# Contents

1 **Installation** 3  
  1.1 How to install PFRL 3  

2 **API Reference** 5  
  2.1 Action values 5  
  2.2 Agents 6  
  2.3 Experiments 19  
  2.4 Explorers 24  
  2.5 Modules 25  
  2.6 Policies 28  
  2.7 Q-functions 29  
  2.8 Replay Buffers 33  

3 **Indices and tables** 37  

Index 39
PFRL is a deep reinforcement learning library that implements various state-of-the-art deep reinforcement algorithms in Python using PyTorch.
CHAPTER 1

Installation

1.1 How to install PFRL

PFRL is tested with 3.7.7. For other requirements, see requirements.txt.

Listing 1: requirements.txt

<table>
<thead>
<tr>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>torch&gt;=1.3.0</td>
</tr>
<tr>
<td>gym&gt;=0.9.7</td>
</tr>
<tr>
<td>numpy&gt;=1.10.4</td>
</tr>
<tr>
<td>filelock</td>
</tr>
<tr>
<td>pillow</td>
</tr>
</tbody>
</table>

PFRL can be installed via PyPI,

```
pip install pfrl
```

or through the source code:

```
git clone https://github.com/pfnet/pfrl.git
cd pfrl
python setup.py install
```
CHAPTER 2

API Reference

2.1 Action values

2.1.1 Action value interfaces

class pfrl.action_value.ActionValue
    Struct that holds state-fixed Q-functions and its subproducts.
    Every operation it supports is done in a batch manner.

    evaluate_actions(actions)
        Evaluate Q(s,a) with a = given actions.

    greedy_actions
        Get argmax_a Q(s,a).

    max
        Evaluate max Q(s,a).

    params
        Learnable parameters of this action value.

        Returns tuple of torch.Tensor

2.1.2 Action value implementations

class pfrl.action_value.DiscreteActionValue(q_values, q_values_formatter=<function DiscreteActionValue.<lambda>>)
    Q-function output for discrete action space.

    Parameters q_values (torch.Tensor) – Array of Q values whose shape is (batchsize, n_actions)

class pfrl.action_value.QuadraticActionValue(mu, mat, v, min_action=None, max_action=None)
    Q-function output for continuous action space.
Define a $Q(s,a)$ with $A(s,a)$ in a quadratic form.

$$Q(s,a) = V(s,a) + A(s,a) = -\frac{1}{2} (u - \mu(s))^T P(s) (u - \mu(s))$$

**Parameters**

- $\mu$ (torch.Tensor) – $\mu(s)$, actions that maximize $A(s,a)$
- $\mat$ (torch.Tensor) – $P(s)$, coefficient matrices of $A(s,a)$. It must be positive definite.
- $v$ (torch.Tensor) – $V(s)$, values of $s$
- $\min\_action$ (ndarray) – minimum action, not batched
- $\max\_action$ (ndarray) – maximum action, not batched

```python
class pfrl.action_value.SingleActionValue (evaluator, maximizer=None)
    ActionValue that can evaluate only a single action.
```

## 2.2 Agents

### 2.2.1 Agent interfaces

```python
class pfrl.agent.Agent
    Abstract agent class.

    act (obs: Any) -> Any
        Select an action.

        Returns  action

        Return type  ~object

    get_statistics () -> List[Tuple[str, Any]]
        Get statistics of the agent.

        Returns

        List of two-item tuples. The first item in a tuple is a str that represents the name of item, while the second item is a value to be recorded.

        Example: [('average_loss', 0), ('average_value', 1), ...]

    load (dirname: str) -> None
        Load internal states.

        Returns  None

    observe (obs: Any, reward: float, done: bool, reset: bool) -> None
        Observe consequences of the last action.

        Returns  None

    save (dirname: str) -> None
        Save internal states.

        Returns  None
```
2.2.2 Agent implementations

```python
class pfrl.agents.A2C(model, optimizer, gamma, num_processes, gpu=None, update_steps=5,
phi=<function A2C.<lambda>>, pi_loss_coef=1.0, v_loss_coef=0.5, entropy_coef=0.01, use_gae=False, tau=0.95, act_deterministically=False, max_grad_norm=None, average_actor_loss_decay=0.999, average_entropy_decay=0.999, average_value_decay=0.999, batch_states=<function batch_states>)
```

A2C: Advantage Actor-Critic.

**A2C is a synchronous, deterministic variant of Asynchronous Advantage Actor Critic** (A3C).


**Parameters**

- `model (nn.Module)` – Model to train
- `optimizer (torch.optim.Optimizer)` – optimizer used to train the model
- `gamma (float)` – Discount factor [0,1]
- `num_processes (int)` – The number of processes
- `gpu (int)` – GPU device id if not None nor negative.
- `update_steps (int)` – The number of update steps
- `phi (callable)` – Feature extractor function
- `pi_loss_coef (float)` – Weight coefficient for the loss of the policy
- `v_loss_coef (float)` – Weight coefficient for the loss of the value function
- `entropy_coef (float)` – Weight coefficient for the loss of the entropy
- `use_gae (bool)` – use generalized advantage estimation(GAE)
- `tau (float)` – gae parameter
- `act_deterministically (bool)` – If set true, choose most probable actions in act method.
- `max_grad_norm (float or None)` – Maximum L2 norm of the gradient used for gradient clipping. If set to None, the gradient is not clipped.
- `average_actor_loss_decay (float)` – Decay rate of average actor loss. Used only to record statistics.
- `average_entropy_decay (float)` – Decay rate of average entropy. Used only to record statistics.
- `average_value_decay (float)` – Decay rate of average value. Used only to record statistics.
- `batch_states (callable)` – method which makes a batch of observations. default is `pfrl.utils.batch_states.batch_states`

```python
class pfrl.agents.A3C(model, optimizer, t_max, gamma, beta=0.01, process_idx=0,
phi=<function A3C.<lambda>>, pi_loss_coef=1.0, v_loss_coef=0.5, keep_loss_scale_same=False, normalize_grad_by_t_max=False, use_average_reward=False, act_deterministically=False, max_grad_norm=None, recurrent=False, average_entropy_decay=0.999, average_value_decay=0.999, batch_states=<function batch_states>)
```

A3C: Asynchronous Advantage Actor-Critic.

Parameters

• model (A3CModel) – Model to train
• optimizer (torch.optim.Optimizer) – optimizer used to train the model
• t_max (int) – The model is updated after every t_max local steps
• gamma (float) – Discount factor [0,1]
• beta (float) – Weight coefficient for the entropy regularization term.
• process_idx (int) – Index of the process.
• phi (callable) – Feature extractor function
• pi_loss_coef (float) – Weight coefficient for the loss of the policy
• v_loss_coef (float) – Weight coefficient for the loss of the value function
• act_deterministically (bool) – If set true, choose most probable actions in act method.
• max_grad_norm (float or None) – Maximum L2 norm of the gradient used for gradient clipping. If set to None, the gradient is not clipped.
• recurrent (bool) – If set to True, model is assumed to implement pfrl.nn.StatelessRecurrent.
• batch_states (callable) – method which makes a batch of observations. default is pfrl.utils.batch_states.batch_states

class pfrl.agents.ACE

ACER (Actor-Critic with Experience Replay).

