prl Documentation

Author

Feb 12, 2020

Contents:

Our main goal is to build a useful tool for the reinforcement learning researchers.

While using PRL library for building agents and conducting experiments you can focus on a structure of an agent, state transformations, neural networks architecture, action transformations and reward shaping. Time and memory profiling, logging, agent-environment interactions, agent state saving, neural network training, early stopping or training visualization happens automatically behind the scenes. You are also provided with very useful tools for handling training history and preparing training sets for neural networks.

CHAPTER 1

People's Reinforcement Learning (PRL)

1.1 Description

This is a reinforcement learning framework made with research activity in mind. You can read mode about PRL in our introductory blog post, in-depth look into library, documentation or wiki.

1.2 System requirements

- `python 3.6`
- `swig`
- `python3-dev`

We recommend using `virtualenv` for installing project dependencies.

1.3 Installation

- clone the project:

```
git clone git@gitlab.com:opium-sh/prl.git
```

- create and activate a virtualenv for the project (you can skip this step if you are not using virtualenv)

```
virtualenv -p python3.6 your/path && source your/path/bin/activate
```

- install dependencies:

```
pip install -r requirements.txt
```

- install library

```
pip install -e .
```

- run example:

```
cd examples
python cart_pole_example_cross_entropy.py
```

1.4 Citation

If you use PRL in your work or research please cite us as:

Tempczyk, P., Sliwowski, M., Kozakowski, P., Smuda, P., Topolski, B., Nabrdalik, F., & Malisz, T. (2020). opium-sh/prl: First release of Peoples's Reinforcement Learning (PRL). Zenodo. https://doi.org/10.5281/ZENODO.3662113

CHAPTER 2

API documentation

Information on specific functions, classes, and methods.

2.1 prl

2.1.1 prl package

2.1.1.1 Subpackages

prl.agents package

Submodules

prl.agents.agents module

class A2CAdvantage
Bases: *prl.agents.agents.Advantage*

Advantage function from Asynchronous Methods for Deep Reinforcement Learning.

calculate_advantages (*rewards*, *baselines*, *dones*, *discount_factor*)

Return type ndarray

class A2CAgent (*policy_network*, *value_network*, *agent_id='A2C_agent'*)
Bases: *prl.agents.agents.ActorCriticAgent*

Advantage Actor Critic agent.

class ActorCriticAgent (*policy_network*, *value_network*, *advantage*, *agent_id='ActorCritic_agent'*)
Bases: *prl.agents.agents.Agent*

Basic actor-critic agent.

act (*state*)
Makes a step based on current environments state

Parameters **state** (ndarray) – state from the environment.

Return type ndarray

Returns Action to execute on the environment.

id
Agent UUID

train_iteration (*env*, *n_steps=32*, *discount_factor=1.0*)
Performs single training iteration. This method should contain repeatable part of training an agent.

Parameters

- **env** (*EnvironmentABC*) – Environment
- ****kwargs** – Kwargs passed from train() method

class Advantage
Bases: *prl.typing.AdvantageABC*, abc.ABC

Base class for advantage functions.

calculate_advantages (*rewards*, *baselines*, *dones*, *discount_factor*)

Return type ndarray

class Agent
Bases: *prl.typing.AgentABC*, abc.ABC

Base class for all agents

act (*state*)
Makes a step based on current environments state

Parameters **state** (ndarray) – state from the environment.

Return type ndarray

Returns Action to execute on the environment.

id
Agent UUID

Return type str

play_episodes (*env*, *episodes*)
Method for playing full episodes used usually to train agents.

Parameters

- **env** (*Environment*) – Environment
- **episodes** (int) – Number of episodes to play.

Return type *History*

Returns History object representing episodes history

play_steps (*env*, *n_steps*, *storage*)
Method for performing some number of steps in the environments. Appends new states to existing storage :type env: *Environment* :param env: Environment :type n_steps: int :param n_steps: Number of steps to play :type storage: *Storage* :param storage: Storage (Memory, History) of the earlier games (used to perform first action)

Return type *Storage*

Returns History with appended states, actions, rewards, etc

post_train_cleanup(*env*, ***kwargs*)
Performs cleaning up fields that are no longer needed after training to keep agent lightweight.

Parameters

- **env** (*Environment*) – Environment
- ****kwargs** – Kwargs passed from train() method

pre_train_setup(*env*, ***kwargs*)
Performs pre-training setup. This method should handle non-repeatable part of training an agent.

Parameters

- **env** (*Environment*) – Environment
- ****kwargs** – Kwargs passed from train() method

test(*env*)
Method for playing full episode used to test agents. Reward in the returned history is the true reward from the environments. This method is used mostly for testing the agent.

Parameters **env** – Environment

Return type *History*

Returns History object representing episode history

train(*env*, *n_iterations*, *callback_list=None*, ***kwargs*)
Trains the agent using environment. Also handles callbacks during training.

Parameters

- **env** (*Environment*) – Environment to train on
- **n_iterations** (int) – Maximum number of iterations to train
- **callback_list** (Optional[list]) – List of callbacks
- **kwargs** – other arguments passed to *train_iteration*, *pre_train_setup* and *post_train_cleanup*

train_iteration(*env*, ***kwargs*)
Performs single training iteration. This method should contain repeatable part of training an agent.

