
d3rlpy

Nov 26, 2020

1	Getting Started	3
1.1	Install	3
1.2	Prepare Dataset	3
1.3	Setup Algorithm	4
1.4	Setup Metrics	4
1.5	Start Training	4
1.6	Save and Load	5
2	Jupyter Notebooks	7
3	API Reference	9
3.1	Algorithms	9
3.2	Q Functions	126
3.3	MDPDataSet	126
3.4	Datasets	136
3.5	Preprocessing	137
3.6	Optimizers	142
3.7	Network Architectures	146
3.8	Data Augmentation	152
3.9	Metrics	161
3.10	Save and Load	169
3.11	Logging	171
3.12	scikit-learn compatibility	171
3.13	Online Training	174
3.14	Model-based Data Augmentation	179
4	Installation	185
4.1	Recommended Platforms	185
4.2	Install d3rlpy	185
5	License	187
6	Indices and tables	189
	Python Module Index	191
	Index	193

d3rlpy is a easy-to-use data-driven deep reinforcement learning library.

d3rlpy provides state-of-the-art data-driven deep reinforcement learning algorithms through out-of-the-box scikit-learn-style APIs. Unlike other RL libraries, the provided algorithms can achieve extremely powerful performance beyond the paper via several tweaks.

CHAPTER 1

Getting Started

This tutorial is also available on [Google Colaboratory](#)

1.1 Install

First of all, let's install `d3rlpy` on your machine:

```
$ pip install d3rlpy
```

Note: `d3rlpy` supports Python 3.6+. Make sure which version you use.

Note: If you use GPU, please setup CUDA first.

1.2 Prepare Dataset

You can make your own dataset without any efforts. In this tutorial, let's use integrated datasets to start. If you want to make a new dataset, see *MDP Dataset*.

`d3rlpy` provides suites of datasets for testing algorithms and research. See more documents at *Datasets*.

```
from d3rlpy.datasets import get_cartpole # CartPole-v0 dataset
from d3rlpy.datasets import get_pendulum # Pendulum-v0 dataset
from d3rlpy.datasets import get_pybullet # PyBullet task datasets
from d3rlpy.datasets import get_atari   # Atari 2600 task datasets
```

Here, we use the CartPole dataset to instantly check training results.

```
dataset, env = get_cartpole()
```

One interesting feature of d3rlpy is full compatibility with scikit-learn utilities. You can split `dataset` into a training dataset and a test dataset just like supervised learning as follows.

```
from sklearn.model_selection import train_test_split

train_episodes, test_episodes = train_test_split(dataset, test_size=0.2)
```

1.3 Setup Algorithm

There are many algorithms available in d3rlpy. Since CartPole is the simple task, let's start from DQN, which is the Q-learning algorithm proposed as the first deep reinforcement learning algorithm.

```
from d3rlpy.algos import DQN

# if you don't use GPU, set use_gpu=False instead.
dqn = DQN(use_gpu=True)
```

See more algorithms and configurations at [Algorithms](#).

1.4 Setup Metrics

Collecting evaluation metrics is important to train algorithms properly. In d3rlpy, the metrics is computed through scikit-learn style scorer functions.

```
from d3rlpy.metrics.scorer import td_error_scorer
from d3rlpy.metrics.scorer import average_value_estimation_scorer

# calculate metrics with test dataset
td_error = td_error_scorer(dqn, test_episodes)
```

Since evaluating algorithms without access to environment is still difficult, the algorithm can be directly evaluated with `evaluate_on_environment` function if the environment is available to interact.

```
from d3rlpy.metrics.scorer import evaluate_on_environment

# set environment in scorer function
evaluate_scorer = evaluate_on_environment(env)

# evaluate algorithm on the environment
rewards = evaluate_scorer(dqn)
```

See more metrics and configurations at [Metrics](#).

1.5 Start Training

Now, you have all to start data-driven training.


```
dqn.fit(train_episodes,
        eval_episodes=test_episodes,
        n_epochs=10,
        scorers={
            'td_error': td_error_scorer,
            'value_scale': average_value_estimation_scorer,
            'environment': evaluate_scorer
        })
```

Then, you will see training progress in the console like below:

```
augmentation=[]
batch_size=32
bootstrap=False
dynamics=None
encoder_params={}
eps=0.00015
gamma=0.99
learning_rate=6.25e-05
n_augmentations=1
n_critics=1
n_frames=1
q_func_type=mean
scaler=None
share_encoder=False
target_update_interval=8000.0
use_batch_norm=True
use_gpu=None
observation_shape=(4,)
action_size=2
100%| 2490/2490 [00:24<00:00, 100.63it/s]
epoch=0 step=2490 value_loss=0.190237
epoch=0 step=2490 td_error=1.483964
epoch=0 step=2490 value_scale=1.241220
epoch=0 step=2490 environment=157.400000
100%| 2490/2490 [00:24<00:00, 100.63it/s]
.
.
.
```

See more about logging at [Logging](#).

Once the training is done, your algorithm is ready to make decisions.

```
observation = env.reset()

# return actions based on the greedy-policy
action = dqn.predict([observation])[0]

# estimate action-values
value = dqn.predict_value([observation], [action])[0]
```

1.6 Save and Load

d3rlpy provides several ways to save trained models.

```
# save full parameters
dqn.save_model('dqn.pt')

# load full parameters
dqn2 = DQN()
dqn2.load_model('dqn.pt')

# save the greedy-policy as TorchScript
dqn.save_policy('policy.pt')

# save the greedy-policy as ONNX
dqn.save_policy('policy.onnx', as_onnx=True)
```

See more information at *Save and Load*.

CHAPTER 2

Jupyter Notebooks

- CartPole
- Toy task (line tracer)
- Continuous Control with PyBullet
- Discrete Control with Atari

3.1 Algorithms

d3rlpy provides state-of-the-art data-driven deep reinforcement learning algorithms as well as online algorithms for the base implementations.

3.1.1 Continuous control algorithms

<code>d3rlpy.algos.BC</code>	Behavior Cloning algorithm.
<code>d3rlpy.algos.DDPG</code>	Deep Deterministic Policy Gradients algorithm.
<code>d3rlpy.algos.TD3</code>	Twin Delayed Deep Deterministic Policy Gradients algorithm.
<code>d3rlpy.algos.SAC</code>	Soft Actor-Critic algorithm.
<code>d3rlpy.algos.BCQ</code>	Batch-Constrained Q-learning algorithm.
<code>d3rlpy.algos.BEAR</code>	Bootstrapping Error Accumulation Reduction algorithm.
<code>d3rlpy.algos.CQL</code>	Conservative Q-Learning algorithm.
<code>d3rlpy.algos.AWR</code>	Advantage-Weighted Regression algorithm.
<code>d3rlpy.algos.AWAC</code>	Advantage Weighted Actor-Critic algorithm.

d3rlpy.algos.BC

```
class d3rlpy.algos.BC(*, learning_rate=0.001, optim_factory=<d3rlpy.optimizers.AdamFactory
                        object>, encoder_factory='default', batch_size=100, n_frames=1,
                        use_gpu=False, scaler=None, augmentation=None, n_augmentations=1,
                        dynamics=None, impl=None, **kwargs)
```

Behavior Cloning algorithm.

Behavior Cloning (BC) is to imitate actions in the dataset via a supervised learning approach. Since BC is only imitating action distributions, the performance will be close to the mean of the dataset even though BC mostly

works better than online RL algorithms.

$$L(\theta) = \mathbb{E}_{a_t, s_t \sim D} [(a_t - \pi_\theta(s_t))^2]$$

Parameters

- **learning_rate** (*float*) – learning rate.
- **optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory.
- **encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **use_gpu** (*bool*, *int* or `d3rlpy.gpu.Device`) – flag to use GPU, device ID or device.
- **scaler** (`d3rlpy.preprocessingScaler` or *str*) – preprocessor. The available options are [*'pixel'*, *'min_max'*, *'standard'*]
- **augmentation** (`d3rlpy.augmentation.AugmentationPipeline` or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.
- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model for data augmentation.
- **impl** (`d3rlpy.algos.torch.bc_impl.BCImpl`) – implementation of the algorithm.

learning_rate

learning rate.

Type *float*

optim_factory

optimizer factory.

Type `d3rlpy.optimizers.OptimizerFactory`

encoder_factory

encoder factory.

Type `d3rlpy.encoders.EncoderFactory`

batch_size

mini-batch size.

Type *int*

n_frames

the number of frames to stack for image observation.

Type *int*

use_gpu

GPU device.

Type `d3rlpy.gpu.Device`

scaler
preprocessor.
Type d3rlpy.preprocessing.Scaler

augmentation
augmentation pipeline.
Type d3rlpy.augmentation.AugmentationPipeline

n_augmentations
the number of data augmentations to update.
Type int

dynamics
dynamics model.
Type d3rlpy.dynamics.base.DynamicsBase

impl
implemenation of the algorithm.
Type d3rlpy.algos.torch.bc_impl.BCImpl

eval_results_
evaluation results.
Type dict

Methods

create_impl (*observation_shape*, *action_size*)
Instantiate implementation objects with the dataset shapes.
This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs=1000*, *save_metrics=True*, *experiment_name=None*, *with_timestamp=True*, *logdir='d3rlpy_logs'*, *verbose=True*, *show_progress=True*, *tensorboard=True*, *eval_episodes=None*, *save_interval=1*, *scorers=None*)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (d3rlpy.dataset.Episode)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.

- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

fit_online (*env*, *buffer*, *explorer=None*, *n_steps=1000000*, *n_steps_per_epoch=10000*,
update_interval=1, *update_start_step=0*, *eval_env=None*, *eval_epsilon=0.0*,
save_metrics=True, *experiment_name=None*, *with_timestamp=True*,
logdir='d3rlpy_logs', *verbose=True*, *show_progress=True*, *tensorboard=True*)

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (*d3rlpy.online.buffers.Buffer*) – replay buffer.
- **explorer** (*d3rlpy.online.explorers.Explorer*) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If *None*, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If *False*, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_online_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod from_json (*fname*, *use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.


```

from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)

```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.**Return type** *d3rlpy.base.LearnableBase***get_params** (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```

params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)

```

Parameters **deep** (*bool*) – flag to deeply copy objects such as *impl*.**Returns** attribute values in dictionary.**Return type** *dict***load_model** (*fname*)

Load neural network parameters.

```

algo.load_model('model.pt')

```

Parameters **fname** (*str*) – source file path.**predict** (*x*)

Returns greedy actions.

```

# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control

```

Parameters **x** (*numpy.ndarray*) – observations

Returns greedy actions

Return type `numpy.ndarray`

predict_value (*x*, *action*)

value prediction is not supported by BC algorithms.

sample_action (*x*)

sampling action is not supported by BC algorithm.

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters *fname* (*str*) – destination file path.

save_policy (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).
- <https://onnx.ai> (for ONNX)

Parameters

- *fname* (*str*) – destination file path.
- *as_onnx* (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ***params* – arbitrary inputs to set as attributes.

Returns itself.

Return type `d3rlpy.algos.base.AlgoBase`

update (*epoch*, *itr*, *batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (`d3rlpy.dataset.TransitionMiniBatch`) – mini-batch data.

Returns loss values.

Return type `list`

d3rlpy.algos.DDPG

```
class d3rlpy.algos.DDPG(*, actor_learning_rate=0.0003, critic_learning_rate=0.0003,
                        actor_optim_factory=<d3rlpy.optimizers.AdamFactory object>,
                        critic_optim_factory=<d3rlpy.optimizers.AdamFactory object>, actor_encoder_factory='default',
                        critic_encoder_factory='default', batch_size=100, n_frames=1, gamma=0.99, tau=0.005, n_critics=1,
                        bootstrap=False, share_encoder=False, regularizing_rate=1e-10, use_batch_norm=False, q_func_type='mean', use_gpu=False,
                        scaler=None, augmentation=None, n_augmentations=1, encoder_params={}, dynamics=None, impl=None, **kwargs)
```

Deep Deterministic Policy Gradients algorithm.

DDPG is an actor-critic algorithm that trains a Q function parametrized with θ and a policy function parametrized with ϕ .

$$L(\theta) = \mathbb{E}_{s_t, a_t, r_{t+1}, s_{t+1} \sim D} [(r_{t+1} + \gamma Q_{\theta'}(s_{t+1}, \pi_{\phi'}(s_{t+1})) - Q_{\theta}(s_t, a_t))^2]$$

$$J(\phi) = \mathbb{E}_{s_t \sim D} [Q_{\theta}(s_t, \pi_{\phi}(s_t))]$$

where θ' and ϕ are the target network parameters. There target network parameters are updated every iteration.

$$\begin{aligned}\theta' &\leftarrow \tau\theta + (1 - \tau)\theta' \\ \phi' &\leftarrow \tau\phi + (1 - \tau)\phi'\end{aligned}$$

References

- Silver et al., Deterministic policy gradient algorithms.
- Lillicrap et al., Continuous control with deep reinforcement learning.

Parameters

- **actor_learning_rate** (*float*) – learning rate for policy function.
- **critic_learning_rate** (*float*) – learning rate for Q function.
- **actor_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **actor_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the actor.
- **critic_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the critic.
- **batch_size** (*int*) – mini-batch size.

- **n_frames** (*int*) – the number of frames to stack for image observation.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficient.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share_encoder** (*bool*) – flag to share encoder network.
- **reguralizing_rate** (*float*) – reguralizing term for policy function.
- **q_func_type** (*str*) – type of Q function. Available options are ['mean', 'qr', 'iqn', 'fqf'].
- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are ['pixel', 'min_max', 'standard']
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.ddpg_impl.DDPGImpl*) – algorithm implementation.

actor_learning_rate

learning rate for policy function.

Type *float*

critic_learning_rate

learning rate for Q function.

Type *float*

actor_optim_factory

optimizer factory for the actor.

Type *d3rlpy.optimizers.OptimizerFactory*

critic_optim_factory

optimizer factory for the critic.

Type *d3rlpy.optimizers.OptimizerFactory*

actor_encoder_factory

encoder factory for the actor.

Type *d3rlpy.encoders.EncoderFactory*

critic_encoder_factory

encoder factory for the critic.

Type *d3rlpy.encoders.EncoderFactory*

batch_size

mini-batch size.

Type `int`

n_frames
the number of frames to stack for image observation.

Type `int`

gamma
discount factor.

Type `float`

tau
target network synchronization coefficient.

Type `float`

n_critics
the number of Q functions for ensemble.

Type `int`

bootstrap
flag to bootstraep Q functions.

Type `bool`

share_encoder
flag to share encoder network.

Type `bool`

reguralizing_rate
reguralizing term for policy function.

Type `float`

q_func_type
type of Q function.

Type `str`

use_gpu
GPU device.

Type `d3rlpy.gpu.Device`

scaler
preprocessor.

Type `d3rlpy.preprocessing.Scaler`

augmentation
augmentation pipeline.

Type `d3rlpy.augmentation.AugmentationPipeline`

n_augmentations
the number of data augmentations to update.

Type `int`

dynamics
dynamics model.

Type `d3rlpy.dynamics.base.DynamicsBase`

impl
algorithm implementation.
Type `d3rlpy.algos.torch.ddpg_impl.DDPGImpl`

eval_results_
evaluation results.
Type `dict`

Methods

create_impl (*observation_shape*, *action_size*)
Instantiate implementation objects with the dataset shapes.
This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs=1000*, *save_metrics=True*, *experiment_name=None*, *with_timestamp=True*, *logdir='d3rlpy_logs'*, *verbose=True*, *show_progress=True*, *tensorboard=True*, *eval_episodes=None*, *save_interval=1*, *scorers=None*)
Trains with the given dataset.

`algo.fit(episodes)`

Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