See http://arxiv.org/abs/1611.01224

Parameters

• model (ACERModel) – Model to train. It must be a callable that accepts observations as input and return three values: action distributions (Distribution), Q values (ActionValue) and state values (torch.Tensor).
• optimizer (torch.optim.Optimizer) – optimizer used to train the model
• t_max (int) – The model is updated after every t_max local steps
• gamma (float) – Discount factor [0,1]
• replay_buffer (EpisodicReplayBuffer) – Replay buffer to use. If set None, this agent won’t use experience replay.
• beta (float) – Weight coefficient for the entropy regularization term.
• phi (callable) – Feature extractor function
• pi_loss_coef (float) – Weight coefficient for the loss of the policy
**PFRL Documentation, Release 0.2.1**

- **Q_loss_coef** (*float*) – Weight coefficient for the loss of the value function
- **use_trust_region** (*bool*) – If set true, use efficient TRPO.
- **trust_region_alpha** (*float*) – Decay rate of the average model used for efficient TRPO.
- **trust_region_delta** (*float*) – Threshold used for efficient TRPO.
- **truncation_threshold** (*float or None*) – Threshold used to truncate larger importance weights. If set None, importance weights are not truncated.
- **disable_online_update** (*bool*) – If set true, disable online on-policy update and rely only on experience replay.
- **n_times_replay** (*int*) – Number of times experience replay is repeated per one time of online update.
- **replay_start_size** (*int*) – Experience replay is disabled if the number of transitions in the replay buffer is lower than this value.
- **normalize_loss_by_steps** (*bool*) – If set true, losses are normalized by the number of steps taken to accumulate the losses
- **act_deterministically** (*bool*) – If set true, choose most probable actions in act method.
- **max_grad_norm** (*float or None*) – Maximum L2 norm of the gradient used for gradient clipping. If set to None, the gradient is not clipped.
- **recurrent** (*bool*) – If set to True, model is assumed to implement `pfrl.nn.StatelessRecurrent`.
- **use_Q_opc** (*bool*) – If set true, use Q_opc, a Q-value estimate without importance sampling, is used to compute advantage values for policy gradients. The original paper recommend to use in case of continuous action.
- **average_entropy_decay** (*float*) – Decay rate of average entropy. Used only to record statistics.
- **average_value_decay** (*float*) – Decay rate of average value. Used only to record statistics.
- **average_kl_decay** (*float*) – Decay rate of kl value. Used only to record statistics.

```python
class pfrl.agents.AL(*args, **kwargs)
    Advantage Learning.


    Parameters alpha (float) – Weight of (persistent) advantages. Convergence is guaranteed only for alpha in [0, 1).
```

For other arguments, see DQN.

Categorical Double DQN.


Categorical DQN.


Arguments are the same as those of DQN except q_function must return DistributionalDiscreteActionValue and clip_delta is ignored.

class pfrl.agents-DDPG(policy, q_func, actor_optimizer, critic_optimizer, replay_buffer, gamma, explorer, gpu=None, replay_start_size=50000, minibatch_size=32, update_interval=1, target_update_interval=10000, phi=<function DDPG.<lambda>>, target_update_method='hard', soft_update_taus=0.01, n_times_updates=1, recurrent=False, episodic_update_len=None, logger=<Logger pfrl.agents.ddpg (WARNING)>, batch_states=<function batch_states>, burnin_action_func=None)

Deep Deterministic Policy Gradients.

This can be used as SVG(0) by specifying a Gaussian policy instead of a deterministic policy.

Parameters

• policy (torch.nn.Module) – Policy
• `q_func`(`torch.nn.Module`) – Q-function
• `actor_optimizer`(`Optimizer`) – Optimizer setup with the policy
• `critic_optimizer`(`Optimizer`) – Optimizer setup with the Q-function
• `replay_buffer`(`ReplayBuffer`) – Replay buffer
• `gamma`(`float`) – Discount factor
• `explorer`(`Explorer`) – Explorer that specifies an exploration strategy.
• `gpu`(`int`) – GPU device id if not None nor negative.
• `replay_start_size`(`int`) – if the replay buffer’s size is less than replay_start_size, skip update
• `minibatch_size`(`int`) – Minibatch size
• `update_interval`(`int`) – Model update interval in step
• `target_update_interval`(`int`) – Target model update interval in step
• `phi`(`callable`) – Feature extractor applied to observations
• `target_update_method`(`str`) – ‘hard’ or ‘soft’.
• `soft_update_tau`(`float`) – Tau of soft target update.
• `n_times_update`(`int`) – Number of repetition of update
• `batch_accumulator`(`str`) – ‘mean’ or ‘sum’
• `episodic_update`(`bool`) – Use full episodes for update if set True
• `episodic_update_len`(`int` or `None`) – Subsequences of this length are used for update if set int and episodic_update=True
• `logger`(`logging.Logger`) – Logger used
• `batch_states`(`callable`) – method which makes a batch of observations. default is `pfrl.utils.batch_states.batch_states`
• `burnin_action_func`(`callable` or `None`) – If not None, this callable object is used to select actions before the model is updated one or more times during training.

```python
```

Double DQN.


```python
class pfrl.agents.DoublePAL(*args, **kwargs)
```

2.2. Agents
class pfrl.agents.DPP(*args, **kwargs)
    Dynamic Policy Programming with softmax operator.

    Parameters eta (float) – Positive constant.

    For other arguments, see DQN.


Deep Q-Network algorithm.

Parameters

- **q_function (StateQFunction)** – Q-function
- **optimizer (Optimizer)** – Optimizer that is already setup
- **replay_buffer (ReplayBuffer)** – Replay buffer
- **gamma (float)** – Discount factor
- **explorer (Explorer)** – Explorer that specifies an exploration strategy.
- **gpu (int)** – GPU device id if not None nor negative.
- **replay_start_size (int)** – if the replay buffer’s size is less than replay_start_size, skip update
- **minibatch_size (int)** – Minibatch size
- **update_interval (int)** – Model update interval in step
- **target_update_interval (int)** – Target model update interval in step
- **clip_delta (bool)** – Clip delta if set True
- **phi (callable)** – Feature extractor applied to observations
- **target_update_method (str)** – ‘hard’ or ‘soft’.
- **soft_update_tau (float)** – Tau of soft target update.
- **n_times_update (int)** – Number of repetition of update
- **batch_accumulator (str)** – ‘mean’ or ‘sum’
- **episodic_update_len (int or None)** – Subsequences of this length are used for update if set int and episodic_update=True
- **logger (Logger)** – Logger used
- **batch_states (callable)** – method which makes a batch of observations. default is pfrl.utils.batch_states.batch_states
• **recurrent** *(bool)* – If set to True, *model* is assumed to implement *pfrl.nn.Recurrent* and is updated in a recurrent manner.

• **max_grad_norm** *(float or None)* – Maximum L2 norm of the gradient used for gradient clipping. If set to None, the gradient is not clipped.

```python
class pfrl.agents.IQN(*args, **kwargs)
    Implicit Quantile Networks.

    Parameters
    • **quantile_thresholds_N** *(int)* – Number of quantile thresholds used in quantile regression.
    • **quantile_thresholds_N_prime** *(int)* – Number of quantile thresholds used to sample from the return distribution at the next state.
    • **quantile_thresholds_K** *(int)* – Number of quantile thresholds used to compute greedy actions.
    • **act_deterministically** *(bool)* – IQN’s action selection is by default stochastic as it samples quantile thresholds every time it acts, even for evaluation. If this option is set to True, it uses equally spaced quantile thresholds instead of randomly sampled ones for evaluation, making its action selection deterministic.

    For other arguments, see *pfrl.agents.DQN*.
```

```python
class pfrl.agents.PAL(*args, **kwargs)
    Persistent Advantage Learning.

    Parameters **alpha** *(float)* – Weight of (persistent) advantages. Convergence is guaranteed only for alpha in [0, 1).

    For other arguments, see DQN.
```

```python
class pfrl.agents.PPO(model, optimizer, obs_normalizer=None, gpu=None, gamma=0.99, lambd=0.95, phi=<function PPO.<lambda>>, value_func_coef=1.0, entropy_coef=0.01, update_interval=2048, minibatch_size=64, epochs=10, clip_eps=0.2, clip_eps_vf=None, standardize_advantages=True, batch_states=<function batch_states>, recurrent=False, max_recurrent_sequence_len=None, act_deterministically=False, max_grad_norm=None, entropy_stats_window=1000, value_stats_window=10000, policy_loss_stats_window=100)
    Proximal Policy Optimization
    See https://arxiv.org/abs/1707.06347

    Parameters
    • **model** *(torch.nn.Module)* – Model to train (including recurrent models) state \( s \) \( \mapsto (\pi(s, _), v(s)) \)
    • **optimizer** *(torch.optim.Optimizer)* – Optimizer used to train the model
    • **gpu** *(int)* – GPU device id if not None nor negative
    • **gamma** *(float)* – Discount factor [0, 1]
    • **lambd** *(float)* – Lambda-return factor [0, 1]
```