Parameters

- **env** (*Environment*) – Environment
- ****kwargs** – Kwargs passed from train() method

class CrossEntropyAgent(*policy_network*, *agent_id='crossentropy_agent'*)
Bases: *prl.agents.agents.Agent*

Agent using cross entropy algorithm

act(*state*)
Makes a step based on current environments state

Parameters **state** (ndarray) – state from the environment.

Return type ndarray

Returns Action to execute on the environment.

id
Agent UUID

train_iteration(*env*, *n_episodes=32*, *percentile=75*)
Performs single training iteration. This method should contain repeatable part of training an agent.

Parameters

- **env** (*EnvironmentABC*) – Environment
- ****kwargs** – Kwargs passed from train() method

class DQNAgent(*q_network*, *replay_buffer_size=10000*, *start_epsilon=1.0*, *end_epsilon=0.05*, *epsilon_decay=1000*, *training_set_size=64*, *target_network_copy_iter=100*, *steps_between_training=10*, *agent_id='DQN_agent'*)
Bases: *prl.agents.agents.Agent*

Agent using DQN algorithm

act(*state*)
Makes a step based on current environments state

Parameters **state** (ndarray) – state from the environment.

Return type ndarray

Returns Action to execute on the environment.

id
Agent UUID

pre_train_setup(*env*, *discount_factor=1.0*, ***kwargs*)
Performs pre-training setup. This method should handle non-repeatable part of training an agent.

Parameters

- **env** (*EnvironmentABC*) – Environment
- ****kwargs** – Kwargs passed from train() method

train_iteration(*env*, *discount_factor=1.0*)
Performs single training iteration. This method should contain repeatable part of training an agent.

Parameters

- **env** (*EnvironmentABC*) – Environment
- ****kwargs** – Kwargs passed from train() method

class GAEAdvantage(*lambda_*)
Bases: *prl.agents.agents.Advantage*

Advantage function from High-Dimensional Continuous Control Using Generalized Advantage Estimation.

calculate_advantages(*rewards*, *baselines*, *dones*, *discount_factor*)

Return type ndarray

class REINFORCEAgent(*policy_network*, *agent_id='REINFORCE_agent'*)
Bases: *prl.agents.agents.Agent*

Agent using REINFORCE algorithm

act(*state*)
Makes a step based on current environments state

Parameters **state** (ndarray) – state from the environment.

Return type ndarray

Returns Action to execute on the environment.

id
Agent UUID

pre_train_setup(*env*, *discount_factor=1.0*, ***kwargs*)
Performs pre-training setup. This method should handle non-repeatable part of training an agent.

Parameters

- **env** (*EnvironmentABC*) – Environment
- ****kwargs** – Kwargs passed from train() method

train_iteration(*env*, *n_episodes=32*, *discount_factor=1.0*)
Performs single training iteration. This method should contain repeatable part of training an agent.

Parameters

- **env** (*EnvironmentABC*) – Environment
- ****kwargs** – Kwargs passed from train() method

class RandomAgent(*agent_id='random_agent'*, *replay_buffer_size=100*)
Bases: *prl.agents.agents.Agent*

Agent performing random actions

act(*state*)
Makes a step based on current environments state

Parameters **state** (ndarray) – state from the environment.

Returns Action to execute on the environment.

id
Agent UUID

pre_train_setup(*env*, ***kwargs*)
Performs pre-training setup. This method should handle non-repeatable part of training an agent.

Parameters

- **env** (*Environment*) – Environment
- ****kwargs** – Kwargs passed from train() method

train_iteration(*env*, *discount_factor=1.0*)
Performs single training iteration. This method should contain repeatable part of training an agent.

Parameters

- **env** (*Environment*) – Environment
- ****kwargs** – Kwargs passed from train() method

Module contents

prl.callbacks package

Submodules

prl.callbacks.callbacks module

class AgentCallback
Bases: *prl.typing.AgentCallbackABC*

Interface for Callbacks defining actions that are executed automatically during different phases of agent training.

on_iteration_end(*agent*)
Method called at the end of every iteration in *prl.base.Agent.train* method.

Parameters **agent** (*AgentABC*) – Agent in which this callback is called.

Return type bool

Returns True if training should be interrupted, False otherwise

on_training_begin(*agent*)
Method called after *prl.base.Agent.pre_train_setup*.

Parameters **agent** (*AgentABC*) – Agent in which this callback is called

on_training_end(*agent*)
Method called after *prl.base.Agent.post_train_cleanup*.

Parameters **agent** (*AgentABC*) – Agent in which this callback is called.

class BaseAgentCheckpoint(*target_path*, *save_best_only=True*, *iteration_interval=1*, *number_of_test_runs=1*)
Bases: *prl.callbacks.callbacks.AgentCallback*

Saving agents during training. This is a base class that implements only logic. One should use classes with saving method matching networks' framework. For more info on methods see base class.

Parameters

- **target_path** (str) – Directory in which agents will be saved. Must exist before
- **this callback.** (*creating*) –
- **save_best_only** (bool) – Whether to save all models, or only the one with highest reward.
- **iteration_interval** (int) – Interval between calculating test reward. Using low values may make training process slower
- **number_of_test_runs** (int) – Number of test runs when calculating reward. Higher value averages variance out, but makes training longer.

on_iteration_end(*agent*)
Method called at the end of every iteration in *prl.base.Agent.train* method.

