

**screen**
Property.

**Returns** The screen created by this viewer.

**size**
Property.

**Returns** The size of the screen.

**line**(*start*, *end*, *color=(255, 255, 255)*, *width=1*)
Draw a line on the screen.

**Parameters**

- **start** (*np.ndarray*) – starting point of the line;
- **end** (*np.ndarray*) – end point of the line;
- **color** (*tuple (255, 255, 255)*) – color of the line;
- **width** (*int, 1*) – width of the line.

**square**(*center*, *angle*, *edge*, *color=(255, 255, 255)*, *width=0*)
Draw a square on the screen and apply a roto-translation to it.

**Parameters**

- **center** (*np.ndarray*) – the center of the polygon;
- **angle** (*float*) – the rotation to apply to the polygon;
- **edge** (*float*) – length of an edge;

- **color** (*tuple, (255, 255, 255)*) – the color of the polygon;
- **width** (*int, 0*) – the width of the polygon line, 0 to fill the polygon.

**polygon**(*center*, *angle*, *points*, *color=(255*, *255*, *255)*, *width=0*)
Draw a polygon on the screen and apply a roto-translation to it.

**Parameters**

- **center** (*np.ndarray*) – the center of the polygon;
- **angle** (*float*) – the rotation to apply to the polygon;
- **points** (*list*) – the points of the polygon w.r.t. the center;
- **color** (*tuple, (255, 255, 255)*) – the color of the polygon;
- **width** (*int, 0*) – the width of the polygon line, 0 to fill the polygon.

**circle**(*center*, *radius*, *color=(255*, *255*, *255)*, *width=0*)
Draw a circle on the screen.

**Parameters**

- **center** (*np.ndarray*) – the center of the circle;
- **radius** (*float*) – the radius of the circle;
- **color** (*tuple, (255, 255, 255)*) – the color of the circle;
- **width** (*int, 0*) – the width of the circle line, 0 to fill the circle.

**torque_arrow**(*center*, *torque*, *max_torque*, *max_radius*, *color=(255*, *255*, *255)*, *width=1*)
Draw a torque arrow, i.e. a circular arrow representing a torque. The radius of the arrow is directly proportional to the torque value.

**Parameters**

- **center** (*np.ndarray*) – the point where the torque is applied;
- **torque** (*float*) – the applied torque value;
- **max_torque** (*float*) – the maximum torque value;
- **max_radius** (*float*) – the radius to use for the maximum torque;
- **color** (*tuple, (255, 255, 255)*) – the color of the arrow;
- **width** (*int, 1*) – the width of the torque arrow.

**arrow_head**(*center*, *scale*, *angle*, *color=(255*, *255*, *255)*)
Draw an harrow head.

**Parameters**

- **center** (*np.ndarray*) – the position of the arrow head;
- **scale** (*float*) – scale of the arrow, correspond to the length;
- **angle** (*float*) – the angle of rotation of the angle head;
- **color** (*tuple, (255, 255, 255)*) – the color of the arrow.

**background_image**(*img*)
Use the given image as background for the window, rescaling it appropriately.

**Parameters** **img** – the image to be used.

**display**(*s*)
Display current frame and initialize the next frame to the background color.

**Parameters** **s** – time to wait in visualization.

**close**()
Close the viewer, destroy the window.

## 3.2 Tutorials

### 3.2.1 How to make a simple experiment

The main purpose of Mushroom is to simplify the scripting of RL experiments. A standard example of a script to run an experiment in Mushroom, consists of:

- an **initial part** where the setting of the experiment are specified;
- a **middle part** where the experiment is run;
- a **final part** where operations like evaluation, plot and save can be done.

A RL experiment consists of:

- a **MDP**;
- an **agent**;
- a **core**.

A **MDP** is the problem to be solved by the agent. It contains the function to move the agent in the environment according to the provided action. The MDP can be simply created with:

```
import numpy as np
from sklearn.ensemble import ExtraTreesRegressor

from mushroom.algorithms.value import FQI
from mushroom.core import Core
from mushroom.environments import CarOnHill
from mushroom.policy import EpsGreedy
from mushroom.utils.dataset import compute_J
from mushroom.utils.parameters import Parameter

mdp = CarOnHill()
```

A Mushroom **agent** is the algorithm that is run to learn in the MDP. It consists of a policy approximator and of the methods to improve the policy during the learning. It also contains the features to extract in the case of MDP with continuous state and action spaces. An agent can be defined this way:

```
# Policy
epsilon = Parameter(value=1.)
pi = EpsGreedy(epsilon=epsilon)

# Approximator
approximator_params = dict(input_shape=mdp.info.observation_space.shape,
                           n_actions=mdp.info.action_space.n,
                           n_estimators=50,
                           min_samples_split=5,
                           min_samples_leaf=2)
```

(continues on next page)

(continued from previous page)

```
approximator = ExtraTreesRegressor

# Agent
agent = FQI(approximator, pi, mdp.info, n_iterations=20,
            approximator_params=approximator_params)
```

This piece of code creates the policy followed by the agent (e.g. $\epsilon$-greedy) with $\varepsilon = 1$. Then, the policy approximator is created specifying the parameters to create it and the class (in this case, the `ExtraTreesRegressor` class of scikit-learn is used). Eventually, the agent is created calling the algorithm class and providing the approximator and the policy, together with parameters used by the algorithm.