### 2.2. Agents

13
• **phi** (*callable*) – Feature extractor function
• **value_func_coef** (*float*) – Weight coefficient for loss of value function (0, inf)
• **entropy_coef** (*float*) – Weight coefficient for entropy bonus [0, inf)
• **update_interval** (*int*) – Model update interval in step
• **minibatch_size** (*int*) – Minibatch size
• **epochs** (*int*) – Training epochs in an update
• **clip_eps** (*float*) – Epsilon for pessimistic clipping of likelihood ratio to update policy
• **clip_eps_vf** (*float*) – Epsilon for pessimistic clipping of value to update value function. If it is None, value function is not clipped on updates.
• **standardize_advantages** (*bool*) – Use standardized advantages on updates
• **recurrent** (*bool*) – If set to True, model is assumed to implement pfrl.nn.Recurrent and update in a recurrent manner.
• **max_recurrent_sequence_len** (*int*) – Maximum length of consecutive sequences of transitions in a minibatch for updating the model. This value is used only when recurrent is True. A smaller value will encourage a minibatch to contain more and shorter sequences.
• **act_deterministically** (*bool*) – If set to True, choose most probable actions in the act method instead of sampling from distributions.
• **max_grad_norm** (*float or None*) – Maximum L2 norm of the gradient used for gradient clipping. If set to None, the gradient is not clipped.
• **value_stats_window** (*int*) – Window size used to compute statistics of value predictions.
• **entropy_stats_window** (*int*) – Window size used to compute statistics of entropy of action distributions.
• **value_loss_stats_window** (*int*) – Window size used to compute statistics of loss values regarding the value function.
• **policy_loss_stats_window** (*int*) – Window size used to compute statistics of loss values regarding the policy.

**Statistics:**

- **average_value**: Average of value predictions on non-terminal states. It’s updated on (batch)act_and_train.
- **average_entropy**: Average of entropy of action distributions on non-terminal states. It’s updated on (batch)act_and_train.
- **average_value_loss**: Average of losses regarding the value function. It’s updated after the model is updated.
- **average_policy_loss**: Average of losses regarding the policy. It’s updated after the model is updated.
- **n_updates**: Number of model updates so far.
- **explained_variance**: Explained variance computed from the last batch.
class pfrl.agents.REINFORCE(model, optimizer, gpu=None, beta=0, phi=<function REINFORCE.<lambda>>, batchsize=1, act_deterministically=False, average_entropy_decay=0.999, backward_separately=False, batch_states=<function batch_states>, recurrent=False, max_grad_norm=None, logger=None)

William's episodic REINFORCE.

Parameters

- **model (Policy)** – Model to train. It must be a callable that accepts observations as input and return action distributions (Distribution).
- **optimizer (torch.optim.Optimizer)** – optimizer used to train the model
- **gpu (int)** – GPU device id if not None nor negative
- **beta (float)** – Weight coefficient for the entropy regularizaiton term.
- **phi (callable)** – Feature extractor function
- **act_deterministically (bool)** – If set true, choose most probable actions in act method.
- **batchsize (int)** – Number of episodes used for each update
- **backward_separately (bool)** – If set true, call backward separately for each episode and accumulate only gradients.
- **average_entropy_decay (float)** – Decay rate of average entropy. Used only to record statistics.
- **batch_states (callable)** – Method which makes a batch of observations. default is pfrl.utils.batch_states
- **recurrent (bool)** – If set to True, model is assumed to implement pfrl.nn.Recurrent and update in a recurrent manner.
- **max_grad_norm (float or None)** – Maximum L2 norm of the gradient used for gradient clipping. If set to None, the gradient is not clipped.
- **logger (logging.Logger)** – Logger to be used.

class pfrl.agents.SoftActorCritic(policy, q_func1, q_func2, policy_optimizer, q_func1_optimizer, q_func2_optimizer, replay_buffer, gamma, gpu=None, replay_start_size=10000, minibatch_size=100, update_interval=1, phi=<function SoftActorCritic.<lambda>>, soft_update_tau=0.005, max_grad_norm=None, logger=<Logger pfrl.agents.soft_actor_critic (WARNING)>, batch_states=<function batch_states>, burnin_action_func=None, initial_temperature=1.0, entropy_target=None, temperature_optimizer_lr=None, act_deterministically=True)

Soft Actor-Critic (SAC).

See https://arxiv.org/abs/1812.05905

Parameters

- **policy (Policy)** – Policy.
- **q_func1 (Module)** – First Q-function that takes state-action pairs as input and outputs predicted Q-values.
• q_func2 (Module) – Second Q-function that takes state-action pairs as input and outputs predicted Q-values.
• policy_optimizer (Optimizer) – Optimizer setup with the policy
• q_func1_optimizer (Optimizer) – Optimizer setup with the first Q-function.
• q_func2_optimizer (Optimizer) – Optimizer setup with the second Q-function.
• replay_buffer (ReplayBuffer) – Replay buffer
• gamma (float) – Discount factor
• gpu (int) – GPU device id if not None nor negative.
• replay_start_size (int) – if the replay buffer’s size is less than replay_start_size, skip update
• minibatch_size (int) – Minibatch size
• update_interval (int) – Model update interval in step
• phi (callable) – Feature extractor applied to observations
• soft_update_tau (float) – Tau of soft target update.
• logger (Logger) – Logger used
• batch_states (callable) – method which makes a batch of observations. default is pfrl.utils.batch_states.batch_states
• burnin_action_func (callable or None) – If not None, this callable object is used to select actions before the model is updated one or more times during training.
• initial_temperature (float) – Initial temperature value. If entropy_target is set to None, the temperature is fixed to it.
• entropy_target (float or None) – If set to a float, the temperature is adjusted during training to match the policy’s entropy to it.
• temperature_optimizer_lr (float) – Learning rate of the temperature optimizer. If set to None, Adam with default hyperparameters is used.
• act_deterministically (bool) – If set to True, choose most probable actions in the act method instead of sampling from distributions.

class pfrl.agents.TD3 (policy, q_func1, q_func2, policy_optimizer, q_func1_optimizer, q_func2_optimizer, replay_buffer, gamma, explorer, gpu=None, replay_start_size=10000, minibatch_size=100, update_interval=1, phi=<function TD3.<lambda>>, soft_update_tau=0.005, n_times_update=1, max_grad_norm=None, logger=Logger pfrl.agents.td3 (WARNING)), batch_states=<function batch_states>, burnin_action_func=None, policy_update_delay=2, target_policy_smoothing_func=<function default_target_policy_smoothing_func>)

Twin Delayed Deep Deterministic Policy Gradients (TD3).
See http://arxiv.org/abs/1802.09477

Parameters

• policy (Policy) – Policy.
• q_func1 (Module) – First Q-function that takes state-action pairs as input and outputs predicted Q-values.
• **q_func2** *(Module)* – Second Q-function that takes state-action pairs as input and outputs predicted Q-values.

• **policy_optimizer** *(Optimizer)* – Optimizer setup with the policy

• **q_func1_optimizer** *(Optimizer)* – Optimizer setup with the first Q-function.

• **q_func2_optimizer** *(Optimizer)* – Optimizer setup with the second Q-function.

• **replay_buffer** *(ReplayBuffer)* – Replay buffer

• **gamma** *(float)* – Discount factor

• **explorer** *(Explorer)* – Explorer that specifies an exploration strategy.

• **gpu** *(int)* – GPU device id if not None nor negative.

• **replay_start_size** *(int)* – if the replay buffer’s size is less than replay_start_size, skip update

• **minibatch_size** *(int)* – Minibatch size

• **update_interval** *(int)* – Model update interval in step

• **phi** *(callable)* – Feature extractor applied to observations

• **soft_update_tau** *(float)* – Tau of soft target update.

• **logger** *(Logger)* – Logger used

• **batch_states** *(callable)* – method which makes a batch of observations. default is `pfrl.utils.batch_states.batch_states`

• **burnin_action_func** *(callable or None)* – If not None, this callable object is used to select actions before the model is updated one or more times during training.

• **policy_update_delay** *(int)* – Delay of policy updates. Policy is updated once in policy_update_delay times of Q-function updates.

• **target_policy_smoothing_func** *(callable)* – Callable that takes a batch of actions as input and outputs a noisy version of it. It is used for target policy smoothing when computing target Q-values.

```python
class pfrl.agents.TRPO(policy, vf, vf_optimizer, obs_normalizer=None, gpu=None, gamma=0.99, lambd=0.95, phi=<function TRPO.<lambda>>, entropy_coef=0.01, update_interval=2048, max_kl=0.01, vf_epochs=3, vf_batch_size=64, standardize_advantages=True, batch_states=<function batch_states>, recurrent=False, max_recurrent_sequence_len=None, line_search_max_backtrack=10, conjugate_gradient_max_iter=10, conjugate_gradient_damping=0.01, act_deterministically=False, max_grad_norm=None, value_stats_window=1000, entropy_stats_window=1000, kl_stats_window=100, policy_step_size_stats_window=100, logger=<Logger pfrl.agents.trpo (WARNING)>)
```

Trust Region Policy Optimization.