Parameters **agent** (*AgentABC*) – Agent in which this callback is called.

Returns True if training should be interrupted, False otherwise

on_training_end(*agent*)
Method called after *prl.base.Agent.post_train_cleanup*.

Parameters **agent** (*AgentABC*) – Agent in which this callback is called.

class CallbackHandler(*callback_list*, *env*)
Bases: object

Callback that handles all given handles. Calls appropriate methods on each callback and aggregates break codes. For more info on methods see base class.

static check_run_condition(*current_count*, *interval*)

on_iteration_end(*agent*)

on_training_begin(*agent*)

on_training_end(*agent*)

run_tests(*agent*)

Return type *HistoryABC*

setup_callbacks()
Sets up callbacks. This calculates optimal intervals for calling callbacks, and for calling testing procedure.

class EarlyStopping(*target_reward*, *iteration_interval=1*, *number_of_test_runs=1*, *verbose=1*)
Bases: *prl.callbacks.callbacks.AgentCallback*

Implements EarlyStopping for RL Agents. Training is stopped after reaching given target reward.

Parameters

- **target_reward** (float) – Target reward.
- **iteration_interval** (int) – Interval between calculating test reward. Using low values may make training process slower.
- **number_of_test_runs** (int) – Number of test runs when calculating reward. Higher value averages variance out, but makes training longer.
- **verbose** (int) – Whether to print message after stopping training (1), or not (0).

Note: By reward, we mean here untransformed reward given by *Agent.test* method. For more info on methods see base class.

---

on_iteration_end(*agent*)
Method called at the end of every iteration in *prl.base.Agent.train* method.

Parameters **agent** (*AgentABC*) – Agent in which this callback is called.

Returns True if training should be interrupted, False otherwise

class PyTorchAgentCheckpoint(*target_path*, *save_best_only=True*, *iteration_interval=1*, *number_of_test_runs=1*)
Bases: *prl.callbacks.callbacks.BaseAgentCheckpoint*

Class for saving PyTorch-based agents. For more details, see parent class.

class TensorboardLogger(*file_path='logs_1581542124'*, *iteration_interval=1*, *number_of_test_runs=1*, *show_time_logs=False*)
Bases: *prl.callbacks.callbacks.AgentCallback*

Writes various information to tensorboard during training. For more info on methods see base class.

Parameters

- **file_path** (str) – Path to file with output.
- **iteration_interval** (int) – Interval between calculating test reward. Using low values may make training process slower.
- **number_of_test_runs** (int) – Number of test runs when calculating reward. Higher value averages variance out, but makes training longer.
- **show_time_logs** (bool) – If shows logs from time_logger.

on_iteration_end(*agent*)
Method called at the end of every iteration in *prl.base.Agent.train* method.

Parameters **agent** (*AgentABC*) – Agent in which this callback is called.

Returns True if training should be interrupted, False otherwise

on_training_end(*agent*)
Method called after *prl.base.Agent.post_train_cleanup*.

Parameters **agent** (*AgentABC*) – Agent in which this callback is called.

class TrainingLogger(*on_screen=True*, *to_file=False*, *file_path=None*, *iteration_interval=1*)
Bases: *prl.callbacks.callbacks.AgentCallback*

Logs training information after certain amount of iterations. Data may appear in output, or be written into a file. For more info on methods see base class.

Parameters

- **on_screen** (bool) – Whether to show info in output.
- **to_file** (bool) – Whether to save info into a file.
- **file_path** (Optional[str]) – Path to file with output.
- **iteration_interval** (int) – How often should info be logged on screen. File output remains logged every iteration.

on_iteration_end(*agent*)
Method called at the end of every iteration in *prl.base.Agent.train* method.

Parameters **agent** (*AgentABC*) – Agent in which this callback is called.

Returns True if training should be interrupted, False otherwise

class ValidationLogger(*on_screen=True*, *to_file=False*, *file_path=None*, *iteration_interval=1*, *number_of_test_runs=3*)
Bases: *prl.callbacks.callbacks.AgentCallback*

Logs validation information after certain amount of iterations. Data may appear in output, or be written into a file. For more info on methods see base class.

Parameters

- **on_screen** (bool) – Whether to show info in output.
- **to_file** (bool) – Whether to save info into a file.
- **file_path** (Optional[str]) – Path to file with output.
- **iteration_interval** (int) – How often should info be logged on screen. File output
- **logged every iteration.** (*remains*) –
- **number_of_test_runs** (int) – Number of played episodes in history's summary logs.

on_iteration_end(*agent*)
Method called at the end of every iteration in *prl.base.Agent.train* method.

Parameters **agent** (*AgentABC*) – Agent in which this callback is called.

Returns True if training should be interrupted, False otherwise

Module contents

prl.environments package

Submodules

prl.environments.environments module

class Environment(*env*, *environment_id='Environment_wrapper'*, *state_transformer=<prl.transformers.state_transformers.NoO*
object>, *reward_transformer=<prl.transformers.reward_transformers.NoOpRewardTransformer*
object>, *action_transformer=<prl.transformers.action_transformers.NoOpActionTransformer*
object>, *expected_episode_length=512*, *dump_history=False*)

Bases: *prl.typing.EnvironmentABC*, abc.ABC

Interface for wrappers for gym-like environments. It can use *StateTransformer* and *RewardTransformer* to shape states and rewards to a convenient form for the agent. It can also use *ActionTransformer* to change representation from the suitable to the agent to the required by the environments.