To run the experiment, the **core** module has to be used. This module requires the agent and the MDP object and contains the function to learn in the MDP and evaluate the learned policy. It can be created with:

```
core = Core(agent, mdp)
```

Once the core has been created, the agent can be trained collecting a dataset and fitting the policy:

```
core.learn(n_episodes=1000, n_episodes_per_fit=1000)
```

In this case, the agent's policy is fitted only once, after that 1000 episodes have been collected. This is a common practice in batch RL algorithms such as `FQI` where, initially, samples are randomly collected and then the policy is fitted using the whole dataset of collected samples.

Eventually, some operations to evaluate the learned policy can be done. This way the user can, for instance, compute the performance of the agent through the collected rewards during an evaluation run. Fixing $\varepsilon = 0$, the greedy policy is applied starting from the provided initial states, then the average cumulative discounted reward is returned.

```
pi.set_epsilon(Parameter(0.))
initial_state = np.array([[-.5, 0.]])
dataset = core.evaluate(initial_states=initial_state)

print(compute_J(dataset, gamma=mdp.info.gamma))
```

### 3.2.2 How to make an advanced experiment

Continuous MDPs are a challenging class of problems to solve in RL. In these problems, a tabular regressor is not enough to approximate the Q-function, since there are an infinite number of states/actions. The solution to solve them is to use a function approximator (e.g. neural network) fed with the raw values of states and actions. In the case a linear approximator is used, it is convenient to enlarge the input space with the space of non-linear **features** extracted from the raw values. This way, the linear approximator is often able to solve the MDPs, despite its simplicity. Many RL algorithms rely on the use of a linear approximator to solve a MDP, therefore the use of features is very important. This tutorial shows how to solve a continuous MDP in Mushroom using an algorithm that requires the use of a linear approximator.

Initially, the MDP and the policy are created:

```
import numpy as np

from mushroom.algorithms.value import SARSALambdaContinuous
from mushroom.approximators.parametric import LinearApproximator
from mushroom.core import Core
from mushroom.environments import *
from mushroom.features import Features
```

(continues on next page)

(continued from previous page)

```
from mushroom.features.tiles import Tiles
from mushroom.policy import EpsGreedy
from mushroom.utils.callbacks import CollectDataset
from mushroom.utils.parameters import Parameter


# MDP
mdp = Gym(name='MountainCar-v0', horizon=np.inf, gamma=1.)

# Policy
epsilon = Parameter(value=0.)
pi = EpsGreedy(epsilon=epsilon)
```

This is an environment created with the Mushroom interface to the OpenAI Gym library. Each environment offered by OpenAI Gym can be created this way simply providing the corresponding id in the `name` parameter, except for the Atari that are managed by a separate class. After the creation of the MDP, the tiles features are created:

```
# Q-function approximator
n_tilings = 10
tilings = Tiles.generate(n_tilings, [10, 10],
                         mdp.info.observation_space.low,
                         mdp.info.observation_space.high)
features = Features(tilings=tilings)

approximator_params = dict(input_shape=(features.size,),
                           output_shape=(mdp.info.action_space.n,),
                           n_actions=mdp.info.action_space.n)
```

In this example, we use sparse coding by means of **tiles** features. The `generate` method generates `n_tilings` grids of 10x10 tilings evenly spaced (the way the tilings are created is explained in *"Reinforcement Learning: An Introduction", Sutton & Barto, 1998*). Eventually, the grid is passed to the `Features` factory method that returns the features class.

Mushroom offers other type of features such a **radial basis functions** and **polynomial** features. The former have also a faster implementation written in Tensorflow that can be used transparently.

Then, the agent is created as usual, but this time passing the feature to it. It is important to notice that the learning rate is divided by the number of tilings for the correctness of the update (see *"Reinforcement Learning: An Introduction", Sutton & Barto, 1998* for details). After that, the learning is run as usual:

```
# Agent
learning_rate = Parameter(.1 / n_tilings)

agent = SARSALambdaContinuous(LinearApproximator, pi, mdp.info,
                              approximator_params=approximator_params,
                              learning_rate=learning_rate,
                              lambda_coeff= .9, features=features)

# Algorithm
collect_dataset = CollectDataset()
callbacks = [collect_dataset]
core = Core(agent, mdp, callbacks=callbacks)

# Train
core.learn(n_episodes=100, n_steps_per_fit=1)
```

To visualize the learned policy the rendering method of OpenAI Gym is used. To activate the rendering in the envi-

ronments that supports it, it is necessary to set `render=True`.

```
# Evaluate
core.evaluate(n_episodes=1, render=True)
```

### 3.2.3 How to create a regressor

Mushroom offers a high-level interface to build function regressors. Indeed, it transparently manages regressors for generic functions and Q-function regressors. The user should not care about the low-level implementation of these regressors and should only use the `Regressor` interface. This interface creates a Q-function regressor or a `GenericRegressor` depending on whether the `n_actions` parameter is provided to the constructor or not.