A given stochastic policy is optimized by the TRPO algorithm. A given value function is also trained to predict by the TD(lambda) algorithm and used for Generalized Advantage Estimation (GAE).

Since the policy is optimized via the conjugate gradient method and line search while the value function is optimized via SGD, these two models should be separate.

Since TRPO requires second-order derivatives to compute Hessian-vector products, your policy must contain only functions that support second-order derivatives.

Parameters

- **policy** *(Policy)* – Stochastic policy. Its forward computation must contain only functions that support second-order derivatives. Recurrent models are not supported.
- **vf** *(ValueFunction)* – Value function. Recurrent models are not supported.
- **vf_optimizer** *(torch.optim.Optimizer)* – Optimizer for the value function.
- **obs_normalizer** *(pfrl.nn.EmpiricalNormalization or None)* – If set to pfrl.nn.EmpiricalNormalization, it is used to normalize observations based on the empirical mean and standard deviation of observations. These statistics are updated after computing advantages and target values and before updating the policy and the value function.
- **gpu** *(int)* – GPU device id if not None nor negative
- **gamma** *(float)* – Discount factor [0, 1]
- **lambd** *(float)* – Lambda-return factor [0, 1]
- **phi** *(callable)* – Feature extractor function
- **entropy_coef** *(float)* – Weight coefficient for entropy bonus [0, inf)
- **update_interval** *(int)* – Interval steps of TRPO iterations. Every time after this amount of steps, this agent updates the policy and the value function using data from these steps.
- **vf_epochs** *(int)* – Number of epochs for which the value function is trained on each TRPO iteration.
- **vf_batch_size** *(int)* – Batch size of SGD for the value function.
- **standardize_advantages** *(bool)* – Use standardized advantages on updates
- **line_search_max_backtrack** *(int)* – Maximum number of backtracking in line search to tune step sizes of policy updates.
- **conjugate_gradient_max_iter** *(int)* – Maximum number of iterations in the conjugate gradient method.
- **conjugate_gradient_damping** *(float)* – Damping factor used in the conjugate gradient method.
- **act_deterministically** *(bool)* – If set to True, choose most probable actions in the act method instead of sampling from distributions.
- **max_grad_norm** *(float or None)* – Maximum L2 norm of the gradient used for gradient clipping. If set to None, the gradient is not clipped.
- **value_stats_window** *(int)* – Window size used to compute statistics of value predictions.
- **entropy_stats_window** *(int)* – Window size used to compute statistics of entropy of action distributions.
- **kl_stats_window** *(int)* – Window size used to compute statistics of KL divergence between old and new policies.
- **policy_step_size_stats_window** *(int)* – Window size used to compute statistics of step sizes of policy updates.

Statistics:
average_value: Average of value predictions on non-terminal states. It’s updated after act or batch_act methods are called in the training mode.

average_entropy: Average of entropy of action distributions on non-terminal states. It’s updated after act or batch_act methods are called in the training mode.

average_kl: Average of KL divergence between old and new policies. It’s updated after the policy is updated.

average_policy_step_size: Average of step sizes of policy updates. It’s updated after the policy is updated.

2.3 Experiments

2.3.1 Training and evaluation

```python
def train_agent_async(outdir, processes, make_env, profile=False, steps=80000000, eval_interval=1000000, eval_n_steps=None, eval_n_episodes=10, eval_success_threshold=0.0, max_episode_len=None, step_offset=0, successful_score=None, agent=None, make_agent=None, global_step_hooks=[], evaluation_hooks=(), save_best_so_far_agent=True, use_tensorboard=False, logger=None, random_seeds=None, stop_event=None, exception_event=None)
```

Train agent asynchronously using multiprocessing.

Either `agent` or `make_agent` must be specified.

**Parameters**

- `outdir (str)` – Path to the directory to output things.
- `processes (int)` – Number of processes.
- `make_env (callable)` – (process_idx, test) -> Environment.
- `profile (bool)` – Profile if set True.
- `steps (int)` – Number of global time steps for training.
- `eval_interval (int)` – Interval of evaluation. If set to None, the agent will not be evaluated at all.
- `eval_n_steps (int)` – Number of eval timesteps at each eval phase
- `eval_n_episodes (int)` – Number of eval episodes at each eval phase
- `eval_success_threshold (float)` – r-threshold above which grasp succeeds
- `max_episode_len (int)` – Maximum episode length.
- `step_offset (int)` – Time step from which training starts.
- `successful_score (float)` – Finish training if the mean score is greater or equal to this value if not None
- `agent (Agent)` – Agent to train.
- `make_agent (callable)` – (process_idx) -> Agent
• **global_step_hooks** (list) – List of callable objects that accepts (env, agent, step) as arguments. They are called every global step. See pfrl.experiments.hooks.

• **evaluation_hooks** (Sequence) – Sequence of pfrl.experiments.evaluation_hooks.EvaluationHook objects. They are called after each evaluation.

• **save_best_so_far_agent** (bool) – If set to True, after each evaluation, if the score (= mean return of evaluation episodes) exceeds the best-so-far score, the current agent is saved.

• **use_tensorboard** (bool) – Additionally log eval stats to tensorboard

• **logger** (logging.Logger) – Logger used in this function.

• **random_seeds** (array-like of ints or None) – Random seeds for processes. If set to None, [0, 1, ..., processes-1] are used.

• **stop_event** (multiprocessing.Event or None) – Event to stop training. If set to None, a new Event object is created and used internally.

• **exception_event** (multiprocessing.Event or None) – Event that indicates other thread raised an exception. The train will be terminated and the current agent will be saved. If set to None, a new Event object is created and used internally.

**Returns** Trained agent.

```python
pfrl.experiments.train_agent_batch(agent, env, steps, outdir, checkpoint_freq=None, log_interval=None, max_episode_len=None, step_offset=0, evaluator=None, successful_score=None, step_hooks=(), return_window_size=100, logger=None)
```

Train an agent in a batch environment.

**Parameters**

• **agent** – Agent to train.

• **env** – Environment to train the agent against.

• **steps** (int) – Number of total time steps for training.

• **outdir** (str) – Path to the directory to output things.

• **checkpoint_freq** (int) – Frequency at which agents are stored.

• **log_interval** (int) – Interval of logging.

• **max_episode_len** (int) – Maximum episode length.

• **step_offset** (int) – Time step from which training starts.

• **return_window_size** (int) – Number of training episodes used to estimate the average returns of the current agent.

• **successful_score** (float) – Finish training if the mean score is greater or equal to this value if not None

• **step_hooks** (Sequence) – Sequence of callable objects that accepts (env, agent, step) as arguments. They are called every step. See pfrl.experiments.hooks.

• **logger** (logging.Logger) – Logger used in this function.

**Returns** List of evaluation episode stats dict.
Train an agent while regularly evaluating it.

**Parameters**

- `agent` – Agent to train.
- `env` – Environment train the agent against.
- `steps` *(int)* – Number of total time steps for training.
- `eval_n_steps` *(int)* – Number of timesteps at each evaluation phase.
- `eval_n_episodes` *(int)* – Number of runs for each time of evaluation.
- `eval_interval` *(int)* – Interval of evaluation.
- `outdir` *(str)* – Path to the directory to output things.
- `log_interval` *(int)* – Interval of logging.
- `checkpoint_freq` *(int)* – Frequency with which to store networks.
- `max_episode_len` *(int)* – Maximum episode length.
- `step_offset` *(int)* – Time step from which training starts.
- `return_window_size` *(int)* – Number of training episodes used to estimate the average returns of the current agent.
- `eval_max_episode_len` *(int or None)* – Maximum episode length of evaluation runs. If set to None, max_episode_len is used instead.
- `eval_env` – Environment used for evaluation.
- `successful_score` *(float)* – Finish training if the mean score is greater or equal to this value if not None.
- `step_hooks` *(Sequence)* – Sequence of callable objects that accepts (env, agent, step) as arguments. They are called every step. See pfrl.experiments.hooks.
- `evaluation_hooks` *(Sequence)* – Sequence of pfrl.experiments.evaluation_hooks.EvaluationHook objects. They are called after each evaluation.
- `save_best_so_far_agent` *(bool)* – If set to True, after each evaluation, if the score (= mean return of evaluation episodes) exceeds the best-so-far score, the current agent is saved.
- `use_tensorboard` *(bool)* – Additionally log eval stats to tensorboard.
- `logger` *(logging.Logger)* – Logger used in this function.
Returns
Trained agent. eval_stats_history: List of evaluation episode stats dict.