Environment also keeps the history of current episode, so it doesn't have to be implemented on the agent side. All the transformers can use this history to transform states, actions and rewards.

Parameters

- **env** (Env) – Environment with gym like API
- **environment_id** (str) – ID of the env
- **state_transformer** (*StateTransformerABC*) – Object of the class *StateTransformer*
- **reward_transformer** (*RewardTransformerABC*) – Object of the class *RewardTransformer*
- **action_transformer** (*ActionTransformerABC*) – Object of the class *ActionTransformer*

action_space

action_space object from the *action_transformer*

Type Returns

Return type Space

action_transformer

Action transformers can be used to change the representation of actions like changing the coordinate system or feeding only a difference from the last action for continuous action space. ActionTransformer is used to change representation from the suitable to the agent to the required by the wrapped environments.

Return type *ActionTransformerABC*

Returns *ActionTransformer* object

close()

Cleans up and closes the environment

id

Environment UUID

observation_space

observation_space object from the *state_transformer*

Type Returns

Return type `Space`

reset()
Resets the environments to initial state and returns this initial state.

Return type `ndarray`

Returns New state

reward_transformer
Reward transformer object for reward shaping like taking the sign of the original reward or adding reward for staying on track in a car racing game.

Return type *RewardTransformerABC*

Returns *RewardTransformer* object

state_history
Current episode history

Type Returns

Return type *HistoryABC*

state_transformer
StateTransformer object for state transformations. It can be used for changing representation of the state. For example it can be used for simply subtracting constant vector from the state, stacking the last N states or transforming image into compressed representation using autoencoder.

Return type *StateTransformer*

Returns *StateTransformer* object

step(*action*)
Transform and perform a given action in the wrapped environment. Returns transformed states and rewards from wrapped environment.

Parameters **action** (`ndarray`) – Action executed by the agent.

Returns New state reward: Reward we get from performing the action is done: Is the simulation finished info: Additional diagnostic information

Return type observation

Note: When true_reward flag is set to True it returns non-transformed reward for the testing purposes.

---

class FrameSkipEnvironment(*env*, *environment_id='frameskip_gym_environment_wrapper'*, *state_transformer=<prl.transformers.state_transformers.NoOpStateTransformer object>*, *reward_transformer=<prl.transformers.reward_transformers.NoOpRewardTransform object>*, *action_transformer=<prl.transformers.action_transformers.NoOpActionTransformer object>*, *expected_episode_length=512*, *n_skip_frames=0*, *cumulative_reward=False*)
Bases: *prl.environments.environments.Environment*

Environment wrapper skipping frames from original environment. Action executed by the agent is repeated on the skipped frames.

Parameters

- **env** (`Env`) – Environment with gym like API
- **environment_id** (`str`) – ID of the env

- **state_transformer** (*StateTransformer*) – Object of the class StateTransformer
- **reward_transformer** (*RewardTransformer*) – Object of the class RewardTransformer
- **action_transformer** (*ActionTransformer*) – Object of the class ActionTransformer
- **n_skip_frames** (int) – Number of frames to skip on each step.
- **cumulative_reward** – If True, reward returned from step() method is cumulative reward from the skipped steps.

step(*action*)

Transform and perform a given action in the wrapped environment. Returns transformed states and rewards from wrapped environment.

Parameters **action** (ndarray) – Action executed by the agent.

Returns New state reward: Reward we get from performing the action is done: Is the simulation finished info: Additional diagnostic information

Return type observation

Note: When true_reward flag is set to True it returns non-transformed reward for the testing purposes.

class TimeShiftEnvironment(*env*, *environment_id='timeshift_gym_environment_wrapper'*, *state_transformer=<prl.transformers.state_transformers.NoOpStateTransformer object>*, *reward_transformer=<prl.transformers.reward_transformers.NoOpRewardTransform object>*, *action_transformer=<prl.transformers.action_transformers.NoOpActionTransformer object>*, *expected_episode_length=512*, *lag=1*)

Bases: *prl.environments.environments.Environment*

Environment wrapper creating lag between action passed to step() method by the agent and action execution in the environment. First 'lag' actions are sampled from action_space.

Parameters

- **env** (Env) – Environment with gym like API
- **environment_id** (str) – ID of the env
- **state_transformer** (*StateTransformer*) – Object of the class StateTransformer
- **reward_transformer** (*RewardTransformer*) – Object of the class RewardTransformer
- **action_transformer** (*ActionTransformer*) – Object of the class ActionTransformer (don't use - not implemented action transformation)

Note: Class doesn't have implemented action transformation.

reset()

Resets the environments to initial state and returns this initial state.

Return type ndarray

Returns New state

step(*action*)
Transform and perform a given action in the wrapped environment. Returns transformed states and rewards from wrapped environment.

Parameters **action** (ndarray) – Action executed by the agent.

Returns New state reward: Reward we get from performing the action is done: Is the simulation finished info: Additional diagnostic information

Return type observation

Note: When true_reward flag is set to True it returns non-transformed reward for the testing purposes.

class TransformedSpace(*shape=None*, *dtype=None*, *transformed_state=None*)
Bases: gym.core.Space

Class created to handle Environments using StateTransformers as the observation space is not directly specified in such a system.

contains(*state*)
This method is not available as TransformedSpace object can't estimate whether *x* is contained by the state representation. It is caused because TransformedSpace object infers the state properties.

sample()
Return sample state. Object of this class returns always the same object. It needs to be created every sample. When used inside Environment with StateTransformer every call of property *observation_space* cause the initialization of new object, so another sample is returned.