```
fit_online (env, buffer, explorer=None, n_steps=1000000, n_steps_per_epoch=10000,
            update_interval=1, update_start_step=0, eval_env=None, eval_epsilon=0.0,
            save_metrics=True, experiment_name=None, with_timestamp=True,
            logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard=True)
```

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (*d3rlpy.online.buffers.Buffer*) – replay buffer.
- **explorer** (*d3rlpy.online.explorers.Explorer*) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If False, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_online_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

```
classmethod from_json (fname, use_gpu=False)
```

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.

- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters **deep** (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type *dict*

load_model (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters **fname** (*str*) – source file path.

predict (*x*)

Returns greedy actions.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters **x** (*numpy.ndarray*) – observations

Returns greedy actions

Return type *numpy.ndarray*

predict_value (*x, action, with_std=False*)

Returns predicted action-values.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

# for continuous control
# 100 actions with shape of (2,)
actions = np.random.random((100, 2))
```

(continues on next page)

(continued from previous page)

```
# for discrete control
# 100 actions in integer values
actions = np.random.randint(2, size=100)

values = algo.predict_value(x, actions)
# values.shape == (100,)

values, stds = algo.predict_value(x, actions, with_std=True)
# stds.shape == (100,)
```

Parameters

- **x** (*numpy.ndarray*) – observations
- **action** (*numpy.ndarray*) – actions
- **with_std** (*bool*) – flag to return standard deviation of ensemble estimation. This deviation reflects uncertainty for the given observations. This uncertainty will be more accurate if you enable *bootstrap* flag and increase *n_critics* value.

Returns predicted action-values**Return type** *numpy.ndarray***sample_action** (*x*)

Returns sampled actions.

The sampled actions are identical to the output of *predict* method if the policy is deterministic.**Parameters** **x** (*numpy.ndarray*) – observations.**Returns** sampled actions.**Return type** *numpy.ndarray***save_model** (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.**save_policy** (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).

- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type `d3rlpy.algos.base.AlgoBase`

update (*epoch, itr, batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (`d3rlpy.dataset.TransitionMiniBatch`) – mini-batch data.

Returns loss values.

Return type `list`

d3rlpy.algos.TD3

```
class d3rlpy.algos.TD3(*, actor_learning_rate=0.0003, critic_learning_rate=0.0003,
                        actor_optim_factory=<d3rlpy.optimizers.AdamFactory object>,
                        critic_optim_factory=<d3rlpy.optimizers.AdamFactory object>, ac-
                        tor_encoder_factory='default', critic_encoder_factory='default',
                        batch_size=100, n_frames=1, gamma=0.99, tau=0.005, reguraliz-
                        ing_rate=0.0, n_critics=2, bootstrap=False, share_encoder=False,
                        target_smoothing_sigma=0.2, target_smoothing_clip=0.5, up-
                        date_actor_interval=2, use_batch_norm=False, q_func_type='mean',
                        use_gpu=False, scaler=None, augmentation=[], n_augmentations=1,
                        encoder_params={}, dynamics=None, impl=None, **kwargs)
```

Twin Delayed Deep Deterministic Policy Gradients algorithm.

TD3 is an improved DDPG-based algorithm. Major differences from DDPG are as follows.

- TD3 has twin Q functions to reduce overestimation bias at TD learning. The number of Q functions can be designated by *n_critics*.
- TD3 adds noise to target value estimation to avoid overfitting with the deterministic policy.
- TD3 updates the policy function after several Q function updates in order to reduce variance of action-value estimation. The interval of the policy function update can be designated by *update_actor_interval*.

$$L(\theta_i) = \mathbb{E}_{s_t, a_t, r_{t+1}, s_{t+1} \sim D} [(r_{t+1} + \gamma \min_j Q_{\theta'_j}(s_{t+1}, \pi_{\phi'}(s_{t+1}) + \epsilon) - Q_{\theta_i}(s_t, a_t))^2]$$

$$J(\phi) = \mathbb{E}_{s_t \sim D} [\min_i Q_{\theta_i}(s_t, \pi_{\phi}(s_t))]$$

where $\epsilon \sim \text{clip}(N(0, \sigma), -c, c)$

References

- Fujimoto et al., Addressing Function Approximation Error in Actor-Critic Methods.

Parameters

- **actor_learning_rate** (*float*) – learning rate for a policy function.
- **critic_learning_rate** (*float*) – learning rate for Q functions.
- **actor_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **actor_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the actor.
- **critic_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the critic.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficient.
- **reguralizing_rate** (*float*) – reguralizing term for policy function.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share_encoder** (*bool*) – flag to share encoder network.
- **target_smoothing_sigma** (*float*) – standard deviation for target noise.
- **target_smoothing_clip** (*float*) – clipping range for target noise.
- **update_actor_interval** (*int*) – interval to update policy function described as *delayed policy update* in the paper.
- **q_func_type** (*str*) – type of Q function. Available options are [*'mean'*, *'qr'*, *'iqn'*, *'fqf'*].
- **use_gpu** (*bool*, *int* or `d3rlpy.gpu.Device`) – flag to use GPU, device ID or device.
- **scaler** (`d3rlpy.preprocessingScaler` or *str*) – preprocessor. The available options are [*'pixel'*, *'min_max'*, *'standard'*]
- **augmentation** (`d3rlpy.augmentation.AugmentationPipeline` or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.

- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.td3_impl.TD3Impl*) – algorithm implementation.

actor_learning_rate

learning rate for a policy function.

Type *float*

critic_learning_rate

learning rate for Q functions.

Type *float*

actor_optim_factory

optimizer factory for the actor.

Type *d3rlpy.optimizers.OptimizerFactory*

critic_optim_factory

optimizer factory for the critic.

Type *d3rlpy.optimizers.OptimizerFactory*

actor_encoder_factory

encoder factory for the actor.

Type *d3rlpy.encoders.EncoderFactory*

critic_encoder_factory

encoder factory for the critic.

Type *d3rlpy.encoders.EncoderFactory*

batch_size

mini-batch size.

Type *int*

n_frames

the number of frames to stack for image observation.

Type *int*

gamma

discount factor.

Type *float*

tau

target network synchronization coefficient.

Type *float*

reguralizing_rate

reguralizing term for policy function.

Type *float*

n_critics

the number of Q functions for ensemble.

Type *int*

bootstrap
flag to bootstrap Q functions.
Type `bool`

share_encoder
flag to share encoder network.
Type `bool`

target_smoothing_sigma
standard deviation for target noise.
Type `float`

target_smoothing_clip
clipping range for target noise.
Type `float`

update_actor_interval
interval to update policy function described as *delayed policy update* in the paper.
Type `int`

q_func_type
type of Q function..
Type `str`

use_gpu
GPU device.
Type `d3rlpy.gpu.Device`

scaler
preprocessor.
Type `d3rlpy.preprocessing.Scaler`

augmentation
augmentation pipeline.
Type `d3rlpy.augmentation.AugmentationPipeline`

n_augmentations
the number of data augmentations to update.
Type `int`

dynamics
dynamics model.
Type `d3rlpy.dynamics.base.DynamicsBase`

impl
algorithm implementation.
Type `d3rlpy.algos.torch.td3_impl.TD3Impl`

eval_results_
evaluation results.
Type `dict`

Methods

create_impl (*observation_shape*, *action_size*)

Instantiate implementation objects with the dataset shapes.

This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs*=1000, *save_metrics*=True, *experiment_name*=None, *with_timestamp*=True, *logdir*='d3rlpy_logs', *verbose*=True, *show_progress*=True, *tensorboard*=True, *eval_episodes*=None, *save_interval*=1, *scorers*=None)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

fit_online (*env*, *buffer*, *explorer*=None, *n_steps*=1000000, *n_steps_per_epoch*=10000, *update_interval*=1, *update_start_step*=0, *eval_env*=None, *eval_epsilon*=0.0, *save_metrics*=True, *experiment_name*=None, *with_timestamp*=True, *logdir*='d3rlpy_logs', *verbose*=True, *show_progress*=True, *tensorboard*=True)
Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (*d3rlpy.online.buffers.Buffer*) – replay buffer.
- **explorer** (*d3rlpy.online.explorers.Explorer*) – action explorer.

- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If False, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_online_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod from_json (*fname, use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool, int or d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters `deep` (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type `dict`

load_model (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters `fname` (*str*) – source file path.

predict (*x*)

Returns greedy actions.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters `x` (*numpy.ndarray*) – observations

Returns greedy actions

Return type `numpy.ndarray`

predict_value (*x, action, with_std=False*)

Returns predicted action-values.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

# for continuous control
# 100 actions with shape of (2,)
actions = np.random.random((100, 2))

# for discrete control
# 100 actions in integer values
actions = np.random.randint(2, size=100)

values = algo.predict_value(x, actions)
# values.shape == (100,)

values, stds = algo.predict_value(x, actions, with_std=True)
# stds.shape == (100,)
```

Parameters

- `x` (*numpy.ndarray*) – observations

- **action** (*numpy.ndarray*) – actions
- **with_std** (*bool*) – flag to return standard deviation of ensemble estimation. This deviation reflects uncertainty for the given observations. This uncertainty will be more accurate if you enable *bootstrap* flag and increase *n_critics* value.

Returns predicted action-values

Return type *numpy.ndarray*

sample_action (*x*)

Returns sampled actions.

The sampled actions are identical to the output of *predict* method if the policy is deterministic.

Parameters **x** (*numpy.ndarray*) – observations.

Returns sampled actions.

Return type *numpy.ndarray*

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.

save_policy (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).
- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type `d3rlpy.algos.base.AlgoBase`

update (*epoch*, *total_step*, *batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (`d3rlpy.dataset.TransitionMiniBatch`) – mini-batch data.

Returns loss values.

Return type `list`

d3rlpy.algos.SAC

```
class d3rlpy.algos.SAC(*, actor_learning_rate=0.0003, critic_learning_rate=0.0003,
    temp_learning_rate=0.0003, actor_optim_factory=<d3rlpy.optimizers.AdamFactory
    object>, critic_optim_factory=<d3rlpy.optimizers.AdamFactory object>,
    temp_optim_factory=<d3rlpy.optimizers.AdamFactory object>,
    actor_encoder_factory='default', critic_encoder_factory='default',
    batch_size=100, n_frames=1, gamma=0.99, tau=0.005, n_critics=2,
    bootstrap=False, share_encoder=False, update_actor_interval=2,
    initial_temperature=1.0, q_func_type='mean', use_gpu=False, scaler=None,
    augmentation=[], n_augmentations=1, dynamics=None, impl=None,
    **kwargs)
```

Soft Actor-Critic algorithm.

SAC is a DDPG-based maximum entropy RL algorithm, which produces state-of-the-art performance in online RL settings. SAC leverages twin Q functions proposed in TD3. Additionally, *delayed policy update* in TD3 is also implemented, which is not done in the paper.

$$L(\theta_i) = \mathbb{E}_{s_t, a_t, r_{t+1}, s_{t+1} \sim D, a_{t+1} \sim \pi_{\phi}(\cdot | s_{t+1})} [(y - Q_{\theta_i}(s_t, a_t))^2]$$

$$y = r_{t+1} + \gamma (\min_j Q_{\theta_j}(s_{t+1}, a_{t+1}) - \alpha \log(\pi_{\phi}(a_{t+1} | s_{t+1})))$$

$$J(\phi) = \mathbb{E}_{s_t \sim D, a_t \sim \pi_{\phi}(\cdot | s_t)} [\alpha \log(\pi_{\phi}(a_t | s_t)) - \min_i Q_{\theta_i}(s_t, \pi_{\phi}(a_t | s_t))]$$

The temperature parameter α is also automatically adjustable.

$$J(\alpha) = \mathbb{E}_{s_t \sim D, a_t \sim \pi_{\phi}(\cdot | s_t)} [-\alpha (\log(\pi_{\phi}(a_t | s_t)) + H)]$$

where H is a target entropy, which is defined as $\dim a$.

References

- Haarnoja et al., Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.
- Haarnoja et al., Soft Actor-Critic Algorithms and Applications.

Parameters

- **actor_learning_rate** (*float*) – learning rate for policy function.
- **critic_learning_rate** (*float*) – learning rate for Q functions.
- **temp_learning_rate** (*float*) – learning rate for temperature parameter.
- **actor_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **temp_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the temperature.
- **actor_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the actor.
- **critic_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the critic.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficient.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share_encoder** (*bool*) – flag to share encoder network.
- **update_actor_interval** (*int*) – interval to update policy function.
- **initial_temperature** (*float*) – initial temperature value.
- **q_func_type** (*str*) – type of Q function. Available options are [*'mean'*, *'qr'*, *'iqn'*, *'fqf'*].
- **use_gpu** (*bool*, *int* or `d3rlpy.gpu.Device`) – flag to use GPU, device ID or device.
- **scaler** (`d3rlpy.preprocessingScaler` or *str*) – preprocessor. The available options are [*'pixel'*, *'min_max'*, *'standard'*].
- **augmentation** (`d3rlpy.augmentation.AugmentationPipeline` or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.
- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model for data augmentation.
- **impl** (`d3rlpy.algos.torch.sac_impl.SACImpl`) – algorithm implementation.

actor_learning_rate

learning rate for policy function.

Type *float***critic_learning_rate**

learning rate for Q functions.

Type *float*

temp_learning_rate
learning rate for temperature parameter.
Type `float`

actor_optim_factory
optimizer factory for the actor.
Type `d3rlpy.optimizers.OptimizerFactory`

critic_optim_factory
optimizer factory for the critic.
Type `d3rlpy.optimizers.OptimizerFactory`

temp_optim_factory
optimizer factory for the temperature.
Type `d3rlpy.optimizers.OptimizerFactory`

actor_encoder_factory
encoder factory for the actor.
Type `d3rlpy.encoders.EncoderFactory`

critic_encoder_factory
encoder factory for the critic.
Type `d3rlpy.encoders.EncoderFactory`

batch_size
mini-batch size.
Type `int`

n_frames
the number of frames to stack for image observation.
Type `int`

gamma
discount factor.
Type `float`

tau
target network synchronization coefficient.
Type `float`

n_critics
the number of Q functions for ensemble.
Type `int`

bootstrap
flag to bootstrap Q functions.
Type `bool`

share_encoder
flag to share encoder network.
Type `bool`

update_actor_interval
interval to update policy function.

Type `int`

initial_temperature
initial temperature value.

Type `float`

q_func_type
type of Q function.

Type `str`

use_gpu
GPU device.

Type `d3rlpy.gpu.Device`

scaler
preprocessor.

Type `d3rlpy.preprocessing.Scaler`

augmentation
augmentation pipeline.

Type `d3rlpy.augmentation.AugmentationPipeline`

n_augmentations
the number of data augmentations to update.

Type `int`

dynamics
dynamics model.

Type `d3rlpy.dynamics.base.DynamicsBase`

impl
algorithm implementation.

Type `d3rlpy.algos.torch.sac_impl.SACImpl`

eval_results_
evaluation results.

Type `dict`

Methods

create_impl (*observation_shape*, *action_size*)
Instantiate implementation objects with the dataset shapes.
This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs=1000*, *save_metrics=True*, *experiment_name=None*, *with_timestamp=True*, *logdir='d3rlpy_logs'*, *verbose=True*, *show_progress=True*, *tensorboard=True*, *eval_episodes=None*, *save_interval=1*, *scorers=None*)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