Return type
agent

`pfrl.experiments.train_agent_with_evaluation(agent, env, steps, eval_n_steps, eval_n_episodes, eval_interval, outdir, checkpoint_freq=None, train_max_episode_len=None, step_offset=0, eval_max_episode_len=None, eval_env=None, successful_score=None, step_hooks=(), evaluation_hooks=(), save_best_so_far_agent=True, use_tensorboard=False, eval_during_episode=False, logger=None)`

Train an agent while periodically evaluating it.

Parameters
- **agent** – A pfrl.agent.Agent
- **env** – Environment train the agent against.
- **steps (int)** – Total number of timesteps for training.
- **eval_n_steps (int)** – Number of timesteps at each evaluation phase.
- **eval_n_episodes (int)** – Number of episodes at each evaluation phase.
- **eval_interval (int)** – Interval of evaluation.
- **outdir (str)** – Path to the directory to output data.
- **checkpoint_freq (int)** – Frequency at which agents are stored.
- **train_max_episode_len (int)** – Maximum episode length during training.
- **step_offset (int)** – Time step from which training starts.
- **eval_max_episode_len (int or None)** – Maximum episode length of evaluation runs. If None, train_max_episode_len is used instead.
- **eval_env** – Environment used for evaluation.
- **successful_score (float)** – Finish training if the mean score is greater than or equal to this value if not None
- **step_hooks (Sequence)** – Sequence of callable objects that accepts (env, agent, step) as arguments. They are called every step. See pfrl.experiments.hooks.
- **evaluation_hooks (Sequence)** – Sequence of pfrl.experiments.evaluation_hooks.EvaluationHook objects. They are called after each evaluation.
- **save_best_so_far_agent (bool)** – If set to True, after each evaluation phase, if the score (= mean return of evaluation episodes) exceeds the best-so-far score, the current agent is saved.
- **use_tensorboard (bool)** – Additionally log eval stats to tensorboard
- **eval_during_episode (bool)** – Allow running evaluation during training episodes. This should be enabled only when env and eval_env are independent.
- **logger (logging.Logger)** – Logger used in this function.
Returns  Trained agent. eval_stats_history: List of evaluation episode stats dict.

Return type  agent

### 2.3.2 Training hooks

class pfrl.experiments.StepHook

Hook function that will be called in training.

This class is for clarifying the interface required for Hook functions. You don’t need to inherit this class to define your own hooks. Any callable that accepts (env, agent, step) as arguments can be used as a hook.

class pfrl.experiments.LinearInterpolationHook(total_steps, start_value, stop_value, setter)

Hook that will set a linearly interpolated value.

You can use this hook to decay the learning rate by using a setter function as follows:

```python
def lr_setter(env, agent, value):
    agent.optimizer.lr = value

hook = LinearInterpolationHook(10 ** 6, 1e-3, 0, lr_setter)
```

Parameters

- **total_steps**(int) – Number of total steps.
- **start_value**(float) – Start value.
- **stop_value**(float) – Stop value.
- **setter**(callable) – (env, agent, value) -> None

### 2.3.3 Experiment Management

pfrl.experiments.generate_exp_id(prefix=None, argv=['/home/docs/checkouts/readthedocs.org/user_builds/pfrl/envs/latest/lib/python3.7/site-packages/sphinx/__main__.py', '-T', '-E', '-b', 'html', '-d', '_build/doctrees', '-D', 'language=en', ';;', '_build/html'])

→ str

Generate reproducible, unique and deterministic experiment id

The generated id will be string generated from prefix, Git checksum, git diff from HEAD and command line arguments.

Returns A generated experiment id in string (str) which if avialable for directory name

pfrl.experiments.prepare_output_dir(args, basedir=None, exp_id=None, argv=None, time_format='%Y%m%dT%H%M%S.%f', make_backup=True) → str

Prepare a directory for outputting training results.

An output directory, which ends with the current datetime string, is created. Then the following infromation is saved into the directory:

- args.txt: argument values and arbitrary parameters
- command.txt: command itself
- environ.txt: environmental variables
- start.txt: timestamp when the experiment executed

Additionally, if the current directory is under git control, the following information is saved:

- git-head.txt: result of `git rev-parse HEAD`
- git-status.txt: result of `git status`
- git-log.txt: result of `git log`
- git-diff.txt: result of `git diff HEAD`

---

**PFRL Documentation, Release 0.2.1**

**2.3. Experiments** 23
Parameters

- **exp_id** (*str or None*) – Experiment identifier. If None is given, reproducible ID will be automatically generated from Git version hash and command arguments. If the code is not under Git control, it is generated from current timestamp under the format of `time_format`.

- **args** (*dict or argparse.Namespace*) – Arguments to save to see parameters

- **basedir** (*str or None*) – If a string is specified, the output directory is created under that path. If not specified, it is created in current directory.

- **argv** (*list or None*) – The list of command line arguments passed to a script. If not specified, sys.argv is used instead.

- **time_format** (*str*) – Format used to represent the current datetime. The default format is the basic format of ISO 8601.

- **make_backup** (*bool*) – If there exists old experiment with same name, copy a backup with additional suffix with `time_format`.

Returns Path of the output directory created by this function (*str*).

### 2.4 Explorers

#### 2.4.1 Explorer interfaces

**class pfrl.explorer.Explorer**

Abstract explorer.

**select_action** (*t, greedy_action_func, action_value=None*)

Select an action.

Parameters

- **t** – current time step

- **greedy_action_func** – function with no argument that returns an action

- **action_value** (*ActionValue*) – ActionValue object

#### 2.4.2 Explorer implementations

**class pfrl.explorers.AdditiveGaussian** (*scale, low=None, high=None*)

Additive Gaussian noise to actions.

Each action must be numpy.ndarray.

Parameters

- **scale** (*float or array_like of floats*) – Scale parameter.

- **low** (*float, array_like of floats, or None*) – Lower bound of action space used to clip an action after adding a noise. If set to None, clipping is not performed on lower edge.

- **high** (*float, array_like of floats, or None*) – Higher bound of action space used to clip an action after adding a noise. If set to None, clipping is not performed on upper edge.
class pfrl.explorers.AdditiveOU(mu=0.0, theta=0.15, sigma=0.3, start_with_mu=False, logger=<Logger pfrl.explorers.additive_ou (WARNING)>)
Additive Ornstein-Uhlenbeck process.

Parameters
- mu (float) – Mean of the OU process
- theta (float) – Friction to pull towards the mean
- sigma (float or ndarray) – Scale of noise
- start_with_mu (bool) – Start the process without noise

class pfrl.explorers.Boltzmann(T=1.0)
Boltzmann exploration.

Parameters
- T (float) – Temperature of Boltzmann distribution.

class pfrl.explorers.ConstantEpsilonGreedy(epsilon, random_action_func, logger=<Logger pfrl.explorers.epsilon_greedy (WARNING)>)
Epsilon-greedy with constant epsilon.

Parameters
- epsilon – epsilon used
- random_action_func – function with no argument that returns action
- logger – logger used

class pfrl.explorers.LinearDecayEpsilonGreedy(start_epsilon, end_epsilon, decay_steps, random_action_func, logger=<Logger pfrl.explorers.epsilon_greedy (WARNING)>)
Epsilon-greedy with linearly decayed epsilon

Parameters
- start_epsilon – max value of epsilon
- end_epsilon – min value of epsilon
- decay_steps – how many steps it takes for epsilon to decay
- random_action_func – function with no argument that returns action
- logger – logger used

class pfrl.explorers.Greedy
No exploration

2.5 Modules

2.5.1 Module interfaces

class pfrl.nn.Recurrent
Recurrent module interface.

This class defines the interface of a recurrent module PFRL support.
The interface is similar to that of `torch.nn.LSTM` except that sequential data are expected to be packed in `torch.nn.utils.rnn.PackedSequence`.