Returns Transformed state

Module contents

prl.function_approximators package

Submodules

prl.function_approximators.function_approximators module

class FunctionApproximator
Bases: *prl.typing.FunctionApproximatorABC*, abc.ABC

Class for function approximators used by the agents. For example it could be a neural network for value function or policy approximation.

id
Function Approximator UUID

Return type str

predict(*x*)
Makes prediction based on input

train(*x*, **loss_args*)
Trains FA for one or more steps. Returns training loss value.

Return type float

prl.function_approximators.pytorch_nn module

class DQNLoss(*mode='huber'*, *size_average=None*, *reduce=None*, *reduction='mean'*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*nn_outputs*, *actions*, *target_outputs*)

class PolicyGradientLoss(*size_average=None*, *reduce=None*, *reduction='mean'*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*nn_outputs*, *actions*, *returns*)

class PytorchConv(*x_shape*, *hidden_sizes*, *y_size*)
Bases: *prl.function_approximators.pytorch_nn.PytorchNet*

forward(*x*)
Defines the computation performed at every training step.

Parameters **x** – input data

Returns network output

predict(*x*)
Makes prediction based on input data.

Parameters **x** – input data

Returns prediction for agent.act(x) method

class PytorchFA(*net*, *loss*, *optimizer*, *device='cpu'*, *batch_size=64*, *last_batch=True*, *network_id='pytorch_nn'*)
Bases: *prl.function_approximators.function_approximators.FunctionApproximator*

Class for pytorch based neural networks function approximators.

Parameters

- **net** (*PytorchNet*) – PytorchNet class neural network
- **loss** (*<sphinx.ext.autodoc.importer._MockObject object at 0x7fdfe23b0cf8>*) – loss function
- **optimizer** (*<sphinx.ext.autodoc.importer._MockObject object at 0x7fdfe2df33c8>*) – optimizer
- **device** (str) – device for computation: “cpu” or “cuda”
- **batch_size** (int) – size of a training batch
- **last_batch** (bool) – flag if the last batch (usually shorter than batch_size) is going to be feed into network
- **network_id** (str) – name of the network for debugging and logging purposes

convert_to_pytorch(*y*)

id
Function Approximator UUID

predict(*x*)
Makes prediction

train(*x*, **loss_args*)
Trains network on a dataset

Parameters

- **x** (ndarray) – input array for the network
- ***loss_args** – arguments passed directly to loss function

class PytorchMLP(*x_shape*, *y_size*, *output_activation*, *hidden_sizes*)

Bases: *prl.function_approximators.pytorch_nn.PytorchNet*

forward(*x*)

Defines the computation performed at every training step.

Parameters **x** – input data

Returns network output

predict(*x*)

Makes prediction based on input data.

Parameters **x** – input data

Returns prediction for agent.act(x) method

class PytorchNet(**args*, ***kwargs*)

Bases: *prl.typing.PytorchNetABC*

Neural networks for PytorchFA. It has separate predict method strictly for Agent.act() method, wchich can act differently than forward() method.

Note: This class has two abstract methods that need to be implemented (listed above).

---

forward(*x*)

Defines the computation performed at every training step.

Parameters **x** (*<sphinx.ext.autodoc.importer._MockObject object at 0x7fdfe2d86cf8>*) – input data

Returns network output

predict(*x*)

Makes prediction based on input data.

Parameters **x** (*<sphinx.ext.autodoc.importer._MockObject object at 0x7fdfe25e9d30>*) – input data

Returns prediction for agent.act(x) method

Module contents

prl.storage package

Submodules

prl.storage.storage module

class History(*initial_state*, *action_type*, *initial_length=512*)

Bases: *prl.storage.storage.Storage*, *prl.typing.HistoryABC*

An object which is used to keep the episodes history (used within *Environment* class and by some agents). Agent can use this object to keep history of past episodes, calculate returns, total rewards, etc. and sample batches from it.

Object also supports indexing and slicing because it supports python Sequence protocol, so functions working on sequences like random.choice can be also used on history.

Parameters

- **initial_state** (ndarray) – initial state from enviroment
- **action_type** (type) – numpy type of action (e.g. np.int32)
- **initial_length** (int) – initial length of a history

get_actions()
Returns an array of all actions.

Return type ndarray

Returns array of all actions

get_dones()
Returns an array of all done flags.

Return type ndarray

Returns array of all done flags

get_last_state()
Returns only the last state.

Return type ndarray

Returns last state

get_number_of_episodes()
Returns a number of full episodes in history.

Return type int

Returns number of full episodes in history

get_returns(*discount_factor=1.0*, *horizon=inf*)
Calculates returns for each step.

Return type ndarray

Returns array of discounted returns for each step

get_rewards()
Returns an array of all rewards.

Return type ndarray

Returns array of all rewards

get_states()
Returns an array of all states.

Return type ndarray

Returns array of all states

get_summary()

Return type (<class 'float'>, <class 'float'>, <class 'int'>)

get_total_rewards()
Calculates sum of all rewards for each episode and reports it for each state, so every state in one episode has the same value of total reward. This can be useful for filtering states for best episodes (e.g. in Cross Entropy Algorithm).