```
fit_online (env, buffer, explorer=None, n_steps=1000000, n_steps_per_epoch=10000,  
            update_interval=1, update_start_step=0, eval_env=None, eval_epsilon=0.0,  
            save_metrics=True, experiment_name=None, with_timestamp=True,  
            logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard=True)
```

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (`d3rlpy.online.buffers.Buffer`) – replay buffer.
- **explorer** (`d3rlpy.online.explorers.Explorer`) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If False, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_online_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.

- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod `from_json` (*fname*, *use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters **deep** (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type *dict*

load_model (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters **fname** (*str*) – source file path.

predict (*x*)

Returns greedy actions.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters *x* (*numpy.ndarray*) – observations

Returns greedy actions

Return type *numpy.ndarray*

predict_value (*x, action, with_std=False*)

Returns predicted action-values.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

# for continuous control
# 100 actions with shape of (2,)
actions = np.random.random((100, 2))

# for discrete control
# 100 actions in integer values
actions = np.random.randint(2, size=100)

values = algo.predict_value(x, actions)
# values.shape == (100,)

values, stds = algo.predict_value(x, actions, with_std=True)
# stds.shape == (100,)
```

Parameters

- *x* (*numpy.ndarray*) – observations
- *action* (*numpy.ndarray*) – actions
- *with_std* (*bool*) – flag to return standard deviation of ensemble estimation. This deviation reflects uncertainty for the given observations. This uncertainty will be more accurate if you enable *bootstrap* flag and increase *n_critics* value.

Returns predicted action-values

Return type *numpy.ndarray*

sample_action (*x*)

Returns sampled actions.

The sampled actions are identical to the output of *predict* method if the policy is deterministic.

Parameters *x* (*numpy.ndarray*) – observations.

Returns sampled actions.

Return type *numpy.ndarray*

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.

save_policy (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).
- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type d3rlpy.algos.base.AlgoBase

update (*epoch*, *total_step*, *batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (d3rlpy.dataset.TransitionMiniBatch) – mini-batch data.

Returns loss values.

Return type list

d3rlpy.algos.BCQ

```
class d3rlpy.algos.BCQ(*, actor_learning_rate=0.001, critic_learning_rate=0.001, imita-
tor_learning_rate=0.001, actor_optim_factory=<d3rlpy.optimizers.AdamFactory
object>, critic_optim_factory=<d3rlpy.optimizers.AdamFactory object>,
imitator_optim_factory=<d3rlpy.optimizers.AdamFactory object>, ac-
tor_encoder_factory='default', critic_encoder_factory='default', imita-
tor_encoder_factory='default', batch_size=100, n_frames=1, gamma=0.99,
tau=0.005, n_critics=2, bootstrap=False, share_encoder=False,
update_actor_interval=1, lam=0.75, n_action_samples=100, ac-
tion_flexibility=0.05, rl_start_epoch=0, latent_size=32, beta=0.5,
q_func_type='mean', use_gpu=False, scaler=None, augmentation=None,
n_augmentations=1, dynamics=None, impl=None, **kwargs)
```

Batch-Constrained Q-learning algorithm.

BCQ is the very first practical data-driven deep reinforcement learning algorithm. The major difference from DDPG is that the policy function is represented as combination of conditional VAE and perturbation function in order to remedy extrapolation error emerging from target value estimation.

The encoder and the decoder of the conditional VAE is represented as E_ω and D_ω respectively.

$$L(\omega) = E_{s_t, a_t \sim D}[(a - \tilde{a})^2 + D_{KL}(N(\mu, \sigma) | N(0, 1))]$$

where $\mu, \sigma = E_\omega(s_t, a_t)$, $\tilde{a} = D_\omega(s_t, z)$ and $z \sim N(\mu, \sigma)$.

The policy function is represented as a residual function with the VAE and the perturbation function represented as $\xi_\phi(s, a)$.

$$\pi(s, a) = a + \Phi \xi_\phi(s, a)$$

where $a = D_\omega(s, z)$, $z \sim N(0, 0.5)$ and Φ is a perturbation scale designated by *action_flexibility*. Although the policy is learned closely to data distribution, the perturbation function can lead to more rewarded states.

BCQ also leverages twin Q functions and computes weighted average over maximum values and minimum values.

$$L(\theta_i) = \mathbb{E}_{s_t, a_t, r_{t+1}, s_{t+1} \sim D}[(y - Q_{\theta_i}(s_t, a_t))^2]$$

$$y = r_{t+1} + \gamma \max_{a_i} [\lambda \min_j Q_{\theta'_j}(s_{t+1}, a_i) + (1 - \lambda) \max_j Q_{\theta_j}(s_{t+1}, a_i)]$$

where $\{a_i \sim D(s_{t+1}, z), z \sim N(0, 0.5)\}_{i=1}^n$. The number of sampled actions is designated with *n_action_samples*.

Finally, the perturbation function is trained just like DDPG's policy function.

$$J(\phi) = \mathbb{E}_{s_t \sim D, a_t \sim D_\omega(s_t, z), z \sim N(0, 0.5)}[Q_{\theta_1}(s_t, \pi(s_t, a_t))]$$

At inference time, action candidates are sampled as many as *n_action_samples*, and the action with highest value estimation is taken.

$$\pi'(s) = \operatorname{argmax}_{\pi(s, a_i)} Q_{\theta_1}(s, \pi(s, a_i))$$

Note: The greedy action is not deterministic because the action candidates are always randomly sampled. This might affect *save_policy* method and the performance at production.

References

- Fujimoto et al., Off-Policy Deep Reinforcement Learning without Exploration.

Parameters

- **actor_learning_rate** (*float*) – learning rate for policy function.
- **critic_learning_rate** (*float*) – learning rate for Q functions.
- **imitator_learning_rate** (*float*) – learning rate for Conditional VAE.
- **actor_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **imitator_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the conditional VAE.
- **actor_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the actor.
- **critic_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the critic.
- **imitator_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the conditional VAE.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficient.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share_encoder** (*bool*) – flag to share encoder network.
- **update_actor_interval** (*int*) – interval to update policy function.
- **lam** (*float*) – weight factor for critic ensemble.
- **n_action_samples** (*int*) – the number of action samples to estimate action-values.
- **action_flexibility** (*float*) – output scale of perturbation function represented as Φ .
- **rl_start_epoch** (*int*) – epoch to start to update policy function and Q functions. If this is large, RL training would be more stabilized.
- **latent_size** (*int*) – size of latent vector for Conditional VAE.
- **beta** (*float*) – KL regularization term for Conditional VAE.
- **q_func_type** (*str*) – type of Q function. Available options are [*'mean'*, *'qr'*, *'iqn'*, *'fqf'*].
- **use_gpu** (*bool*, *int* or `d3rlpy.gpu.Device`) – flag to use GPU, device ID or device.

- **scaler** (*d3rlpy.preprocessing.Scaler* or *str*) – preprocessor. The available options are [*'pixel'*, *'min_max'*, *'standard'*]
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.bcq_impl.BCQImpl*) – algorithm implementation.

actor_learning_rate

learning rate for policy function.

Type *float*

critic_learning_rate

learning rate for Q functions.

Type *float*

imitator_learning_rate

learning rate for Conditional VAE.

Type *float*

actor_optim_factory

optimizer factory for the actor.

Type *d3rlpy.optimizers.OptimizerFactory*

critic_optim_factory

optimizer factory for the critic.

Type *d3rlpy.optimizers.OptimizerFactory*

imitator_optim_factory

optimizer factory for the conditional VAE.

Type *d3rlpy.optimizers.OptimizerFactory*

actor_encoder_factory

encoder factory for the actor.

Type *d3rlpy.encoders.EncoderFactory*

critic_encoder_factory

encoder factory for the critic.

Type *d3rlpy.encoders.EncoderFactory*

imitator_encoder_factory

encoder factory for the conditional VAE.

Type *d3rlpy.encoders.EncoderFactory*

batch_size

mini-batch size.

Type *int*

n_frames

the number of frames to stack for image observation.

Type `int`

gamma
discount factor.

Type `float`

tau
target network synchronization coefficient.

Type `float`

n_critics
the number of Q functions for ensemble.

Type `int`

bootstrap
flag to bootstrap Q functions.

Type `bool`

share_encoder
flag to share encoder network.

Type `bool`

update_actor_interval
interval to update policy function.

Type `int`

lam
weight factor for critic ensemble.

Type `float`

n_action_samples
the number of action samples to estimate action-values.

Type `int`

action_flexibility
output scale of perturbation function.

Type `float`

rl_start_epoch
epoch to start to update policy function and Q functions.

Type `int`

latent_size
size of latent vector for Conditional VAE.

Type `int`

beta
KL regularization term for Conditional VAE.

Type `float`

q_func_type
type of Q function.

Type `str`

use_gpu
GPU device.
Type d3rlpy.gpu.Device

scaler
preprocessor.
Type d3rlpy.preprocessingScaler

augmentation
augmentation pipeline.
Type d3rlpy.augmentation.AugmentationPipeline

n_augmentations
the number of data augmentations to update.
Type int

dynamics
dynamics model.
Type d3rlpy.dynamics.base.DynamicsBase

impl
algorithm implementation.
Type d3rlpy.algos.torch.bcq_impl.BCQImpl

eval_results_
evaluation results.
Type dict

Methods

create_impl (*observation_shape, action_size*)
Instantiate implementation objects with the dataset shapes.
This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes, n_epochs=1000, save_metrics=True, experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard=True, eval_episodes=None, save_interval=1, scorers=None*)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (d3rlpy.dataset.Episode)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.

- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

fit_online (*env*, *buffer*, *explorer=None*, *n_steps=1000000*, *n_steps_per_epoch=10000*, *update_interval=1*, *update_start_step=0*, *eval_env=None*, *eval_epsilon=0.0*, *save_metrics=True*, *experiment_name=None*, *with_timestamp=True*, *logdir='d3rlpy_logs'*, *verbose=True*, *show_progress=True*, *tensorboard=True*)

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (*d3rlpy.online.buffers.Buffer*) – replay buffer.
- **explorer** (*d3rlpy.online.explorers.Explorer*) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If False, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_online_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod `from_json` (*fname*, *use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters **deep** (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type *dict*

load_model (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters **fname** (*str*) – source file path.

predict (*x*)

Returns greedy actions.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
```

(continues on next page)

(continued from previous page)

```
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters **x** (*numpy.ndarray*) – observations

Returns greedy actions

Return type *numpy.ndarray*

predict_value (*x, action, with_std=False*)

Returns predicted action-values.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

# for continuous control
# 100 actions with shape of (2,)
actions = np.random.random((100, 2))

# for discrete control
# 100 actions in integer values
actions = np.random.randint(2, size=100)

values = algo.predict_value(x, actions)
# values.shape == (100,)

values, stds = algo.predict_value(x, actions, with_std=True)
# stds.shape == (100,)
```

Parameters

- **x** (*numpy.ndarray*) – observations
- **action** (*numpy.ndarray*) – actions
- **with_std** (*bool*) – flag to return standard deviation of ensemble estimation. This deviation reflects uncertainty for the given observations. This uncertainty will be more accurate if you enable *bootstrap* flag and increase *n_critics* value.

Returns predicted action-values

Return type *numpy.ndarray*

sample_action (*x*)

BCQ does not support sampling action.

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.

save_policy (*fname, as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).
- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type d3rlpy.algos.base.AlgoBase

update (*epoch, total_step, batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (d3rlpy.dataset.TransitionMiniBatch) – mini-batch data.

Returns loss values.

Return type list

d3rlpy.algos.BEAR