To implement a model with recurrent layers, you can either use default container classes such as `pfrl.nn.RecurrentSequential` and `pfrl.nn.RecurrentBranched` or write your module extending this class and `torch.nn.Module`.

**forward**(*packed_input, recurrent_state*)

Multi-step batch forward computation.

**Parameters**
- `recurrent_state` (**object or None**) – Batched recurrent state. If set to None, it is initialized.

**Returns**
- **Output sequences. Tensors will be packed in** `torch.nn.utils.rnn.PackedSequence`.
- object or None: New batched recurrent state.

**Return type** **object**

### 2.5.2 Module implementations

**class** `pfrl.nn.Branched(*modules)`

Module that calls forward functions of child modules in parallel.

When the `forward` method of this module is called, all the arguments are forwarded to each child module’s `forward` method.

The returned values from the child modules are returned as a tuple.

**Parameters**
- `*modules` – Child modules. Each module should be callable.

**class** `pfrl.nn.EmpiricalNormalization(shape, batch_axis=0, eps=0.01, dtype=<class 'numpy.float32'>, until=None, clip_threshold=None)`

Normalize mean and variance of values based on empirical values.

**Parameters**
- `shape` (**int or tuple of int**) – Shape of input values except batch axis.
- `batch_axis` (**int**) – Batch axis.
- `eps` (**float**) – Small value for stability.
- `dtype` (**dtype**) – Dtype of input values.
- `until` (**int or None**) – If this arg is specified, the link learns input values until the sum of batch sizes exceeds it.

**class** `pfrl.nn.FactorizedNoisyLinear(mu_link, sigma_scale=0.4)`

Linear layer in Factorized Noisy Network

**Parameters**
- `mu_link` (**nn.Linear**) – Linear link that computes mean of output.
- `sigma_scale` (**float**) – The hyperparameter sigma_0 in the original paper. Scaling factor of the initial weights of noise-scaling parameters.
**class** pfrl.nn.MLP *(in_size, out_size, hidden_sizes, nonlinearity=<function relu>, last_wscale=1)*

Multi-Layer Perceptron

**class** pfrl.nn.MLPBN *(in_size, out_size, hidden_sizes, normalize_input=True, normalize_output=False, nonlinearity=<function relu>, last_wscale=1)*

Multi-Layer Perceptron with Batch Normalization.

**Parameters**

- **in_size (int)** – Input size.
- **out_size (int)** – Output size.
- **hidden_sizes (list of ints)** – Sizes of hidden channels.
- **normalize_input (bool)** – If set to True, Batch Normalization is applied to inputs.
- **normalize_output (bool)** – If set to True, Batch Normalization is applied to outputs.
- **nonlinearity (callable)** – Nonlinearity between layers. It must accept a Variable as an argument and return a Variable with the same shape. Nonlinearities with learnable parameters such as PReLU are not supported.
- **last_wscale (float)** – Scale of weight initialization of the last layer.

**class** pfrl.nn.SmallAtariCNN *(n_input_channels=4, n_output_channels=256, activation=<function relu>, bias=0.1)*


**class** pfrl.nn.LargeAtariCNN *(n_input_channels=4, n_output_channels=512, activation=<function relu>, bias=0.1)*


See: [https://www.nature.com/articles/nature14236](https://www.nature.com/articles/nature14236)

**class** pfrl.nn.RecurrentBranched *(*modules)*

Recurrent module that bundles parallel branches.

This is a recurrent analog to pfrl.nn.Branched. It bundles multiple recurrent modules.

**Parameters** *modules – Child modules. Each module should be recurrent and callable.

**class** pfrl.nn.RecurrentSequential *(*args)*

Sequential model that can contain stateless recurrent modules.

This is a recurrent analog to torch.nn.Sequential. It supports the recurrent interface by automatically detecting recurrent modules and handles recurrent states properly.

For non-recurrent layers, this module automatically concatenates the input to the layers for efficient computation.

**Parameters** *layers – Callable objects.

### 2.5.3 Module utility functions

**pfrl.nn.to_factorized_noisy** *(module, *args, **kwargs)*

Add noisiness to components of given module.

Currently this fn. only supports torch.nn.Linear (with and without bias)
2.6 Policies

2.6.1 Head modules for Gaussian policies

class pfrl.policies.GaussianHeadWithFixedCovariance(scale=1)
Gaussian head with fixed covariance.

This module is intended to be attached to a neural network that outputs the mean of a Gaussian policy. Its covariance is fixed to a diagonal matrix with a given scale.

Parameters

- scale (float) – Scale parameter.

class pfrl.policies.GaussianHeadWithDiagonalCovariance(var_func=<built-in function softplus>)
Gaussian head with diagonal covariance.

This module is intended to be attached to a neural network that outputs a vector that is twice the size of an action vector. The vector is split and interpreted as the mean and diagonal covariance of a Gaussian policy.

Parameters

- var_func (callable) – Callable that computes the variance from the second input.
  It should always return positive values.

class pfrl.policies.GaussianHeadWithStateIndependentCovariance(action_size, var_type='spherical', var_func=<built-in function softplus>, var_param_init=0)
Gaussian head with state-independent learned covariance.

This link is intended to be attached to a neural network that outputs the mean of a Gaussian policy. The only learnable parameter this link has determines the variance in a state-independent way.

State-independent parameterization of the variance of a Gaussian policy is often used with PPO and TRPO, e.g., in https://arxiv.org/abs/1709.06560.

Parameters

- action_size (int) – Number of dimensions of the action space.
- var_type (str) – Type of parameterization of variance. It must be ‘spherical’ or ‘diagonal’.
- var_func (callable) – Callable that computes the variance from the var parameter. It should always return positive values.
- var_param_init (float) – Initial value the var parameter.

2.6.2 Head modules for deterministic policies

class pfrl.policies.DeterministicHead
Head module for a deterministic policy.

2.6.3 Head modules for categorical policies

class pfrl.policies.SoftmaxCategoricalHead
2.7 Q-functions

2.7.1 Q-function interfaces

class pfrl.q_function.StateQFunction
    Abstract Q-function with state input.

    __call__(x)
    Evaluates Q-function

    Parameters
    • x (ndarray) – state input

    Returns
    An instance of ActionValue that allows to calculate the Q-values for state x and every possible action

class pfrl.q_function.StateActionQFunction
    Abstract Q-function with state and action input.

    __call__(x, a)
    Evaluates Q-function

    Parameters
    • x (ndarray) – state input
    • a (ndarray) – action input

    Returns
    Q-value for state x and action a

2.7.2 Q-function implementations

class pfrl.q_functions.DuelingDQN(n_actions, n_input_channels=4, activation=<function relu>, bias=0.1)
    Dueling Q-Network

    See: http://arxiv.org/abs/1511.06581

class pfrl.q_functions.DistributionalDuelingDQN(n_actions, n_atoms, v_min, v_max, n_input_channels=4, activation=<built-in method relu of type object>, bias=0.1)
    Distributional dueling fully-connected Q-function with discrete actions.

class pfrl.q_functions.SingleModelStateQFunctionWithDiscreteAction(model)
    Q-function with discrete actions.

    Parameters
    • model (nn.Module) – Model that is callable and outputs action values.

class pfrl.q_functions.FCStateQFunctionWithDiscreteAction(n_dim_obs, n_actions, n_hidden_channels, n_hidden_layers, nonlinearity=<function relu>, last_wscale=1.0)
    Fully-connected state-input Q-function with discrete actions.

    Parameters
    • n_dim_obs – number of dimensions of observation space
    • n_actions (int) – Number of actions in action space.
• **n_hidden_channels** – number of hidden channels
• **n_hidden_layers** – number of hidden layers
• **nonlinearity** (*callable*) – Nonlinearity applied after each hidden layer.
• **last_wscale** (*float*) – Weight scale of the last layer.

```python
class pfrl.q_functions.DistributionalSingleModelStateQFunctionWithDiscreteAction(model, z_values)
```

Distributional Q-function with discrete actions.

**Parameters**

• **model** (*nn.Module*) – model that is callable and outputs atoms for each action.
• **z_values** (*ndarray*) – Returns represented by atoms. Its shape must be (n_atoms,).

```python
class pfrl.q_functions.DistributionalFCStateQFunctionWithDiscreteAction(ndim_obs, n_actions, n_atoms, v_min, v_max, n_hidden_channels, n_hidden_layers, nonlinearity=<function relu>, last_wscale=1.0)
```

Distributional fully-connected Q-function with discrete actions.