Return type ndarray

Returns total reward for each state

new_state_update(*state*)
Overwrites newest state in the History

Parameters **state** (ndarray) – state array.

sample_batch(*replay_buffer_size*, *batch_size=64*, *returns=False*, *next_states=False*)
Samples batch of examples from the Storage.

Parameters

- **replay_buffer_size** (int) – length of a replay buffor to sample examples from
- **batch_size** (int) – number of returned examples
- **returns** (bool) – if True, the method will return the returns from each step instead of the rewards
- **next_states** (bool) – if True, the method will return also next states (i.e. for DQN algorithm)

Returns states, actions, rewards, dones, (new_states)

Return type batch of samples from history in form of a tuple with np.ndarrays in order

update(*action*, *reward*, *done*, *state*)
Updates the object with latest states, reward, actions and done flag.

Parameters

- **action** (ndarray) – action executed by the agent
- **reward** (Real) – reward from environments
- **done** (bool) – done flag from environments
- **state** (ndarray) – new state returned by wrapped environments after executing action

class Memory(*initial_state*, *action_type*, *maximum_length=1000*)
Bases: *prl.storage.storage.Storage*, *prl.typing.StorageABC*

An object to be used as replay buffer. Doesn't contain full episodes and acts as limited FIFO queue. Implemented as double size numpy arrays with duplicated data to support very fast slicing and sampling at the cost of higher memory usage.

Parameters

- **initial_state** (ndarray) – initial state from enviroment
- **action_type** – numpy type of action (e.g. np.int32)
- **maximum_length** (int) – maximum number of examples to keep in queue

clear(*initial_state*)

get_actions()
Returns an array of all actions.

Return type ndarray

Returns array of all actions

get_dones()
Returns an array of all done flags.

Return type ndarray

Returns array of all done flags

get_last_state()
Returns only the last state.

Return type ndarray

Returns last state

get_rewards()
Returns an array of all rewards.

Return type ndarray

Returns array of all rewards

get_states(*include_last=False*)
Returns an array of all states.

Return type ndarray

Returns array of all states

new_state_update(*state*)
Overwrites newest state in the History

Parameters **state** – state array.

sample_batch(*replay_buffor_size*, *batch_size=64*, *returns=False*, *next_states=False*)
Samples batch of examples from the Storage.

Parameters

- **replay_buffer_size** – length of a replay buffor to sample examples from
- **batch_size** (int) – number of returned examples
- **returns** (bool) – if True, the method will return the returns from each step instead of the rewards
- **next_states** (bool) – if True, the method will return also next states (i.e. for DQN algorithm)

Returns states, actions, rewards, dones, (new_states)

Return type batch of samples from history in form of a tuple with np.ndarrays in order

update(*action*, *reward*, *done*, *state*)
Updates the object with latest states, reward, actions and done flag.

Parameters

- **action** – action executed by the agent
- **reward** – reward from environments
- **done** – done flag from environments
- **state** – new state returned by wrapped environments after executing action

class Storage

Bases: *prl.typing.StorageABC*, abc.ABC

get_actions()

Returns an array of all actions.

Return type ndarray

Returns array of all actions

get_dones()

Returns an array of all done flags.

Return type ndarray

Returns array of all done flags

get_last_state()

Returns only the last state.

Return type ndarray

Returns last state

get_rewards()

Returns an array of all rewards.

Return type ndarray

Returns array of all rewards

get_states()

Returns an array of all states.

Return type ndarray

Returns array of all states

new_state_update(*state*)

Overwrites newest state in the History

Parameters **state** – state array.

sample_batch(*replay_buffor_size*, *batch_size*, *returns*, *next_states*)

Samples batch of examples from the Storage.

Parameters

- **replay_buffer_size** – length of a replay buffor to sample examples from
- **batch_size** (int) – number of returned examples
- **returns** (bool) – if True, the method will return the returns from each step instead of the rewards
- **next_states** (bool) – if True, the method will return also next states (i.e. for DQN algorithm)

Returns states, actions, rewards, dones, (new_states)

Return type batch of samples from history in form of a tuple with np.ndarrays in order

update(*action*, *reward*, *done*, *state*)

Updates the object with latest states, reward, actions and done flag.

Parameters

- **action** – action executed by the agent

- **reward** – reward from environments
- **done** – done flag from environments
- **state** – new state returned by wrapped environments after executing action

calculate_returns(*all_rewards*, *dones*, *horizon*, *discount_factor*, *_index*)

calculate_total_rewards(*all_rewards*, *dones*, *_index*)

Module contents

prl.transformers package

Submodules

prl.transformers.action_transformers module

class ActionTransformer

Bases: *prl.typing.ActionTransformerABC*, abc.ABC

Interface for raw action (original actions from agent) transformers. Object of this class are used by the classes implementing EnvironmentABC interface. Action transformers can use all episode history from the beginning of the episode up to the moment of transformation.

action_space(*original_space*)

Returns: action_space object of class gym.Space, which defines type and shape of transformed action.

Note: If transformed action is from the same action_space as original state, then action_space is None. Information contained within action_space can be important for agents, so it is important to properly define an action_space.