```
class d3rlpy.algos.BEAR(*, actor_learning_rate=0.0003, critic_learning_rate=0.0003,
    imitator_learning_rate=0.001, temp_learning_rate=0.0003, alpha_learning_rate=0.001,
    actor_optim_factory=<d3rlpy.optimizers.AdamFactory object>, critic_optim_factory=<d3rlpy.optimizers.AdamFactory object>,
    imitator_optim_factory=<d3rlpy.optimizers.AdamFactory object>, temp_optim_factory=<d3rlpy.optimizers.AdamFactory object>,
    alpha_optim_factory=<d3rlpy.optimizers.AdamFactory object>, actor_encoder_factory='default',
    critic_encoder_factory='default', imitator_encoder_factory='default', batch_size=100, n_frames=1,
    gamma=0.99, tau=0.005, n_critics=2, bootstrap=False, share_encoder=False, update_actor_interval=1,
    initial_temperature=1.0, initial_alpha=1.0, alpha_threshold=0.05, lam=0.75, n_action_samples=4,
    mmd_sigma=20.0, rl_start_epoch=0, q_func_type='mean', use_gpu=False, scaler=None,
    augmentation=None, n_augmentations=1, dynamics=None, impl=None, **kwargs)
```

Bootstrapping Error Accumulation Reduction algorithm.

BEAR is a SAC-based data-driven deep reinforcement learning algorithm.

BEAR constrains the support of the policy function within data distribution by minimizing Maximum Mean Discrepancy (MMD) between the policy function and the approximated behavior policy function $\pi_\beta(a|s)$ which is optimized through L2 loss.

$$L(\beta) = \mathbb{E}_{s_t, a_t \sim D, a \sim \pi_\beta(\cdot|s_t)} [(a - a_t)^2]$$

The policy objective is a combination of SAC's objective and MMD penalty.

$$J(\phi) = J_{SAC}(\phi) - \mathbb{E}_{s_t \sim D} \alpha (\text{MMD}(\pi_\beta(\cdot|s_t), \pi_\phi(\cdot|s_t)) - \epsilon)$$

where MMD is computed as follows.

$$\text{MMD}(x, y) = \frac{1}{N^2} \sum_{i, i'} k(x_i, x_{i'}) - \frac{2}{NM} \sum_{i, j} k(x_i, y_j) + \frac{1}{M^2} \sum_{j, j'} k(y_j, y_{j'})$$

where $k(x, y)$ is a gaussian kernel $k(x, y) = \exp(-(x - y)^2 / (2\sigma^2))$.

α is also adjustable through dual gradient descent where α becomes smaller if MMD is smaller than the threshold ϵ .

References

- Kumar et al., Stabilizing Off-Policy Q-Learning via Bootstrapping Error Reduction.

Parameters

- **actor_learning_rate** (*float*) – learning rate for policy function.
- **critic_learning_rate** (*float*) – learning rate for Q functions.
- **imitator_learning_rate** (*float*) – learning rate for behavior policy function.
- **temp_learning_rate** (*float*) – learning rate for temperature parameter.
- **alpha_learning_rate** (*float*) – learning rate for α .
- **actor_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the actor.

- **critic_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **imitator_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the behavior policy.
- **temp_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the temperature.
- **alpha_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for α .
- **actor_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or `str`) – encoder factory for the actor.
- **critic_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or `str`) – encoder factory for the critic.
- **imitator_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or `str`) – encoder factory for the behavior policy.
- **batch_size** (`int`) – mini-batch size.
- **n_frames** (`int`) – the number of frames to stack for image observation.
- **gamma** (`float`) – discount factor.
- **tau** (`float`) – target network synchronization coefficient.
- **n_critics** (`int`) – the number of Q functions for ensemble.
- **bootstrap** (`bool`) – flag to bootstrap Q functions.
- **share_encoder** (`bool`) – flag to share encoder network.
- **update_actor_interval** (`int`) – interval to update policy function.
- **initial_temperature** (`float`) – initial temperature value.
- **initial_alpha** (`float`) – initial α value.
- **alpha_threshold** (`float`) – threshold value described as ϵ .
- **lam** (`float`) – weight for critic ensemble.
- **n_action_samples** (`int`) – the number of action samples to estimate action-values.
- **mmd_sigma** (`float`) – σ for gaussian kernel in MMD calculation.
- **rl_start_epoch** (`int`) – epoch to start to update policy function and Q functions. If this is large, RL training would be more stabilized.
- **q_func_type** (`str`) – type of Q function. Available options are [`'mean'`, `'qr'`, `'iqn'`, `'fqf'`].
- **use_gpu** (`bool`, `int` or `d3rlpy.gpu.Device`) – flag to use GPU, device iD or device.
- **scaler** (`d3rlpy.preprocessingScaler` or `str`) – preprocessor. The available options are [`'pixel'`, `'min_max'`, `'standard'`].
- **augmentation** (`d3rlpy.augmentation.AugmentationPipeline` or `list(str)`) – augmentation pipeline.
- **n_augmentations** (`int`) – the number of data augmentations to update.

- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.bear_impl.BEARImpl*) – algorithm implementation.

actor_learning_rate

learning rate for policy function.

Type *float*

critic_learning_rate

learning rate for Q functions.

Type *float*

imitator_learning_rate

learning rate for behavior policy function.

Type *float*

temp_learning_rate

learning rate for temperature parameter.

Type *float*

alpha_learning_rate

learning rate for α .

Type *float*

actor_optim_factory

optimizer factory for the actor.

Type *d3rlpy.optimizers.OptimizerFactory*

critic_optim_factory

optimizer factory for the critic.

Type *d3rlpy.optimizers.OptimizerFactory*

imitator_optim_factory

optimizer factory for the behavior policy.

Type *d3rlpy.optimizers.OptimizerFactory*

temp_optim_factory

optimizer factory for the temperature.

Type *d3rlpy.optimizers.OptimizerFactory*

alpha_optim_factory

optimizer factory for α .

Type *d3rlpy.optimizers.OptimizerFactory*

actor_encoder_factory

encoder factory for the actor.

Type *d3rlpy.encoders.EncoderFactory*

critic_encoder_factory

encoder factory for the critic.

Type *d3rlpy.encoders.EncoderFactory*

imitator_encoder_factory
encoder factory for the behavior policy.
Type `d3rlpy.encoders.EncoderFactory`

batch_size
mini-batch size.
Type `int`

n_frames
the number of frames to stack for image observation.
Type `int`

gamma
discount factor.
Type `float`

tau
target network synchronization coefficient.
Type `float`

n_critics
the number of Q functions for ensemble.
Type `int`

bootstrap
flag to bootstrap Q functions.
Type `bool`

share_encoder
flag to share encoder network.
Type `bool`

update_actor_interval
interval to update policy function.
Type `int`

initial_temperature
initial temperature value.
Type `float`

initial_alpha
initial α value.
Type `float`

alpha_threshold
threshold value described as ϵ .
Type `float`

lam
weight for critic ensemble.
Type `float`

n_action_samples
the number of action samples to estimate action-values.

Type `int`

mmd_sigma
 σ for gaussian kernel in MMD calculation.

Type `float`

rl_start_epoch
 epoch to start to update policy function and Q functions. If this is large, RL training would be more stabilized.

Type `int`

q_func_type
 type of Q function..

Type `str`

use_gpu
 GPU device.

Type `d3rlpy.gpu.Device`

scaler
 preprocessor.

Type `d3rlpy.preprocessing.Scaler`

augmentation
 augmentation pipeline.

Type `d3rlpy.augmentation.AugmentationPipeline`

n_augmentations
 the number of data augmentations to update.

Type `int`

dynamics
 dynamics model.

Type `d3rlpy.dynamics.base.DynamicsBase`

impl
 algorithm implementation.

Type `d3rlpy.algos.torch.bear_impl.BEARImpl`

eval_results_
 evaluation results.

Type `dict`

Methods

create_impl (*observation_shape*, *action_size*)
 Instantiate implementation objects with the dataset shapes.
 This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs*=1000, *save_metrics*=True, *experiment_name*=None, *with_timestamp*=True, *logdir*='d3rlpy_logs', *verbose*=True, *show_progress*=True, *tensorboard*=True, *eval_episodes*=None, *save_interval*=1, *scorers*=None)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

fit_online (*env*, *buffer*, *explorer*=None, *n_steps*=1000000, *n_steps_per_epoch*=10000, *update_interval*=1, *update_start_step*=0, *eval_env*=None, *eval_epsilon*=0.0, *save_metrics*=True, *experiment_name*=None, *with_timestamp*=True, *logdir*='d3rlpy_logs', *verbose*=True, *show_progress*=True, *tensorboard*=True)

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (`d3rlpy.online.buffers.Buffer`) – replay buffer.
- **explorer** (`d3rlpy.online.explorers.Explorer`) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If False, the log directory is not created and the model parameters are not saved.

- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_online_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod from_json (*fname, use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool, int or d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters **deep** (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type *dict*

load_model (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters **fname** (*str*) – source file path.

predict (*x*)

Returns greedy actions.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters **x** (*numpy.ndarray*) – observations

Returns greedy actions

Return type *numpy.ndarray*

predict_value (*x, action, with_std=False*)

Returns predicted action-values.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

# for continuous control
# 100 actions with shape of (2,)
actions = np.random.random((100, 2))

# for discrete control
# 100 actions in integer values
actions = np.random.randint(2, size=100)

values = algo.predict_value(x, actions)
# values.shape == (100,)

values, stds = algo.predict_value(x, actions, with_std=True)
# stds.shape == (100,)
```

Parameters

- **x** (*numpy.ndarray*) – observations
- **action** (*numpy.ndarray*) – actions
- **with_std** (*bool*) – flag to return standard deviation of ensemble estimation. This deviation reflects uncertainty for the given observations. This uncertainty will be more accurate if you enable *bootstrap* flag and increase *n_critics* value.

Returns predicted action-values

Return type *numpy.ndarray*

sample_action (*x*)

Returns sampled actions.

The sampled actions are identical to the output of *predict* method if the policy is deterministic.

Parameters **x** (*numpy.ndarray*) – observations.

Returns sampled actions.

Return type *numpy.ndarray*

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.

save_policy (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).
- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type *d3rlpy.algos.base.AlgoBase*

update (*epoch*, *total_step*, *batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.

- **batch** (`d3rlpy.dataset.TransitionMiniBatch`) – mini-batch data.

Returns loss values.

Return type `list`

d3rlpy.algos.CQL

```
class d3rlpy.algos.CQL(*, actor_learning_rate=3e-05, critic_learning_rate=0.0003,
    temp_learning_rate=3e-05, alpha_learning_rate=0.0003, actor_optim_factory=<d3rlpy.optimizers.AdamFactory object>,
    critic_optim_factory=<d3rlpy.optimizers.AdamFactory object>, temp_optim_factory=<d3rlpy.optimizers.AdamFactory object>,
    alpha_optim_factory=<d3rlpy.optimizers.AdamFactory object>, actor_encoder_factory='default', critic_encoder_factory='default',
    batch_size=100, n_frames=1, gamma=0.99, tau=0.005, n_critics=2, bootstrap=False, share_encoder=False, update_actor_interval=1,
    initial_temperature=1.0, initial_alpha=5.0, alpha_threshold=10.0, n_action_samples=10, q_func_type='mean', use_gpu=False, scaler=None,
    augmentation=None, n_augmentations=1, dynamics=None, impl=None, **kwargs)
```

Conservative Q-Learning algorithm.

CQL is a SAC-based data-driven deep reinforcement learning algorithm, which achieves state-of-the-art performance in offline RL problems.

CQL mitigates overestimation error by minimizing action-values under the current policy and maximizing values under data distribution for underestimation issue.

$$L(\theta_i) = \alpha \mathbb{E}_{s_t \sim D} [\log \sum_a \exp Q_{\theta_i}(s_t, a) - \mathbb{E}_{a \sim D} [Q_{\theta_i}(s, a)] - \tau] + L_{SAC}(\theta_i)$$

where α is an automatically adjustable value via Lagrangian dual gradient descent and τ is a threshold value. If the action-value difference is smaller than τ , the α will become smaller. Otherwise, the α will become larger to aggressively penalize action-values.

In continuous control, $\log \sum_a \exp Q(s, a)$ is computed as follows.

$$\log \sum_a \exp Q(s, a) \approx \log \left(\frac{1}{2N} \sum_{a_i \sim \text{Unif}(a)} \left[\frac{\exp Q(s, a_i)}{\text{Unif}(a)} \right] + \frac{1}{2N} \sum_{a_i \sim \pi_\phi(a|s)} \left[\frac{\exp Q(s, a_i)}{\pi_\phi(a_i|s)} \right] \right)$$

where N is the number of sampled actions.

The rest of optimization is exactly same as `d3rlpy.algos.SAC`.

References

- Kumar et al., Conservative Q-Learning for Offline Reinforcement Learning.

Parameters

- **actor_learning_rate** (`float`) – learning rate for policy function.
- **critic_learning_rate** (`float`) – learning rate for Q functions.
- **temp_learning_rate** (`float`) – learning rate for temperature parameter of SAC.
- **alpha_learning_rate** (`float`) – learning rate for α .

- **actor_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **temp_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the temperature.
- **alpha_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for α .
- **actor_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or `str`) – encoder factory for the actor.
- **critic_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or `str`) – encoder factory for the critic.
- **batch_size** (`int`) – mini-batch size.
- **n_frames** (`int`) – the number of frames to stack for image observation.
- **gamma** (`float`) – discount factor.
- **tau** (`float`) – target network synchronization coefficient.
- **n_critics** (`int`) – the number of Q functions for ensemble.
- **bootstrap** (`bool`) – flag to bootstrap Q functions.
- **share_encoder** (`bool`) – flag to share encoder network.
- **update_actor_interval** (`int`) – interval to update policy function.
- **initial_temperature** (`float`) – initial temperature value.
- **initial_alpha** (`float`) – initial α value.
- **alpha_threshold** (`float`) – threshold value described as τ .
- **n_action_samples** (`int`) – the number of sampled actions to compute $\log \sum_a \exp Q(s, a)$.
- **q_func_type** (`str`) – type of Q function. Available options are [`'mean'`, `'qr'`, `'iqn'`, `'fqf'`].
- **use_gpu** (`bool`, `int` or `d3rlpy.gpu.Device`) – flag to use GPU, device ID or device.
- **scaler** (`d3rlpy.preprocessingScaler` or `str`) – preprocessor. The available options are [`'pixel'`, `'min_max'`, `'standard'`].
- **augmentation** (`d3rlpy.augmentation.AugmentationPipeline` or `list(str)`) – augmentation pipeline.
- **n_augmentations** (`int`) – the number of data augmentations to update.
- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model for data augmentation.
- **impl** (`d3rlpy.algos.torch.cql_impl.CQLImpl`) – algorithm implementation.

actor_learning_rate

learning rate for policy function.

Type `float`

critic_learning_rate

learning rate for Q functions.

Type `float`

temp_learning_rate

learning rate for temperature parameter of SAC.

Type `float`

alpha_learning_rate

learning rate for α .

Type `float`

actor_optim_factory

optimizer factory for the actor.