**Parameters**

• **n_dim_obs** (*int*) – Number of dimensions of observation space.
• **n_actions** (*int*) – Number of actions in action space.
• **n_atoms** (*int*) – Number of atoms of return distribution.
• **v_min** (*float*) – Minimum value this model can approximate.
• **v_max** (*float*) – Maximum value this model can approximate.
• **n_hidden_channels** (*int*) – Number of hidden channels.
• **n_hidden_layers** (*int*) – Number of hidden layers.
• **nonlinearity** (*callable*) – Nonlinearity applied after each hidden layer.
• **last_wscale** (*float*) – Weight scale of the last layer.

```python
class pfrl.q_functions.FCQuadraticStateQFunction(n_input_channels, n_dim_action, n_hidden_channels, n_hidden_layers, action_space, scale_mu=True)
```

Fully-connected state-input continuous Q-function.


**Parameters**

• **n_input_channels** – number of input channels
• **n_dim_action** – number of dimensions of action space
• \texttt{n\_hidden\_channels} – number of hidden channels
• \texttt{n\_hidden\_layers} – number of hidden layers
• \texttt{action\_space} – action\_space
• \texttt{scale\_mu} (\texttt{bool}) – scale mu by applying tanh if True

\texttt{class pfrl.q\_functions.SingleModelStateActionQFunction(model)}

Q-function with discrete actions.

Parameters \texttt{model (nn.Module)} – Module that is callable and outputs action values.

\texttt{class pfrl.q\_functions.FCSAQFunction(n\_dim\_obs, n\_dim\_action, n\_hidden\_channels, n\_hidden\_layers, nonlinearity=<function relu>, last\_wscale=1.0)}

Fully-connected (s,a)-input Q-function.

Parameters

• \texttt{n\_dim\_obs (int)} – Number of dimensions of observation space.
• \texttt{n\_dim\_action (int)} – Number of dimensions of action space.
• \texttt{n\_hidden\_channels (int)} – Number of hidden channels.
• \texttt{n\_hidden\_layers (int)} – Number of hidden layers.
• \texttt{nonlinearity (callable)} – Nonlinearity between layers. It must accept a Variable as an argument and return a Variable with the same shape. Nonlinearities with learnable parameters such as PReLU are not supported. It is not used if \texttt{n\_hidden\_layers} is zero.
• \texttt{last\_wscale (float)} – Scale of weight initialization of the last layer.

\texttt{class pfrl.q\_functions.FCLSTMSAQFunction(n\_dim\_obs, n\_dim\_action, n\_hidden\_channels, n\_hidden\_layers, nonlinearity=<function relu>, last\_wscale=1.0)}

Fully-connected + LSTM (s,a)-input Q-function.

Parameters

• \texttt{n\_dim\_obs (int)} – Number of dimensions of observation space.
• \texttt{n\_dim\_action (int)} – Number of dimensions of action space.
• \texttt{n\_hidden\_channels (int)} – Number of hidden channels.
• \texttt{n\_hidden\_layers (int)} – Number of hidden layers.
• \texttt{nonlinearity (callable)} – Nonlinearity between layers. It must accept a Variable as an argument and return a Variable with the same shape. Nonlinearities with learnable parameters such as PReLU are not supported.
• \texttt{last\_wscale (float)} – Scale of weight initialization of the last layer.

\texttt{class pfrl.q\_functions.FCBNSAQFunction(n\_dim\_obs, n\_dim\_action, n\_hidden\_channels, n\_hidden\_layers, normalize\_input=True, nonlinearity=<function relu>, last\_wscale=1.0)}

Fully-connected + BN (s,a)-input Q-function.

Parameters

• \texttt{n\_dim\_obs (int)} – Number of dimensions of observation space.
• \texttt{n\_dim\_action (int)} – Number of dimensions of action space.
• \texttt{n\_hidden\_channels (int)} – Number of hidden channels.
• **n_hidden_layers** *(int)* – Number of hidden layers.

• **normalize_input** *(bool)* – If set to True, Batch Normalization is applied to both observations and actions.

• **nonlinearity** *(callable)* – Nonlinearity between layers. It must accept a Variable as an argument and return a Variable with the same shape. Nonlinearities with learnable parameters such as PReLU are not supported. It is not used if n_hidden_layers is zero.

• **last_wscale** *(float)* – Scale of weight initialization of the last layer.

```python
class pfrl.q_functions.FCBNLateActionSAQFunction(n_dim_obs, n_dim_action, n_hidden_channels, n_hidden_layers, normalize_input=True, nonlinearity=<function relu>, last_wscale=1.0)
```

Fully-connected + BN (s,a)-input Q-function with late action input.

Actions are not included until the second hidden layer and not normalized. This architecture is used in the DDPG paper: http://arxiv.org/abs/1509.02971

**Parameters**

• **n_dim_obs** *(int)* – Number of dimensions of observation space.

• **n_dim_action** *(int)* – Number of dimensions of action space.

• **n_hidden_channels** *(int)* – Number of hidden channels.

• **n_hidden_layers** *(int)* – Number of hidden layers. It must be greater than or equal to 1.

• **normalize_input** *(bool)* – If set to True, Batch Normalization is applied

• **nonlinearity** *(callable)* – Nonlinearity between layers. It must accept a Variable as an argument and return a Variable with the same shape. Nonlinearities with learnable parameters such as PReLU are not supported.

• **last_wscale** *(float)* – Scale of weight initialization of the last layer.

```python
class pfrl.q_functions.FCLateActionSAQFunction(n_dim_obs, n_dim_action, n_hidden_channels, n_hidden_layers, nonlinearity=<function relu>, last_wscale=1.0)
```

Fully-connected (s,a)-input Q-function with late action input.

Actions are not included until the second hidden layer and not normalized. This architecture is used in the DDPG paper: http://arxiv.org/abs/1509.02971

**Parameters**

• **n_dim_obs** *(int)* – Number of dimensions of observation space.

• **n_dim_action** *(int)* – Number of dimensions of action space.

• **n_hidden_channels** *(int)* – Number of hidden channels.

• **n_hidden_layers** *(int)* – Number of hidden layers. It must be greater than or equal to 1.

• **nonlinearity** *(callable)* – Nonlinearity between layers. It must accept a Variable as an argument and return a Variable with the same shape. Nonlinearities with learnable parameters such as PReLU are not supported.
• **last_wscale** *(float)* – Scale of weight initialization of the last layer.

## 2.8 Replay Buffers

### 2.8.1 ReplayBuffer interfaces

```python
class pfrl.replay_buffers.ReplayBuffer(capacity: Optional[int] = None, num_steps: int = 1)
```

Experience Replay Buffer


**Parameters**

- **capacity** *(int)* – capacity in terms of number of transitions
- **num_steps** *(int)* – Number of timesteps per stored transition (for N-step updates)

```python
append(state, action, reward, next_state=None, next_action=None, is_state_terminal=False, env_id=0, **kwargs)
```

Append a transition to this replay buffer.

**Parameters**

- **state** – $s_t$
- **action** – $a_t$
- **reward** – $r_t$
- **next_state** – $s_{t+1}$ (can be None if terminal)
- **next_action** – $a_{t+1}$ (can be None for off-policy algorithms)
- **is_state_terminal** *(bool)* –
- **env_id** *(object)* – Object that is unique to each env. It indicates which env a given transition came from in multi-env training.
- ****kwargs – Any other information to store.

```python
load(filename)
```

Load the content of the buffer from a file.

**Parameters**

- **filename** *(str)* – Path to a file.

```python
sample(num_experiences)
```

Sample $n$ unique transitions from this replay buffer.

**Parameters**

- **n** *(int)* – Number of transitions to sample.

**Returns**

Sequence of $n$ sampled transitions.

```python
save(filename)
```

Save the content of the buffer to a file.

**Parameters**

- **filename** *(str)* – Path to a file.

```python
stop_current_episode(env_id=0)
```

Notify the buffer that the current episode is interrupted.

You may want to interrupt the current episode and start a new one before observing a terminal state. This is typical in continuing envs. In such cases, you need to call this method before appending a new transition so that the buffer will treat it as an initial transition of a new episode.
This method should not be called after an episode whose termination is already notified by appending a transition with `is_state_terminal=True`.