Return type Space

id

State transformer UUID

Return type str

reset()

Action transformer can be stateful, so it have to be reset after each episode.

transform(*action*, *history*)

Transforms action into another representation, which must be of the form defined by action_space object. Input action can be in a form of numpy array, list, tuple, int, etc.

Parameters

- **action** (ndarray) – Action from the agent
- **history** (*HistoryABC*) – History object of an episode

Return type ndarray

Returns Transformed action in form defined by the action_space object.

class NoOpActionTransformer

Bases: *prl.transformers.action_transformers.ActionTransformer*

ActionTransformer doing nothing

action_space(*original_space*)

Returns: action_space object of class gym.Space, which defines type and shape of transformed action.

Note: If transformed action is from the same action_space as original state, then action_space is None. Information contained within action_space can be important for agents, so it is important to properly define an action_space.

Return type Space

id

State transformer UUID

reset()

Action transformer can be stateful, so it have to be reset after each episode.

transform(*action*, *history*)

Transforms action into another representation, which must be of the form defined by action_space object. Input action can be in a form of numpy array, list, tuple, int, etc.

Parameters

- **action** (ndarray) – Action from the agent
- **history** (*HistoryABC*) – History object of an episode

Return type ndarray

Returns Transformed action in form defined by the action_space object.

prl.transformers.reward_transformers module

class NoOpRewardTransformer

Bases: *prl.transformers.reward_transformers.RewardTransformer*

RewardTransformer doing nothing

id()

Reward transformer UUID

reset()

Reward transformer can be stateful, so it have to be reset after each episode.

transform(*reward*, *history*)

Transforms a reward.

Parameters

- **reward** (Real) – Raw reward from the wrapped environment
- **history** (*HistoryABC*) – History object

Return type Number

Returns Transformed reward

class RewardShiftTransformer(*shift*)
Bases: *prl.transformers.reward_transformers.RewardTransformer*

RewardTransformer shifting reward by some constant value

id()
Reward transformer UUID

reset()
Reward transformer can be stateful, so it have to be reset after each episode.

transform(*reward*, *history*)
Transforms a reward.

Parameters

- **reward** (Real) – Raw reward from the wrapped environment
- **history** (*HistoryABC*) – History object

Return type Number

Returns Transformed reward

class RewardTransformer
Bases: *prl.typing.RewardTransformerABC*, abc.ABC

Interface for classes for shaping the raw reward from wrapped environments. Object inherited from this class are used by the Environment class objects. Reward transformers can use all episode history from the beginning of the episode up to the moment of transformation.

id
Reward transformer UUID

Return type str

reset()
Reward transformer can be stateful, so it have to be reset after each episode.

transform(*reward*, *history*)
Transforms a reward.

Parameters

- **reward** (Real) – Raw reward from the wrapped environment
- **history** (*HistoryABC*) – History object

Return type Real

Returns Transformed reward

prl.transformers.state_transformers module

class NoOpStateTransformer
Bases: *prl.transformers.state_transformers.StateTransformer*

StateTransformer doing nothing

id
State transformer UUID

reset()
State transformer can be stateful, so it have to be reset after each episode.

transform(*state*, *history*)
Transforms observed state into another representation, which must be of the form defined by observation_space object. Input state must be in a form of numpy.ndarray.

Parameters

- **state** (ndarray) – State from wrapped environment
- **history** (*HistoryABC*) – History object

Return type ndarray

Returns Transformed state in form defined by the observation_space object.

class PongTransformer(*resize_factor=2*, *crop=True*, *flatten=False*)
Bases: *prl.transformers.state_transformers.StateTransformer*

StateTransformer for Pong atari game

id
State transformer UUID

reset()
State transformer can be stateful, so it have to be reset after each episode.

transform(*observation*, *history*)
Transforms observed state into another representation, which must be of the form defined by observation_space object. Input state must be in a form of numpy.ndarray.

Parameters

- **state** – State from wrapped environment
- **history** (*HistoryABC*) – History object

Return type ndarray

Returns Transformed state in form defined by the observation_space object.

class StateShiftTransformer(*shift_tensor*)
Bases: *prl.transformers.state_transformers.StateTransformer*

StateTransformer shifting reward by some constant vector

id
State transformer UUID

reset()
State transformer can be stateful, so it have to be reset after each episode.

transform(*state*, *history*)
Transforms observed state into another representation, which must be of the form defined by observation_space object. Input state must be in a form of numpy.ndarray.

Parameters

- **state** (ndarray) – State from wrapped environment
- **history** (*HistoryABC*) – History object

Return type ndarray

Returns Transformed state in form defined by the observation_space object.

class StateTransformer
Bases: *prl.typing.StateTransformerABC*, abc.ABC

Interface for raw states (original states from wrapped environments) transformers. Object of this class are used by the classes implementing EnvironmentABC interface. State transformers can use all episode history from the beginning of the episode up to the moment of transformation.

id
State transformer UUID

Return type str

reset()
State transformer can be stateful, so it have to be reset after each episode.

transform(*state*, *history*)
Transforms observed state into another representation, which must be of the form defined by observation_space object. Input state must be in a form of numpy.ndarray.

Parameters

- **state** (ndarray) – State from wrapped environment
- **history** (*HistoryABC*) – History object

Return type ndarray

Returns Transformed state in form defined by the observation_space object.