Type `d3rlpy.optimizers.OptimizerFactory`

critic_optim_factory

optimizer factory for the critic.

Type `d3rlpy.optimizers.OptimizerFactory`

temp_optim_factory

optimizer factory for the temperature.

Type `d3rlpy.optimizers.OptimizerFactory`

alpha_optim_factory

optimizer factory for α .

Type `d3rlpy.optimizers.OptimizerFactory`

actor_encoder_factory

encoder factory for the actor.

Type `d3rlpy.encoders.EncoderFactory`

critic_encoder_factory

encoder factory for the critic.

Type `d3rlpy.encoders.EncoderFactory`

batch_size

mini-batch size.

Type `int`

n_frames

the number of frames to stack for image observation.

Type `int`

gamma

discount factor.

Type `float`

tau

target network synchronization coefficient.

Type `float`

n_critics

the number of Q functions for ensemble.

Type `int`

bootstrap
flag to bootstrap Q functions.

Type `bool`

share_encoder
flag to share encoder network.

Type `bool`

update_actor_interval
interval to update policy function.

Type `int`

initial_temperature
initial temperature value.

Type `float`

initial_alpha
initial α value.

Type `float`

alpha_threshold
threshold value described as τ .

Type `float`

n_action_samples
the number of sampled actions to compute $\log \sum_a \exp Q(s, a)$.

Type `int`

q_func_type
type of Q function.

Type `str`

use_gpu
GPU device.

Type `d3rlpy.gpu.Device`

scaler
preprocessor.

Type `d3rlpy.preprocessing.Scaler`

augmentation
augmentation pipeline.

Type `d3rlpy.augmentation.AugmentationPipeline`

n_augmentations
the number of data augmentations to update.

Type `int`

dynamics
dynamics model.

Type `d3rlpy.dynamics.base.DynamicsBase`

impl
algorithm implementation.
Type `d3rlpy.algos.torch.cql_impl.CQLImpl`

eval_results_
evaluation results.
Type `dict`

Methods

create_impl (*observation_shape*, *action_size*)
Instantiate implementation objects with the dataset shapes.
This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs=1000*, *save_metrics=True*, *experiment_name=None*, *with_timestamp=True*, *logdir='d3rlpy_logs'*, *verbose=True*, *show_progress=True*, *tensorboard=True*, *eval_episodes=None*, *save_interval=1*, *scorers=None*)
Trains with the given dataset.

`algo.fit(episodes)`

Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.


```
fit_online (env, buffer, explorer=None, n_steps=1000000, n_steps_per_epoch=10000,
            update_interval=1, update_start_step=0, eval_env=None, eval_epsilon=0.0,
            save_metrics=True, experiment_name=None, with_timestamp=True,
            logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard=True)
```

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (*d3rlpy.online.buffers.Buffer*) – replay buffer.
- **explorer** (*d3rlpy.online.explorers.Explorer*) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If False, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_online_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

```
classmethod from_json (fname, use_gpu=False)
```

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.

- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters **deep** (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type *dict*

load_model (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters **fname** (*str*) – source file path.

predict (*x*)

Returns greedy actions.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters **x** (*numpy.ndarray*) – observations

Returns greedy actions

Return type *numpy.ndarray*

predict_value (*x, action, with_std=False*)

Returns predicted action-values.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

# for continuous control
# 100 actions with shape of (2,)
actions = np.random.random((100, 2))
```

(continues on next page)

(continued from previous page)

```
# for discrete control
# 100 actions in integer values
actions = np.random.randint(2, size=100)

values = algo.predict_value(x, actions)
# values.shape == (100,)

values, stds = algo.predict_value(x, actions, with_std=True)
# stds.shape == (100,)
```

Parameters

- **x** (*numpy.ndarray*) – observations
- **action** (*numpy.ndarray*) – actions
- **with_std** (*bool*) – flag to return standard deviation of ensemble estimation. This deviation reflects uncertainty for the given observations. This uncertainty will be more accurate if you enable *bootstrap* flag and increase *n_critics* value.

Returns predicted action-values**Return type** *numpy.ndarray***sample_action** (*x*)

Returns sampled actions.

The sampled actions are identical to the output of *predict* method if the policy is deterministic.**Parameters** **x** (*numpy.ndarray*) – observations.**Returns** sampled actions.**Return type** *numpy.ndarray***save_model** (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.**save_policy** (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).

- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type d3rlpy.algos.base.AlgoBase

update (*epoch, total_step, batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (d3rlpy.dataset.TransitionMiniBatch) – mini-batch data.

Returns loss values.

Return type list

d3rlpy.algos.AWR

```
class d3rlpy.algos.AWR(*, actor_learning_rate=5e-05, critic_learning_rate=0.0001, actor_optim_factory=<d3rlpy.optimizers.SGDFactory object>, critic_optim_factory=<d3rlpy.optimizers.SGDFactory object>, actor_encoder_factory='default', critic_encoder_factory='default', batch_size=2048, n_frames=1, gamma=0.99, batch_size_per_update=256, n_actor_updates=1000, n_critic_updates=200, lam=0.95, beta=1.0, max_weight=20.0, use_gpu=False, scaler=None, augmentation=None, n_augmentations=1, dynamics=None, impl=None, **kwargs)
```

Advantage-Weighted Regression algorithm.

AWR is an actor-critic algorithm that trains via supervised regression way, and has shown strong performance in online and offline settings.

The value function is trained as a supervised regression problem.

$$L(\theta) = \mathbb{E}_{s_t, R_t \sim D} [(R_t - V(s_t | \theta))^2]$$

where R_t is approximated using TD(λ) to mitigate high variance issue.

The policy function is also trained as a supervised regression problem.

$$J(\phi) = \mathbb{E}_{s_t, a_t, R_t \sim D} [\log \pi(a_t | s_t, \phi) \exp(\frac{1}{B} (R_t - V(s_t | \theta)))]$$

where B is a constant factor.

References

- Peng et al., Advantage-Weighted Regression: Simple and Scalable Off-Policy Reinforcement Learning

Parameters

- **actor_learning_rate** (*float*) – learning rate for policy function.
- **critic_learning_rate** (*float*) – learning rate for value function.
- **actor_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **actor_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the actor.
- **critic_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the critic.
- **batch_size** (*int*) – batch size per iteration.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **gamma** (*float*) – discount factor.
- **batch_size_per_update** (*int*) – mini-batch size.
- **n_actor_updates** (*int*) – actor gradient steps per iteration.
- **n_critic_updates** (*int*) – critic gradient steps per iteration.
- **lam** (*float*) – λ for TD(λ).
- **beta** (*float*) – B for weight scale.
- **max_weight** (*float*) – w_{\max} for weight clipping.
- **use_gpu** (*bool*, *int* or `d3rlpy.gpu.Device`) – flag to use GPU, device ID or device.
- **scaler** (`d3rlpy.preprocessingScaler` or *str*) – preprocessor. The available options are ['pixel', 'min_max', 'standard']
- **augmentation** (`d3rlpy.augmentation.AugmentationPipeline` or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.
- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model for data augmentation.
- **impl** (`d3rlpy.algos.torch.awr_impl.AWRImpl`) – algorithm implementation.

actor_learning_rate

learning rate for policy function.

Type *float*

critic_learning_rate

learning rate for value function.

Type float

actor_optim_factory

optimizer factory for the actor.

Type *d3rlpy.optimizers.OptimizerFactory*

critic_optim_factory

optimizer factory for the critic.

Type *d3rlpy.optimizers.OptimizerFactory*

actor_encoder_factory

encoder factory for the actor.

Type *d3rlpy.encoders.EncoderFactory*

critic_encoder_factory

encoder factory for the critic.

Type *d3rlpy.encoders.EncoderFactory*

batch_size

batch size per iteration.

Type int

n_frames

the number of frames to stack for image observation.

Type int

gamma

discount factor.

Type float

batch_size_per_update

mini-batch size.

Type int

n_actor_updates

actor gradient steps per iteration.

Type int

n_critic_updates

critic gradient steps per iteration.

Type int

lam

λ for TD(λ).

Type float

beta

B for weight scale.

Type float

max_weight

w_{\max} for weight clipping.

Type float

use_gpu
GPU device.
Type d3rlpy.gpu.Device

scaler
preprocessor.
Type d3rlpy.preprocessingScaler

augmentation
augmentation pipeline.
Type d3rlpy.augmentation.AugmentationPipeline

n_augmentations
the number of data augmentations to update.
Type int

dynamics
dynamics model.
Type d3rlpy.dynamics.base.DynamicsBase

impl
algorithm implementation.
Type d3rlpy.algos.torch.awr_impl.AWRImpl

eval_results_
evaluation results.
Type dict

Methods

create_impl (*observation_shape, action_size*)
Instantiate implementation objects with the dataset shapes.
This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes, n_epochs=1000, save_metrics=True, experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard=True, eval_episodes=None, save_interval=1, scorers=None*)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (d3rlpy.dataset.Episode)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.

- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

```
fit_online (env, buffer, explorer=None, n_steps=1000000, n_steps_per_epoch=10000,  
            update_interval=1, update_start_step=0, eval_env=None, eval_epsilon=0.0,  
            save_metrics=True, experiment_name=None, with_timestamp=True,  
            logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard=True)
```

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (*d3rlpy.online.buffers.Buffer*) – replay buffer.
- **explorer** (*d3rlpy.online.explorers.Explorer*) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If False, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_online_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod `from_json` (*fname*, *use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters **deep** (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type *dict*

load_model (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters **fname** (*str*) – source file path.

predict (*x*)

Returns greedy actions.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
```

(continues on next page)

(continued from previous page)

```
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters **x** (*numpy.ndarray*) – observations

Returns greedy actions

Return type *numpy.ndarray*

predict_value (*x*, **args*, ***kwargs*)

Returns predicted state values.

Parameters **x** (*numpy.ndarray*) – observations.

Returns predicted state values.

Return type *numpy.ndarray*

sample_action (*x*)

Returns sampled actions.

The sampled actions are identical to the output of *predict* method if the policy is deterministic.

Parameters **x** (*numpy.ndarray*) – observations.

Returns sampled actions.

Return type *numpy.ndarray*

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.

save_policy (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).
- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (**params)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type d3rlpy.algos.base.AlgoBase

update (epoch, itr, batch)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (d3rlpy.dataset.TransitionMiniBatch) – mini-batch data.

Returns loss values.

Return type list

d3rlpy.algos.AWAC

```
class d3rlpy.algos.AWAC(*, actor_learning_rate=0.0003, critic_learning_rate=0.0003,
                        actor_optim_factory=<d3rlpy.optimizers.AdamFactory object>,
                        critic_optim_factory=<d3rlpy.optimizers.AdamFactory object>, ac-
                        tor_encoder_factory='default', critic_encoder_factory='default',
                        batch_size=1024, n_frames=1, gamma=0.99, tau=0.005, lam=1.0,
                        n_action_samples=1, max_weight=20.0, n_critics=2, bootstrap=False,
                        share_encoder=False, update_actor_interval=1, q_func_type='mean',
                        use_gpu=False, scaler=None, augmentation=None, n_augmentations=1,
                        dynamics=None, impl=None, **kwargs)
```

Advantage Weighted Actor-Critic algorithm.

AWAC is a TD3-based actor-critic algorithm that enables efficient fine-tuning where the policy is trained with offline datasets and is deployed to online training.

The policy is trained as a supervised regression.

$$J(\phi) = \mathbb{E}_{s_t, a_t \sim D} [\log \pi_\phi(a_t | s_t) \exp(\frac{1}{\lambda} A^\pi(s_t, a_t))]$$

where $A^\pi(s_t, a_t) = Q_\theta(s_t, a_t) - Q_\theta(s_t, a'_t)$ and $a'_t \sim \pi_\phi(\cdot | s_t)$

The key difference from AWR is that AWAC uses Q-function trained via TD learning for the better sample-efficiency.

References

- Nair et al., Accelerating Online Reinforcement Learning with Offline Datasets.

Parameters

- **actor_learning_rate** (*float*) – learning rate for policy function.
- **critic_learning_rate** (*float*) – learning rate for Q functions.
- **actor_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **actor_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the actor.
- **critic_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the critic.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficient.
- **lam** (*float*) – λ for weight calculation.
- **n_action_samples** (*int*) – the number of sampled actions to calculate $A^\pi(s_t, a_t)$.
- **max_weight** (*float*) – maximum weight for cross-entropy loss.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share_encoder** (*bool*) – flag to share encoder network.
- **update_actor_interval** (*int*) – interval to update policy function.
- **q_func_type** (*str*) – type of Q function. Available options are ['mean', 'qr', 'iqn', 'fqf'].
- **use_gpu** (*bool*, *int* or `d3rlpy.gpu.Device`) – flag to use GPU, device ID or device.
- **scaler** (`d3rlpy.preprocessingScaler` or *str*) – preprocessor. The available options are ['pixel', 'min_max', 'standard']
- **augmentation** (`d3rlpy.augmentation.AugmentationPipeline` or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.
- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model for data augmentation.
- **impl** (`d3rlpy.algos.torch.sac_impl.SACImpl`) – algorithm implementation.

actor_learning_rate

learning rate for policy function.

Type *float*

critic_learning_rate

learning rate for Q functions.

Type float

actor_optim_factory

optimizer factory for the actor.

Type *d3rlpy.optimizers.OptimizerFactory*

critic_optim_factory

optimizer factory for the critic.

Type *d3rlpy.optimizers.OptimizerFactory*

actor_encoder_factory

encoder factory for the actor.

Type *d3rlpy.encoders.EncoderFactory*

critic_encoder_factory

encoder factory for the critic.

Type *d3rlpy.encoders.EncoderFactory*

batch_size

mini-batch size.

Type int

n_frames

the number of frames to stack for image observation.

Type int

gamma

discount factor.

Type float

tau

target network synchronization coefficient.

Type float

lam

λ for weight calculation.

Type float

n_action_samples

the number of sampled actions to calculate $A^\pi(s_t, a_t)$.

Type int

max_weight

maximum weight for cross-entropy loss.

Type float

n_critics

the number of Q functions for ensemble.

Type int

bootstrap

flag to bootstrap Q functions.

Type bool

share_encoder

flag to share encoder network.

Type `bool`

update_actor_interval

interval to update policy function.

Type `int`

q_func_type

type of Q function.

Type `str`

use_gpu

flag to use GPU, device ID or device.

Type `bool`, `int` or `d3rlpy.gpu.Device`

scaler

preprocessor.

Type `d3rlpy.preprocessingScaler` or `str`

augmentation

augmentation pipeline.

Type `d3rlpy.augmentation.AugmentationPipeline` or `list(str)`

n_augmentations

the number of data augmentations to update.

Type `int`

dynamics

dynamics model for data augmentation.

Type `d3rlpy.dynamics.base.DynamicsBase`

impl

algorithm implementation.

Type `d3rlpy.algos.torch.sac_impl.SACImpl`

Methods

create_impl (*observation_shape*, *action_size*)

Instantiate implementation objects with the dataset shapes.

This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs=1000*, *save_metrics=True*, *experiment_name=None*, *with_timestamp=True*, *logdir='d3rlpy_logs'*, *verbose=True*, *show_progress=True*, *tensorboard=True*, *eval_episodes=None*, *save_interval=1*, *scorers=None*)
Trains with the given dataset.

`algo.fit(episodes)`

Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

```
fit_online (env, buffer, explorer=None, n_steps=1000000, n_steps_per_epoch=10000,
            update_interval=1, update_start_step=0, eval_env=None, eval_epsilon=0.0,
            save_metrics=True, experiment_name=None, with_timestamp=True,
            logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard=True)
```

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (`d3rlpy.online.buffers.Buffer`) – replay buffer.
- **explorer** (`d3rlpy.online.explorers.Explorer`) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If False, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_online_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.

- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod **from_json** (*fname, use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool, int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters **deep** (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type *dict*

load_model (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters **fname** (*str*) – source file path.

predict (*x*)

Returns greedy actions.