**Parameters**

- `env_id (object)` – Object that is unique to each env. It indicates which env’s current episode is interrupted in multi-env training.

### 2.8.2 ReplayBuffer implementations

**class pfrl.replay_buffers.EpisodicReplayBuffer (capacity=None)**

**class pfrl.replay_buffers.ReplayBuffer (capacity: Optional[int] = None, num_steps: int = 1)**

Experience Replay Buffer


**Parameters**

- `capacity (int)` – capacity in terms of number of transitions
- `num_steps (int)` – Number of timesteps per stored transition (for N-step updates)

**class pfrl.replay_buffers.PrioritizedReplayBuffer (capacity=None, alpha=0.6, beta0=0.4, betasteps=20000.0, eps=0.01, normalize_by_max=True, error_min=0, error_max=1, num_steps=1)**

Stochastic Prioritization

https://arxiv.org/pdf/1511.05952.pdf Section 3.3 proportional prioritization

**Parameters**

- `capacity (int)` – capacity in terms of number of transitions
- `alpha (float)` – Exponent of errors to compute probabilities to sample
- `beta0 (float)` – Initial value of beta
- `betasteps (int)` – Steps to anneal beta to 1
- `eps (float)` – To revisit a step after its error becomes near zero
- `normalize_by_max (bool)` – Method to normalize weights. 'batch' or True (default): divide by the maximum weight in the sampled batch. 'memory': divide by the maximum weight in the memory. False: do not normalize

**class pfrl.replay_buffers.PrioritizedEpisodicReplayBuffer (capacity=None, alpha=0.6, beta0=0.4, betasteps=20000.0, eps=1e-08, normalize_by_max=True, default_priority_func=None, uniform_ratio=0, wait_priority_after_sampling=True, return_sample_weights=True, error_min=None, error_max=None)**
class pfrl.replayBuffers.PersistentReplayBuffer (dirname, capacity, *, ancestor=None, logger=None, distributed=False, group=None)

Experience replay buffer that are saved to disk storage

ReplayBuffer is used to store sampled experience data, but the data is stored in DRAM memory and removed after program termination. This class adds persistence to ReplayBuffer, so that the learning process can be restarted from a previously saved replay data.

Parameters

- **dirname** (str) – Directory name where the buffer data is saved. Please note that it tries to load data from it as well. Also, it would be important to note that it can’t be used with ancestor.
- **capacity** (int) – Capacity in terms of number of transitions
- **ancestor** (str) – Path to pre-generated replay buffer. The ancestor directory is used to load/save, instead of dirname.
- **logger** – logger object
- **distributed** (bool) – Use a distributed version for the underlying persistent queue class. You need the private package pfrlmm to use this option.
- **group** – torch.distributed group object. Only used when distributed=True and pfrlmm package is available

**Note:** Contrary to the original ReplayBuffer implementation, state and next_state, action and next_action are pickled and stored as different objects even they point to the same object. This may lead to inefficient usage of storage space, but it is recommended to buy more storage - hardware is sometimes cheaper than software.

class pfrl.replayBuffers.PersistentEpisodicReplayBuffer (dirname, capacity, *, ancestor=None, logger=None, distributed=False, group=None)

Episodic version of PersistentReplayBuffer

Parameters

- **dirname** (str) – Directory name where the buffer data is saved. This cannot be used with ancestor
- **capacity** (int) – Capacity in terms of number of transitions
- **ancestor** (str) – Path to pre-generated replay buffer. The ancestor directory is used to load/save, instead of dirname.
- **logger** – logger object
- **distributed** (bool) – Use a distributed version for the underlying persistent queue class. You need the private package pfrlmm to use this option.
- **group** – torch.distributed group object. Only used when distributed=True and pfrlmm package is available

**Note:** Current implementation is inefficient, as episodic memory and memory data shares the almost same data in EpisodicReplayBuffer by reference but shows different data structure. Otherwise, persistent version
of them does not share the data between them but backing file structure is separated.
CHAPTER 3

Indices and tables

• genindex
• modindex
• search
Symbols

___call__ () (pfrl.q_function.StateActionQFunction method), 29
___call__ () (pfrl.q_function.StateQFunction method), 29

A
A2C (class in pfrl.agents), 7
A3C (class in pfrl.agents), 7
ACER (class in pfrl.agents), 8
act() (pfrl.agent.Agent method), 6
ActionValue (class in pfrl.action_value), 5
AdditiveGaussian (class in pfrl.explorers), 24
AdditiveOU (class in pfrl.explorers), 24
Agent (class in pfrl.agent), 6
AL (class in pfrl.agents), 9
append() (pfrl.replayBuffers.ReplayBuffer method), 33

B
Boltzmann (class in pfrl.explorers), 25
Branched (class in pfrl.nn), 26

C
CategoricalDoubleDQN (class in pfrl.agents), 9
CategoricalDQN (class in pfrl.agents), 10
ConstantEpsilonGreedy (class in pfrl.explorers), 25

D
DDPG (class in pfrl.agents), 10
DeterministicHead (class in pfrl.policies), 28
DiscreteActionValue (class in pfrl.action_value), 5
DistributionalDuelingDQN (class in pfrl.q_functions), 29
DistributionalFCStateQFunctionWithDiscreteAction (class in pfrl.q_functions), 30
DistributionalSingleModelStateQFunctionWithDiscreteAction (class in pfrl.q_functions), 30
DoubleDQN (class in pfrl.agents), 11
DoublePAL (class in pfrl.agents), 11
DPP (class in pfrl.agents), 11
DQN (class in pfrl.agents), 12
DuelingDQN (class in pfrl.q_functions), 29

E
EmpiricalNormalization (class in pfrl.nn), 26
EpisodicReplayBuffer (class in pfrl.replay_buffers), 34
evaluate_actions() (pfrl.action_value.ActionValue method), 5
Explorer (class in pfrl.explorers), 24

F
FactorizedNoisyLinear (class in pfrl.nn), 26
FCBNLateActionSAQFunction (class in pfrl.q_functions), 32
FCBNStateQFunctionWithDiscreteAction (class in pfrl.q_functions), 31
FCLateActionSAQFunction (class in pfrl.q_functions), 32
FCLSTMStateQFunctionWithDiscreteAction (class in pfrl.q_functions), 29
forward() (pfrl.nn.Recurrent method), 26

G
GaussianHeadWithDiagonalCovariance (class in pfrl.policies), 28
GaussianHeadWithFixedCovariance (class in pfrl.policies), 28
GaussianHeadWithStateIndependentCovariance (class in pfrl.policies), 28

39
generate_exp_id() (in module pfrl.experiments), 23
get_statistics() (pfrl.agent.Agent method), 6
Greedy (class in pfrl.explorers), 25
greedy_actions (pfrl.action_value.ActionValue attribute), 5

I
IQN (class in pfrl.agents), 13

L
LargeAtariCNN (class in pfrl.nn), 27
LinearDecayEpsilonGreedy (class in pfrl.explorers), 25
LinearInterpolationHook (class in pfrl.experiments), 23
load() (pfrl.agent.Agent method), 6
load() (pfrl.replay_buffers.ReplayBuffer method), 33

M
max (pfrl.action_value.ActionValue attribute), 5
MLP (class in pfrl.nn), 26
MLPBN (class in pfrl.nn), 27

O
observe() (pfrl.agent.Agent method), 6

P
PAL (class in pfrl.agents), 13
params (pfrl.action_value.ActionValue attribute), 5
PersistentEpisodicReplayBuffer (class in pfrl.replay_buffers), 35
PersistentReplayBuffer (class in pfrl.replay_buffers), 34
PPO (class in pfrl.agents), 13
prepare_output_dir() (in module pfrl.experiments), 23
PrioritiedEpisodicReplayBuffer (class in pfrl.replay_buffers), 34
PrioritiedReplayBuffer (class in pfrl.replay_buffers), 34

Q
QuadraticActionValue (class in pfrl.action_value), 5

R
Recurrent (class in pfrl.nn), 25
RecurrentBranched (class in pfrl.nn), 27
RecurrentSequential (class in pfrl.nn), 27
REINFORCE (class in pfrl.agents), 14
ReplayBuffer (class in pfrl.replay_buffers), 33, 34