Module contents

prl.utils package

Submodules

prl.utils.loggers module

class Logger
Bases: object

Class for logging scalar values to limited queues. Logged data send to each client is tracked by the Logger, so each client can ask for unseen data and recieve it.

add(*key*, *value*)
Add a value to queue assigned to key value.

Parameters

- **key** (str) – logged value name
- **value** (Number) – logged number

flush(*consumer_id*)
Method used by clients to recieve only new unseed data from logger.

Parameters **consumer_id** (int) – value returned by register method.

Return type (typing.Dict[str, typing.List], typing.Dict[str, range], typing.Dict[str, typing.List])

Returns dict with new data.

get_data()

Return type Dict[str, deque]

Returns all logged data.

register()
Registers client in order to receive data from Logger object.

Return type int

Returns client ID used to identify client while requesting for a new data.

save(*path*)
Saves data to file.

Parameters **path** (str) – path to the file.

class TimeLogger
Bases: *prl.utils.loggers.Logger*

Storage for measurements of function and methods exectuion time. Used by timeit function/decorator. Can be used to print summary of a time profiling or save all data to generate a plot how execution times are changing during the program execution.

limited_deque()
Auxiliary function for Logger class.

Returns: Deque with maximum length set to DEQUE_MAX_LEN

prl.utils.misc module

class colors
Bases: object

Color codes for unocode strings. Used for output string formatting.

BLUE = '\x1b[94m'

BOLD = '\x1b[1m'

END_FORMAT = '\x1b[0m'

GREEN = '\x1b[92m'

RED = '\x1b[91m'

UNDERLINE = '\x1b[4m'

YELLOW = '\x1b[93m'

prl.utils.utils module

timeit(*func*, *profiled_function_name=None*)
Decorator for profiling execution time for the functions and methods. To measure time of a method or function you have to put @timeit in line nefore function, or wrap a function in the code:

@timeit def func(a, b, c=”1”):

pass

or in the code:

result = timeit(func, profiled_function_name=”Profiled function func”)(5,5)

To print results of measurment you have to print time_logger object from this package at the end of the program execution. When the name of the function can be ambiguous in the profiler data use profiled_function_name parameter.

Parameters

- **func** – function, which execution time we wan to measure
- **profiled_function_name** – user defined name for the wrapped function.

Returns wrapped function

Module contents

2.1.1.2 Submodules

2.1.1.3 prl.typing module

class ActionTransformerABC

Bases: abc.ABC

action_space(*original_space*)

Return type Space

id

Return type str

reset()

transform(*action*, *history*)

Return type ndarray

class AdvantageABC

Bases: abc.ABC

class AgentABC

Bases: abc.ABC

act(*state*)

id

Return type str

play_episodes(*env*, *episodes*)

Return type *HistoryABC*

play_steps(*env*, *n_steps*, *history*)

Return type *HistoryABC*

post_train_cleanup(*env*, ***kwargs*)

pre_train_setup(*env*, ***kwargs*)

test(*env*)

Return type *HistoryABC*

train(*env*, *n_iterations*, *callback_list*, ***kwargs*)

train_iteration(*env*, ***kwargs*)

Return type (<class 'float'>, <class 'prl.typing.HistoryABC'>)

class AgentCallbackABC

Bases: abc.ABC

on_iteration_end(*agent*)

Return type bool

on_training_begin(*agent*)

on_training_end(*agent*)

class EnvironmentABC

Bases: abc.ABC

action_space

Return type Space

action_transformer

Return type *ActionTransformerABC*

close()

id

observation_space

Return type Space

reset()

Return type ndarray

reward_transformer

Return type *RewardTransformerABC*

state_history

Return type *HistoryABC*

state_transformer

Return type *StateTransformerABC*

step(*action*)

Return type Tuple[ndarray, Real, bool, Dict[~KT, ~VT]]

class FunctionApproximatorABC

Bases: abc.ABC

id

Return type str

predict(*x*)

train(*x*, **loss_args*)

Return type float

class HistoryABC

Bases: abc.ABC

get_actions()

Return type ndarray

get_dones()

Return type ndarray

get_last_state()

Return type ndarray

get_number_of_episodes()

Return type int

get_returns(*discount_factor*, *horizon*)

Return type ndarray

get_rewards()

Return type ndarray

get_states()

Return type ndarray

get_summary()

get_total_rewards()

Return type ndarray

new_state_update(*state*)

sample_batch(*replay_buffor_size*, *batch_size*, *returns*, *next_states*)

Return type tuple

update(*action*, *reward*, *done*, *state*)

MemoryABC
alias of *prl.typing.StorageABC*

class PytorchNetABC(**args*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*x*)

predict(*x*)

class RewardTransformerABC
Bases: abc.ABC

id

Return type str

reset()

transform(*reward*, *history*)

Return type Real

class StateTransformerABC
Bases: abc.ABC

id

Return type str

reset()

transform(*state*, *history*)

Return type ndarray

class StorageABC

Bases: abc.ABC

get_actions()

Return type ndarray

get_dones()

Return type ndarray

get_last_state()

Return type ndarray

get_rewards()

Return type ndarray

get_states()

Return type ndarray

new_state_update(*state*)

sample_batch(*replay_buffor_size*, *batch_size*, *returns*, *next_states*)

Return type tuple

update(*action*, *reward*, *done*, *state*)

2.1.1.4 Module contents

Python Module Index

p

Index

G

H

I

L

M

N

O

P

R

S

T

U

V

Y