```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters **x** (*numpy.ndarray*) – observations

Returns greedy actions

Return type *numpy.ndarray*

predict_value (*x, action, with_std=False*)

Returns predicted action-values.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

# for continuous control
# 100 actions with shape of (2,)
actions = np.random.random((100, 2))

# for discrete control
# 100 actions in integer values
actions = np.random.randint(2, size=100)

values = algo.predict_value(x, actions)
# values.shape == (100,)

values, stds = algo.predict_value(x, actions, with_std=True)
# stds.shape == (100,)
```

Parameters

- **x** (*numpy.ndarray*) – observations
- **action** (*numpy.ndarray*) – actions
- **with_std** (*bool*) – flag to return standard deviation of ensemble estimation. This deviation reflects uncertainty for the given observations. This uncertainty will be more accurate if you enable *bootstrap* flag and increase *n_critics* value.

Returns predicted action-values

Return type *numpy.ndarray*

sample_action (*x*)

Returns sampled actions.

The sampled actions are identical to the output of *predict* method if the policy is deterministic.

Parameters **x** (*numpy.ndarray*) – observations.

Returns sampled actions.

Return type *numpy.ndarray*

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.

save_policy (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).
- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type d3rlpy.algos.base.AlgoBase

update (*epoch*, *total_step*, *batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (d3rlpy.dataset.TransitionMiniBatch) – mini-batch data.

Returns loss values.

Return type list

3.1.2 Discrete control algorithms

<code>d3rlpy.algos.DiscreteBC</code>	Behavior Cloning algorithm for discrete control.
<code>d3rlpy.algos.DQN</code>	Deep Q-Network algorithm.
<code>d3rlpy.algos.DoubleDQN</code>	Double Deep Q-Network algorithm.
<code>d3rlpy.algos.DiscreteSAC</code>	Soft Actor-Critic algorithm for discrete action-space.
<code>d3rlpy.algos.DiscreteBCQ</code>	Discrete version of Batch-Constrained Q-learning algorithm.
<code>d3rlpy.algos.DiscreteCQL</code>	Discrete version of Conservative Q-Learning algorithm.
<code>d3rlpy.algos.DiscreteAWR</code>	Discrete version of Advantage-Weighted Regression algorithm.

d3rlpy.algos.DiscreteBC

```
class d3rlpy.algos.DiscreteBC (*, learning_rate=0.001, optim_factory=<d3rlpy.optimizers.AdamFactory
                                object>, encoder_factory='default', batch_size=100,
                                n_frames=1, beta=0.5, use_gpu=False, scaler=None, augmen-
                                tation=None, n_augmentations=1, dynamics=None, impl=None,
                                **kwargs)
```

Behavior Cloning algorithm for discrete control.

Behavior Cloning (BC) is to imitate actions in the dataset via a supervised learning approach. Since BC is only imitating action distributions, the performance will be close to the mean of the dataset even though BC mostly works better than online RL algorithms.

$$L(\theta) = \mathbb{E}_{a_t, s_t \sim D} \left[- \sum_a p(a|s_t) \log \pi_\theta(a|s_t) \right]$$

where $p(a|s_t)$ is implemented as a one-hot vector.

Parameters

- **learning_rate** (*float*) – learning rate.
- **optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory.
- **encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **beta** (*float*) – regularization factor.
- **use_gpu** (*bool*, *int* or `d3rlpy.gpu.Device`) – flag to use GPU, device ID or device.
- **scaler** (`d3rlpy.preprocessingScaler` or *str*) – preprocessor. The available options are [*'pixel'*, *'min_max'*, *'standard'*]
- **augmentation** (`d3rlpy.augmentation.AugmentationPipeline` or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.
- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model for data augmentation.

- **impl** (*d3rlpy.algos.torch.bc_impl.DiscreteBCImpl*) – implemenation of the algorithm.

learning_rate

learing rate.

Type *float*

optim_factory

optimizer factory.

Type *d3rlpy.optimizers.OptimizerFactory*

encoder_factory

encoder factory.

Type *d3rlpy.encoders.EncoderFactory*

batch_size

mini-batch size.

Type *int*

n_frames

the number of frames to stack for image observation.

Type *int*

beta

reguralization factor.

Type *float*

use_gpu

GPU device.

Type *d3rlpy.gpu.Device*

scaler

preprocessor.

Type *d3rlpy.preprocessingScaler*

augmentation

augmentation pipeline.

Type *d3rlpy.augmentation.AugmentationPipeline*

n_augmentations

the number of data augmentations to update.

Type *int*

dynamics

dynamics model.

Type *d3rlpy.dynamics.base.DynamicsBase*

impl

implemenation of the algorithm.

Type *d3rlpy.algos.torch.bc_impl.DiscreteBCImpl*

eval_results_

evaluation results.

Type *dict*

Methods

create_impl (*observation_shape*, *action_size*)

Instantiate implementation objects with the dataset shapes.

This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs*=1000, *save_metrics*=True, *experiment_name*=None, *with_timestamp*=True, *logdir*='d3rlpy_logs', *verbose*=True, *show_progress*=True, *tensorboard*=True, *eval_episodes*=None, *save_interval*=1, *scorers*=None)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

fit_online (*env*, *buffer*, *explorer*=None, *n_steps*=1000000, *n_steps_per_epoch*=10000, *update_interval*=1, *update_start_step*=0, *eval_env*=None, *eval_epsilon*=0.0, *save_metrics*=True, *experiment_name*=None, *with_timestamp*=True, *logdir*='d3rlpy_logs', *verbose*=True, *show_progress*=True, *tensorboard*=True)

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (*d3rlpy.online.buffers.Buffer*) – replay buffer.
- **explorer** (*d3rlpy.online.explorers.Explorer*) – action explorer.

- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If False, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_online_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod from_json (*fname*, *use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters `deep` (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type `dict`

load_model (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters `fname` (*str*) – source file path.

predict (*x*)

Returns greedy actions.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters `x` (*numpy.ndarray*) – observations

Returns greedy actions

Return type `numpy.ndarray`

predict_value (*x, action*)

value prediction is not supported by BC algorithms.

sample_action (*x*)

sampling action is not supported by BC algorithm.

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters `fname` (*str*) – destination file path.

save_policy (*fname, as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).
- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type d3rlpy.algos.base.AlgoBase

update (*epoch, itr, batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (d3rlpy.dataset.TransitionMiniBatch) – mini-batch data.

Returns loss values.

Return type list

d3rlpy.algos.DQN

```
class d3rlpy.algos.DQN(*, learning_rate=6.25e-05, optim_factory=<d3rlpy.optimizers.AdamFactory
object>, encoder_factory='default', batch_size=32, n_frames=1,
gamma=0.99, n_critics=1, bootstrap=False, share_encoder=False,
target_update_interval=8000.0, q_func_type='mean', use_gpu=False,
scaler=None, augmentation=None, n_augmentations=1, dynamics=None,
impl=None, **kwargs)
```

Deep Q-Network algorithm.

$$L(\theta) = \mathbb{E}_{s_t, a_t, r_{t+1}, s_{t+1} \sim D} [(r_{t+1} + \gamma \max_a Q_{\theta'}(s_{t+1}, a) - Q_{\theta}(s_t, a_t))^2]$$

where θ' is the target network parameter. The target network parameter is synchronized every *target_update_interval* iterations.

References

- Mnih et al., Human-level control through deep reinforcement learning.

Parameters

- **learning_rate** (*float*) – learning rate.
- **optim_factory** (`d3rlpy.optimizers.OptimizerFactory` or *str*) – optimizer factory.
- **encoder_factory** (`d3rlpy.encoders.EncoderFactory`) – encoder factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **gamma** (*float*) – discount factor.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share_encoder** (*bool*) – flag to share encoder network.
- **target_update_interval** (*int*) – interval to update the target network.
- **q_func_type** (*str*) – type of Q function. Available options are [*'mean'*, *'qr'*, *'iqn'*, *'fqf'*].
- **use_gpu** (*bool*, *int* or `d3rlpy.gpu.Device`) – flag to use GPU, device ID or device.
- **scaler** (`d3rlpy.preprocessingScaler` or *str*) – preprocessor. The available options are [*'pixel'*, *'min_max'*, *'standard'*].
- **augmentation** (`d3rlpy.augmentation.AugmentationPipeline` or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.
- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model for data augmentation.
- **impl** (`d3rlpy.algos.torch.dqn_impl.DQNImpl`) – algorithm implementation.

learning_rate

learning rate.

Type *float*

optim_factory

optimizer factory.

Type `d3rlpy.optimizers.OptimizerFactory`

encoder_factory

encoder factory.

Type `d3rlpy.encoders.EncoderFactory`

batch_size

mini-batch size.

Type *int*

n_frames
the number of frames to stack for image observation.
Type `int`

gamma
discount factor.
Type `float`

n_critics
the number of Q functions for ensemble.
Type `int`

bootstrap
flag to bootstrap Q functions.
Type `bool`

share_encoder
flag to share encoder network.
Type `bool`

target_update_interval
interval to update the target network.
Type `int`

q_func_type
type of Q function.
Type `str`

use_gpu
GPU device.
Type `d3rlpy.gpu.Device`

scaler
preprocessor.
Type `d3rlpy.preprocessing.Scaler`

augmentation
augmentation pipeline.
Type `d3rlpy.augmentation.AugmentationPipeline`

n_augmentations
the number of data augmentations to update.
Type `int`

dynamics
dynamics model.
Type `d3rlpy.dynamics.base.DynamicsBase`

impl
algorithm implementation.
Type `d3rlpy.algos.torch.dqn_impl.DQNImpl`

eval_results_
evaluation results.

Type `dict`

Methods

create_impl (*observation_shape*, *action_size*)

Instantiate implementation objects with the dataset shapes.

This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs=1000*, *save_metrics=True*, *experiment_name=None*, *with_timestamp=True*, *logdir='d3rlpy_logs'*, *verbose=True*, *show_progress=True*, *tensorboard=True*, *eval_episodes=None*, *save_interval=1*, *scorers=None*)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

fit_online (*env*, *buffer*, *explorer=None*, *n_steps=1000000*, *n_steps_per_epoch=10000*, *update_interval=1*, *update_start_step=0*, *eval_env=None*, *eval_epsilon=0.0*, *save_metrics=True*, *experiment_name=None*, *with_timestamp=True*, *logdir='d3rlpy_logs'*, *verbose=True*, *show_progress=True*, *tensorboard=True*)

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*`gym.Env`*) – gym-like environment.
- **buffer** (*`d3rlpy.online.buffers.Buffer`*) – replay buffer.

- **explorer** (*d3rlpy.online.explorers.Explorer*) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If False, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_online_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod from_json (*fname, use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool, int or d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```

params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)

```

Parameters `deep` (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type `dict`

load_model (*fname*)

Load neural network parameters.

```

algo.load_model('model.pt')

```

Parameters `fname` (*str*) – source file path.

predict (*x*)

Returns greedy actions.

```

# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control

```

Parameters `x` (*numpy.ndarray*) – observations

Returns greedy actions

Return type `numpy.ndarray`

predict_value (*x, action, with_std=False*)

Returns predicted action-values.

```

# 100 observations with shape of (10,)
x = np.random.random((100, 10))

# for continuous control
# 100 actions with shape of (2,)
actions = np.random.random((100, 2))

# for discrete control
# 100 actions in integer values
actions = np.random.randint(2, size=100)

values = algo.predict_value(x, actions)
# values.shape == (100,)

values, stds = algo.predict_value(x, actions, with_std=True)
# stds.shape == (100,)

```

Parameters

- `x` (*numpy.ndarray*) – observations

- **action** (*numpy.ndarray*) – actions
- **with_std** (*bool*) – flag to return standard deviation of ensemble estimation. This deviation reflects uncertainty for the given observations. This uncertainty will be more accurate if you enable *bootstrap* flag and increase *n_critics* value.

Returns predicted action-values

Return type *numpy.ndarray*

sample_action (*x*)

Returns sampled actions.

The sampled actions are identical to the output of *predict* method if the policy is deterministic.

Parameters **x** (*numpy.ndarray*) – observations.

Returns sampled actions.

Return type *numpy.ndarray*

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.

save_policy (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).
- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type `d3rlpy.algos.base.AlgoBase`

update (*epoch*, *total_step*, *batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (`d3rlpy.dataset.TransitionMiniBatch`) – mini-batch data.

Returns loss values.

Return type `list`

d3rlpy.algos.DoubleDQN

```
class d3rlpy.algos.DoubleDQN(*,
                             learning_rate=6.25e-05,
                             optim_factory=<d3rlpy.optimizers.AdamFactory object>,
                             encoder_factory='default',
                             batch_size=32,
                             n_frames=1,
                             gamma=0.99,
                             n_critics=1,
                             bootstrap=False,
                             share_encoder=False,
                             target_update_interval=8000.0,
                             q_func_type='mean',
                             use_gpu=False,
                             scaler=None,
                             augmentation=None,
                             n_augmentations=1,
                             dynamics=None,
                             impl=None,
                             **kwargs)
```

Double Deep Q-Network algorithm.