#### Usage of the `Regressor` interface

**When the action space of RL problems is finite and the adopted approach is value-based,** we want to compute the Q-function of each action. In Mushroom, this is possible using:

- a Q-function regressor with a different approximator for each action (`ActionRegressor`);
- a single Q-function regressor with a different output for each action (`QRegressor`).

The `QRegressor` is suggested when the number of discrete actions is high, due to memory reasons.

The user can create create a `QRegressor` or an `ActionRegressor`, setting the `output_shape` parameter of the `Regressor` interface. If it is set to (1,), an `ActionRegressor` is created; otherwise if it is set to the number of discrete actions, a `QRegressor` is created.

#### Example

Initially, the MDP, the policy and the features are created:

```
import numpy as np

from mushroom.algorithms.value import SARSALambdaContinuous
from mushroom.approximators.parametric import LinearApproximator
from mushroom.core import Core
from mushroom.environments import *
from mushroom.features import Features
from mushroom.features.tiles import Tiles
from mushroom.policy import EpsGreedy
from mushroom.utils.callbacks import CollectDataset
from mushroom.utils.parameters import Parameter


# MDP
mdp = Gym(name='MountainCar-v0', horizon=np.inf, gamma=1.)

# Policy
epsilon = Parameter(value=0.)
pi = EpsGreedy(epsilon=epsilon)

# Q-function approximator
n_tilings = 10
tilings = Tiles.generate(n_tilings, [10, 10],
                         mdp.info.observation_space.low,
```

(continues on next page)

(continued from previous page)

```
                            mdp.info.observation_space.high)
features = Features(tilings=tilings)

# Agent
learning_rate = Parameter(.1 / n_tilings)
```

The following snippet, sets the output shape of the regressor to the number of actions, creating a `QRegressor`:

```
approximator_params = dict(input_shape=(features.size,),
                           output_shape=(mdp.info.action_space.n,),
                           n_actions=mdp.info.action_space.n)
```

If you prefer to use an `ActionRegressor`, simply set the number of actions to (1,):

```
approximator_params = dict(input_shape=(features.size,),
                           output_shape=(1,),
                           n_actions=mdp.info.action_space.n)
```

Then, the rest of the code fits the approximator and runs the evaluation rendering the behaviour of the agent:

```
agent = SARSALambdaContinuous(LinearApproximator, pi, mdp.info,
                              approximator_params=approximator_params,
                              learning_rate=learning_rate,
                              lambda_coeff= .9, features=features)

# Algorithm
collect_dataset = CollectDataset()
callbacks = [collect_dataset]
core = Core(agent, mdp, callbacks=callbacks)

# Train
core.learn(n_episodes=100, n_steps_per_fit=1)

# Evaluate
core.evaluate(n_episodes=1, render=True)
```

### Generic regressor

Whenever the `n_actions` parameter is not provided, the `Regressor` interface creates a `GenericRegressor`. This regressor can be used for general purposes and it is more flexible to be used. It is commonly used in policy search algorithms.

### Example

Create a dataset of points distributed on a line with random gaussian noise.

```
import numpy as np
from matplotlib import pyplot as plt

from mushroom.approximators import Regressor
from mushroom.approximators.parametric import LinearApproximator
```

(continues on next page)

(continued from previous page)

```
x = np.arange(10).reshape(-1, 1)

intercept = 10
noise = np.random.randn(10, 1) * 1
y = 2 * x + intercept + noise
```

To fit the intercept, polynomial features of degree 1 are created by hand:

```
phi = np.concatenate((np.ones(10).reshape(-1, 1), x), axis=1)
```

The regressor is then created and fit (note that `n_actions` is not provided):

```
regressor = Regressor(LinearApproximator,
                      input_shape=(2,),
                      output_shape=(1,))

regressor.fit(phi, y)
```

Eventually, the approximated function of the regressor is plotted together with the target points. Moreover, the weights and the gradient in point 5 of the linear approximator are printed.

```
print('Weights: ' + str(regressor.get_weights()))
print('Gradient: ' + str(regressor.diff(np.array([[5.]]))))

plt.scatter(x, y)
plt.plot(x, regressor.predict(phi))
plt.show()
```

# Python Module Index

## m

# Index

## Symbols

## A



## B

## C



## D

## E

## F

## G

## N



## O



## P

## Q



## R

## S

## T



## U



## V

## W