The difference from DQN is that the action is taken from the current Q function instead of the target Q function. This modification significantly decreases overestimation bias of TD learning.

$$L(\theta) = \mathbb{E}_{s_t, a_t, r_{t+1}, s_{t+1} \sim D} [(r_{t+1} + \gamma Q_{\theta'}(s_{t+1}, \operatorname{argmax}_a Q_{\theta}(s_{t+1}, a)) - Q_{\theta}(s_t, a_t))^2]$$

where θ' is the target network parameter. The target network parameter is synchronized every *target_update_interval* iterations.

References

- Hasselt et al., Deep reinforcement learning with double Q-learning.

Parameters

- **learning_rate** (*float*) – learning rate.
- **optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory.
- **encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **gamma** (*float*) – discount factor.
- **n_critics** (*int*) – the number of Q functions.

- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share_encoder** (*bool*) – flag to share encoder network.
- **target_update_interval** (*int*) – interval to synchronize the target network.
- **q_func_type** (*str*) – type of Q function. Available options are [*'mean'*, *'qr'*, *'iqn'*, *'fqf'*].
- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are [*'pixel'*, *'min_max'*, *'standard'*]
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.dqn_impl.DoubleDQNImpl*) – algorithm implementation.

learning_rate

learning rate.

Type *float*

optim_factory

optimizer factory.

Type *d3rlpy.optimizers.OptimizerFactory*

encoder_factory

encoder factory.

Type *d3rlpy.encoders.EncoderFactory*

batch_size

mini-batch size.

Type *int*

n_frames

the number of frames to stack for image observation.

Type *int*

gamma

discount factor.

Type *float*

n_critics

the number of Q functions.

Type *int*

bootstrap

flag to bootstrap Q functions.

Type *bool*

share_encoder
flag to share encoder network.
Type `bool`

target_update_interval
interval to synchronize the target network.
Type `int`

q_func_type
type of Q function.
Type `str`

use_gpu
GPU device.
Type `d3rlpy.gpu.Device`

scaler
preprocessor.
Type `d3rlpy.preprocessing.Scaler`

augmentation
augmentation pipeline.
Type `d3rlpy.augmentation.AugmentationPipeline` or `list(str)`

n_augmentations
the number of data augmentations to update.
Type `int`

dynamics
dynamics model.
Type `d3rlpy.dynaics.base.DynamicsBase`

impl
algorithm implementation.
Type `d3rlpy.algos.torch.dqn_impl.DoubleDQNImpl`

Methods

create_impl (*observation_shape*, *action_size*)
Instantiate implementation objects with the dataset shapes.
This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs=1000*, *save_metrics=True*, *experiment_name=None*, *with_timestamp=True*, *logdir='d3rlpy_logs'*, *verbose=True*, *show_progress=True*, *tensorboard=True*, *eval_episodes=None*, *save_interval=1*, *scorers=None*)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

```
fit_online (env, buffer, explorer=None, n_steps=1000000, n_steps_per_epoch=10000,  
            update_interval=1, update_start_step=0, eval_env=None, eval_epsilon=0.0,  
            save_metrics=True, experiment_name=None, with_timestamp=True,  
            logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard=True)
```

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (`d3rlpy.online.buffers.Buffer`) – replay buffer.
- **explorer** (`d3rlpy.online.explorers.Explorer`) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If False, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_online_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.

- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod from_json (*fname*, *use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters **deep** (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type *dict*

load_model (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters **fname** (*str*) – source file path.

predict (*x*)

Returns greedy actions.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters **x** (*numpy.ndarray*) – observations

Returns greedy actions

Return type *numpy.ndarray*

predict_value (*x*, *action*, *with_std=False*)

Returns predicted action-values.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

# for continuous control
# 100 actions with shape of (2,)
actions = np.random.random((100, 2))

# for discrete control
# 100 actions in integer values
actions = np.random.randint(2, size=100)

values = algo.predict_value(x, actions)
# values.shape == (100,)

values, stds = algo.predict_value(x, actions, with_std=True)
# stds.shape == (100,)
```

Parameters

- **x** (*numpy.ndarray*) – observations
- **action** (*numpy.ndarray*) – actions
- **with_std** (*bool*) – flag to return standard deviation of ensemble estimation. This deviation reflects uncertainty for the given observations. This uncertainty will be more accurate if you enable *bootstrap* flag and increase *n_critics* value.

Returns predicted action-values

Return type *numpy.ndarray*

sample_action (*x*)

Returns sampled actions.

The sampled actions are identical to the output of *predict* method if the policy is deterministic.

Parameters **x** (*numpy.ndarray*) – observations.

Returns sampled actions.

Return type *numpy.ndarray*

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.

save_policy (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).
- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type d3rlpy.algos.base.AlgoBase

update (*epoch*, *total_step*, *batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (d3rlpy.dataset.TransitionMiniBatch) – mini-batch data.

Returns loss values.

Return type list

d3rlpy.algos.DiscreteSAC

```
class d3rlpy.algos.DiscreteSAC(*, actor_learning_rate=0.0003, critic_learning_rate=0.0003,
                               temp_learning_rate=0.0003,
                               actor_optim_factory=<d3rlpy.optimizers.AdamFactory object>,
                               critic_optim_factory=<d3rlpy.optimizers.AdamFactory object>,
                               temp_optim_factory=<d3rlpy.optimizers.AdamFactory object>,
                               actor_encoder_factory='default',
                               critic_encoder_factory='default',
                               batch_size=64,
                               n_frames=1, gamma=0.99, n_critics=2, bootstrap=False,
                               share_encoder=False, initial_temperature=1.0,
                               target_update_interval=8000, q_func_type='mean',
                               use_gpu=False, scaler=None, augmentation=None,
                               n_augmentations=1, dynamics=None, impl=None, **kwargs)
```

Soft Actor-Critic algorithm for discrete action-space.

This discrete version of SAC is built based on continuous version of SAC with additional modifications.

The target state-value is calculated as expectation of all action-values.

$$V(s_t) = \pi_\phi(s_t)^T [Q_\theta(s_t) - \alpha \log(\pi_\phi(s_t))]$$

Similarly, the objective function for the temperature parameter is as follows.

$$J(\alpha) = \pi_\phi(s_t)^T [-\alpha(\log(\pi_\phi(s_t)) + H)]$$

Finally, the objective function for the policy function is as follows.

$$J(\phi) = \mathbb{E}_{s_t \sim D} [\pi_\phi(s_t)^T [\alpha \log(\pi_\phi(s_t)) - Q_\theta(s_t)]]$$

References

- Christodoulou, Soft Actor-Critic for Discrete Action Settings.

Parameters

- **actor_learning_rate** (*float*) – learning rate for policy function.
- **critic_learning_rate** (*float*) – learning rate for Q functions.
- **temp_learning_rate** (*float*) – learning rate for temperature parameter.
- **actor_optim_factory** (*d3rlpy.optimizers.OptimizerFactory*) – optimizer factory for the actor.
- **critic_optim_factory** (*d3rlpy.optimizers.OptimizerFactory*) – optimizer factory for the critic.
- **temp_optim_factory** (*d3rlpy.optimizers.OptimizerFactory*) – optimizer factory for the temperature.
- **actor_encoder_factory** (*d3rlpy.encoders.EncoderFactory* or *str*) – encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.encoders.EncoderFactory* or *str*) – encoder factory for the critic.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.

- **gamma** (*float*) – discount factor.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share_encoder** (*bool*) – flag to share encoder network.
- **initial_temperature** (*float*) – initial temperature value.
- **q_func_type** (*str*) – type of Q function. Available options are ['mean', 'qr', 'iqn', 'fqf'].
- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are ['pixel', 'min_max', 'standard']
- **augmentation** (*d3rlpy.augmentation.AugmentationPipeline* or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.
- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model for data augmentation.
- **impl** (*d3rlpy.algos.torch.sac_impl.SACImpl*) – algorithm implementation.

actor_learning_rate

learning rate for policy function.

Type *float***critic_learning_rate**

learning rate for Q functions.

Type *float***temp_learning_rate**

learning rate for temperature parameter.

Type *float***actor_optim_factory**

optimizer factory for the actor.

Type *d3rlpy.optimizers.OptimizerFactory***critic_optim_factory**

optimizer factory for the critic.

Type *d3rlpy.optimizers.OptimizerFactory***temp_optim_factory**

optimizer factory for the temperature.

Type *d3rlpy.optimizers.OptimizerFactory***actor_encoder_factory**

encoder factory for the actor.

Type *d3rlpy.encoders.EncoderFactory***critic_encoder_factory**

encoder factory for the critic.

Type d3rlpy.encoders.EncoderFactory

batch_size
mini-batch size.

Type int

n_frames
the number of frames to stack for image observation.

Type int

gamma
discount factor.

Type float

n_critics
the number of Q functions for ensemble.

Type int

bootstrap
flag to bootstrap Q functions.

Type bool

share_encoder
flag to share encoder network.

Type bool

initial_temperature
initial temperature value.

Type float

q_func_type
type of Q function.

Type str

use_gpu
GPU device.

Type d3rlpy.gpu.Device

scaler
preprocessor.

Type d3rlpy.preprocessing.Scaler

augmentation
augmentation pipeline.

Type d3rlpy.augmentation.AugmentationPipeline

n_augmentations
the number of data augmentations to update.

Type int

dynamics
dynamics model.

Type d3rlpy.dynamics.base.DynamicsBase

impl
algorithm implementation.
Type `d3rlpy.algos.torch.sac_impl.SACImpl`

eval_results_
evaluation results.
Type `dict`

Methods

create_impl (*observation_shape*, *action_size*)
Instantiate implementation objects with the dataset shapes.
This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs*=1000, *save_metrics*=True, *experiment_name*=None, *with_timestamp*=True, *logdir*='d3rlpy_logs', *verbose*=True, *show_progress*=True, *tensorboard*=True, *eval_episodes*=None, *save_interval*=1, *scorers*=None)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

```
fit_online(env, buffer, explorer=None, n_steps=1000000, n_steps_per_epoch=10000,
            update_interval=1, update_start_step=0, eval_env=None, eval_epsilon=0.0,
            save_metrics=True, experiment_name=None, with_timestamp=True,
            logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard=True)
```

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (*d3rlpy.online.buffers.Buffer*) – replay buffer.
- **explorer** (*d3rlpy.online.explorers.Explorer*) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If False, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_online_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod from_json (*fname, use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.

- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters **deep** (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type *dict*

load_model (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters **fname** (*str*) – source file path.

predict (*x*)

Returns greedy actions.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters **x** (*numpy.ndarray*) – observations

Returns greedy actions

Return type *numpy.ndarray*

predict_value (*x, action, with_std=False*)

Returns predicted action-values.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

# for continuous control
# 100 actions with shape of (2,)
actions = np.random.random((100, 2))
```

(continues on next page)

(continued from previous page)

```
# for discrete control
# 100 actions in integer values
actions = np.random.randint(2, size=100)

values = algo.predict_value(x, actions)
# values.shape == (100,)

values, stds = algo.predict_value(x, actions, with_std=True)
# stds.shape == (100,)
```

Parameters

- **x** (*numpy.ndarray*) – observations
- **action** (*numpy.ndarray*) – actions
- **with_std** (*bool*) – flag to return standard deviation of ensemble estimation. This deviation reflects uncertainty for the given observations. This uncertainty will be more accurate if you enable *bootstrap* flag and increase *n_critics* value.

Returns predicted action-values**Return type** *numpy.ndarray***sample_action** (*x*)

Returns sampled actions.

The sampled actions are identical to the output of *predict* method if the policy is deterministic.**Parameters** **x** (*numpy.ndarray*) – observations.**Returns** sampled actions.**Return type** *numpy.ndarray***save_model** (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.**save_policy** (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).

- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type d3rlpy.algos.base.AlgoBase

update (*epoch, total_step, batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (d3rlpy.dataset.TransitionMiniBatch) – mini-batch data.

Returns loss values.

Return type list

d3rlpy.algos.DiscreteBCQ

```
class d3rlpy.algos.DiscreteBCQ(*,
                               learning_rate=6.25e-05,
                               optim_factory=<d3rlpy.optimizers.AdamFactory object>,
                               encoder_factory='default',
                               batch_size=32,
                               n_frames=1,
                               gamma=0.99,
                               n_critics=1,
                               bootstrap=False,
                               share_encoder=False,
                               action_flexibility=0.3,
                               beta=0.5,
                               target_update_interval=8000.0,
                               q_func_type='mean',
                               use_gpu=False,
                               scaler=None,
                               augmentation=None,
                               n_augmentations=1,
                               dynamics=None,
                               impl=None,
                               **kwargs)
```

Discrete version of Batch-Constrained Q-learning algorithm.

Discrete version takes theories from the continuous version, but the algorithm is much simpler than that. The imitation function $G_{\omega}(a|s)$ is trained as supervised learning just like Behavior Cloning.

$$L(\omega) = \mathbb{E}_{a_t, s_t \sim D} \left[- \sum_a p(a|s_t) \log G_{\omega}(a|s_t) \right]$$

With this imitation function, the greedy policy is defined as follows.

$$\pi(s_t) = \operatorname{argmax}_{a|G_{\omega}(a|s_t)/\max_{\bar{a}} G_{\omega}(\bar{a}|s_t) > \tau} Q_{\theta}(s_t, a)$$

which eliminates actions with probabilities τ times smaller than the maximum one.

Finally, the loss function is computed in Double DQN style with the above constrained policy.

$$L(\theta) = \mathbb{E}_{s_t, a_t, r_{t+1}, s_{t+1} \sim D} [(r_{t+1} + \gamma Q_{\theta'}(s_{t+1}, \pi(s_{t+1})) - Q_{\theta}(s_t, a_t))^2]$$

References

- Fujimoto et al., Off-Policy Deep Reinforcement Learning without Exploration.
- Fujimoto et al., Benchmarking Batch Deep Reinforcement Learning Algorithms.

Parameters

- **learning_rate** (*float*) – learning rate.
- **optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory.
- **encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **gamma** (*float*) – discount factor.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **share_encoder** (*bool*) – flag to share encoder network.
- **action_flexibility** (*float*) – probability threshold represented as :math: ‘au’.
- **beta** (*float*) – reguralization term for imitation function.
- **target_update_interval** (*int*) – interval to update the target network.
- **q_func_type** (*str*) – type of Q function. Available options are [*‘mean’*, *‘qr’*, *‘iqn’*, *‘fqf’*].
- **use_gpu** (*bool*, *int* or `d3rlpy.gpu.Device`) – flag to use GPU, device ID or device.
- **scaler** (`d3rlpy.preprocessingScaler` or *str*) – preprocessor. The available options are [*‘pixel’*, *‘min_max’*, *‘standard’*]
- **augmentation** (`d3rlpy.augmentation.AugmentationPipeline` or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.
- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model for data augmentation.
- **impl** (`d3rlpy.algos.torch.bcq_impl.DiscreteBCQImpl`) – algorithm implementation.

learning_rate

learning rate.

Type *float*

optim_factory
optimizer factory.
Type *d3rlpy.optimizers.OptimizerFactory*

encoder_factory
encoder factory.
Type *d3rlpy.encoders.EncoderFactory*

batch_size
mini-batch size.
Type *int*

n_frames
the number of frames to stack for image observation.
Type *int*

gamma
discount factor.
Type *float*

n_critics
the number of Q functions for ensemble.
Type *int*

bootstrap
flag to bootstrap Q functions.
Type *bool*

share_encoder
flag to share encoder network.
Type *bool*

action_flexibility
probability threshold represented as ϵ .
Type *float*

beta
regularization term for imitation function.
Type *float*

target_update_interval
interval to update the target network.
Type *int*

q_func_type
type of Q function.
Type *str*

use_gpu
GPU device.
Type *d3rlpy.gpu.Device*

scaler
preprocessor.

Type `d3rlpy.preprocessing.Scaler`

augmentation

augmentation pipeline.

Type `d3rlpy.augmentation.AugmentationPipeline`

n_augmentations

the number of data augmentations to update.

Type `int`

dynamics

dynamics model.

Type `d3rlpy.dynamics.base.DynamicsBase`

impl

algorithm implementation.

Type `d3rlpy.algos.torch.bcq_impl.DiscreteBCQImpl`

eval_results_

evaluation results.

Type `dict`

Methods

create_impl (*observation_shape*, *action_size*)

Instantiate implementation objects with the dataset shapes.

This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs=1000*, *save_metrics=True*, *experiment_name=None*, *with_timestamp=True*, *logdir='d3rlpy_logs'*, *verbose=True*, *show_progress=True*, *tensorboard=True*, *eval_episodes=None*, *save_interval=1*, *scorers=None*)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.

- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

fit_online (*env*, *buffer*, *explorer=None*, *n_steps=1000000*, *n_steps_per_epoch=10000*, *update_interval=1*, *update_start_step=0*, *eval_env=None*, *eval_epsilon=0.0*, *save_metrics=True*, *experiment_name=None*, *with_timestamp=True*, *logdir='d3rlpy_logs'*, *verbose=True*, *show_progress=True*, *tensorboard=True*)

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (*d3rlpy.online.buffers.Buffer*) – replay buffer.
- **explorer** (*d3rlpy.online.explorers.Explorer*) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If *None*, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If *False*, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_online_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod from_json (*fname*, *use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')
```

(continues on next page)

(continued from previous page)

```
# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.**Return type** *d3rlpy.base.LearnableBase***get_params** (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters **deep** (*bool*) – flag to deeply copy objects such as *impl*.**Returns** attribute values in dictionary.**Return type** *dict***load_model** (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters **fname** (*str*) – source file path.**predict** (*x*)

Returns greedy actions.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters **x** (*numpy.ndarray*) – observations**Returns** greedy actions**Return type** *numpy.ndarray*

predict_value (*x*, *action*, *with_std=False*)

Returns predicted action-values.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

# for continuous control
# 100 actions with shape of (2,)
actions = np.random.random((100, 2))

# for discrete control
# 100 actions in integer values
actions = np.random.randint(2, size=100)

values = algo.predict_value(x, actions)
# values.shape == (100,)

values, stds = algo.predict_value(x, actions, with_std=True)
# stds.shape == (100,)
```

Parameters

- **x** (*numpy.ndarray*) – observations
- **action** (*numpy.ndarray*) – actions
- **with_std** (*bool*) – flag to return standard deviation of ensemble estimation. This deviation reflects uncertainty for the given observations. This uncertainty will be more accurate if you enable *bootstrap* flag and increase *n_critics* value.

Returns predicted action-values

Return type *numpy.ndarray*

sample_action (*x*)

Returns sampled actions.

The sampled actions are identical to the output of *predict* method if the policy is deterministic.

Parameters **x** (*numpy.ndarray*) – observations.

Returns sampled actions.

Return type *numpy.ndarray*

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.

save_policy (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).
- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type d3rlpy.algos.base.AlgoBase

update (*epoch, total_step, batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (d3rlpy.dataset.TransitionMiniBatch) – mini-batch data.

Returns loss values.

Return type list

d3rlpy.algos.DiscreteCQL

```
class d3rlpy.algos.DiscreteCQL(*, learning_rate=6.25e-05, op-
tim_factory=<d3rlpy.optimizers.AdamFactory ob-
ject>, encoder_factory='default', batch_size=32,
n_frames=1, gamma=0.99, n_critics=1, bootstrap=False,
share_encoder=False, target_update_interval=8000.0,
q_func_type='mean', use_gpu=False, scaler=None, augmenta-
tion=None, n_augmentations=1, dynamics=None, impl=None,
**kwargs)
```

Discrete version of Conservative Q-Learning algorithm.

Discrete version of CQL is a DoubleDQN-based data-driven deep reinforcement learning algorithm (the original paper uses DQN), which achieves state-of-the-art performance in offline RL problems.

CQL mitigates overestimation error by minimizing action-values under the current policy and maximizing values under data distribution for underestimation issue.

$$L(\theta) = \mathbb{E}_{s_t \sim D} [\log \sum_a \exp Q_\theta(s_t, a) - \mathbb{E}_{a \sim D} [Q_\theta(s, a)]] + L_{DoubleDQN}(\theta)$$

References

- Kumar et al., Conservative Q-Learning for Offline Reinforcement Learning.

Parameters

- **learning_rate** (*float*) – learning rate.
- **optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory.
- **encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **gamma** (*float*) – discount factor.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **bootstrap** (*bool*) – flag to bootstrap Q functions.
- **target_update_interval** (*int*) – interval to synchronize the target network.
- **q_func_type** (*str*) – type of Q function. Available options are ['mean', 'qr', 'iqn', 'fqf'].
- **use_gpu** (*bool*, *int* or `d3rlpy.gpu.Device`) – flag to use GPU, device ID or device.
- **scaler** (`d3rlpy.preprocessingScaler` or *str*) – preprocessor. The available options are ['pixel', 'min_max', 'standard']
- **augmentation** (`d3rlpy.augmentation.AugmentationPipeline` or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.
- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model for data augmentation.
- **impl** (`d3rlpy.algos.torch.cql_impl.DiscreteCQLImpl`) – algorithm implementation.

learning_rate

learning rate.

Type *float*

optim_factory

optimizer factory.

Type `d3rlpy.optimizers.OptimizerFactory`

encoder_factory

encoder factory.

Type d3rlpy.encoders.EncoderFactory

batch_size
mini-batch size.

Type int

n_frames
the number of frames to stack for image observation.

Type int

gamma
discount factor.

Type float

n_critics
the number of Q functions for ensemble.

Type int

bootstrap
flag to bootstrap Q functions.

Type bool

target_update_interval
interval to synchronize the target network.

Type int

q_func_type
type of Q function.

Type str

use_gpu
GPU device.

Type d3rlpy.gpu.Device

scaler
preprocessor.

Type d3rlpy.preprocessing.Scaler

augmentation
augmentation pipeline.

Type d3rlpy.augmentation.AugmentationPipeline

n_augmentations
the number of data augmentations to update.

Type int

dynamics
dynamics model.

Type d3rlpy.dynamics.base.DynamicsBase

impl
algorithm implementation.

Type d3rlpy.algos.torch.CQLImpl.DiscreteCQLImpl

eval_results_
evaluation results.

Type `dict`

Methods

create_impl (*observation_shape*, *action_size*)

Instantiate implementation objects with the dataset shapes.

This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs*=1000, *save_metrics*=True, *experiment_name*=None, *with_timestamp*=True, *logdir*='d3rlpy_logs', *verbose*=True, *show_progress*=True, *tensorboard*=True, *eval_episodes*=None, *save_interval*=1, *scorers*=None)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

fit_online (*env*, *buffer*, *explorer*=None, *n_steps*=1000000, *n_steps_per_epoch*=10000, *update_interval*=1, *update_start_step*=0, *eval_env*=None, *eval_epsilon*=0.0, *save_metrics*=True, *experiment_name*=None, *with_timestamp*=True, *logdir*='d3rlpy_logs', *verbose*=True, *show_progress*=True, *tensorboard*=True)

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (*d3rlpy.online.buffer.Buffer*) – replay buffer.
- **explorer** (*d3rlpy.online.explorers.Explorer*) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If False, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_online_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod from_json (*fname, use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool, int or d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters `deep` (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type `dict`

load_model (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters `fname` (*str*) – source file path.

predict (*x*)

Returns greedy actions.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters `x` (*numpy.ndarray*) – observations

Returns greedy actions

Return type `numpy.ndarray`

predict_value (*x, action, with_std=False*)

Returns predicted action-values.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

# for continuous control
# 100 actions with shape of (2,)
actions = np.random.random((100, 2))

# for discrete control
# 100 actions in integer values
actions = np.random.randint(2, size=100)

values = algo.predict_value(x, actions)
# values.shape == (100,)

values, stds = algo.predict_value(x, actions, with_std=True)
# stds.shape == (100,)
```

Parameters

- **x** (*numpy.ndarray*) – observations
- **action** (*numpy.ndarray*) – actions
- **with_std** (*bool*) – flag to return standard deviation of ensemble estimation. This deviation reflects uncertainty for the given observations. This uncertainty will be more accurate if you enable *bootstrap* flag and increase *n_critics* value.

Returns predicted action-values

Return type *numpy.ndarray*

sample_action (*x*)

Returns sampled actions.

The sampled actions are identical to the output of *predict* method if the policy is deterministic.

Parameters **x** (*numpy.ndarray*) – observations.

Returns sampled actions.

Return type *numpy.ndarray*

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.

save_policy (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).
- <https://onnx.ai> (for ONNX)

Parameters

- **fname** (*str*) – destination file path.
- **as_onnx** (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type d3rlpy.algos.base.AlgoBase

update (*epoch*, *total_step*, *batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (d3rlpy.dataset.TransitionMiniBatch) – mini-batch data.

Returns loss values.

Return type list

d3rlpy.algos.DiscreteAWR

```
class d3rlpy.algos.DiscreteAWR(*, actor_learning_rate=5e-05, critic_learning_rate=0.0001,
                                actor_optim_factory=<d3rlpy.optimizers.SGDFactory object>,
                                critic_optim_factory=<d3rlpy.optimizers.SGDFactory object>,
                                actor_encoder_factory='default',
                                critic_encoder_factory='default', batch_size=2048,
                                n_frames=1, gamma=0.99, batch_size_per_update=256,
                                n_actor_updates=1000, n_critic_updates=200, lam=0.95,
                                beta=1.0, max_weight=20.0, use_gpu=False, scaler=None,
                                augmentation=None, n_augmentations=1, dynamics=None,
                                impl=None, **kwargs)
```

Discrete version of Advantage-Weighted Regression algorithm.

AWR is an actor-critic algorithm that trains via supervised regression way, and has shown strong performance in online and offline settings.

The value function is trained as a supervised regression problem.

$$L(\theta) = \mathbb{E}_{s_t, R_t \sim D} [(R_t - V(s_t|\theta))^2]$$

where R_t is approximated using TD(λ) to mitigate high variance issue.

The policy function is also trained as a supervised regression problem.

$$J(\phi) = \mathbb{E}_{s_t, a_t, R_t \sim D} [\log \pi(a_t|s_t, \phi) \exp(\frac{1}{B}(R_t - V(s_t|\theta)))]$$

where B is a constant factor.

References

- Peng et al., Advantage-Weighted Regression: Simple and Scalable Off-Policy Reinforcement Learning

Parameters

- **actor_learning_rate** (*float*) – learning rate for policy function.
- **critic_learning_rate** (*float*) – learning rate for value function.
- **actor_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic_optim_factory** (`d3rlpy.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **actor_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the actor.
- **critic_encoder_factory** (`d3rlpy.encoders.EncoderFactory` or *str*) – encoder factory for the critic.
- **batch_size** (*int*) – batch size per iteration.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **gamma** (*float*) – discount factor.
- **batch_size_per_update** (*int*) – mini-batch size.
- **n_actor_updates** (*int*) – actor gradient steps per iteration.
- **n_critic_updates** (*int*) – critic gradient steps per iteration.
- **lam** (*float*) – λ for TD(λ).
- **beta** (*float*) – B for weight scale.
- **max_weight** (*float*) – w_{\max} for weight clipping.
- **use_gpu** (*bool*, *int* or `d3rlpy.gpu.Device`) – flag to use GPU, device ID or device.
- **scaler** (`d3rlpy.preprocessing.Scaler` or *str*) – preprocessor. The available options are [*'pixel'*, *'min_max'*, *'standard'*]
- **augmentation** (`d3rlpy.augmentation.AugmentationPipeline` or *list(str)*) – augmentation pipeline.
- **n_augmentations** (*int*) – the number of data augmentations to update.
- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model for data augmentation.
- **impl** (`d3rlpy.algos.torch.awr_impl.DiscreteAWRImpl`) – algorithm implementation.

actor_learning_rate
learning rate for policy function.

Type *float*

critic_learning_rate
learning rate for value function.

Type *float*

actor_optim_factory
optimizer factory for the actor.

Type *d3rlpy.optimizers.OptimizerFactory*

critic_optim_factory
optimizer factory for the critic.

Type *d3rlpy.optimizers.OptimizerFactory*

actor_encoder_factory
encoder factory for the actor.

Type *d3rlpy.encoders.EncoderFactory*

critic_encoder_factory
encoder factory for the critic.

Type *d3rlpy.encoders.EncoderFactory*

batch_size
batch size per iteration.

Type *int*

n_frames
the number of frames to stack for image observation.

Type *int*

gamma
discount factor.

Type *float*

batch_size_per_update
mini-batch size.

Type *int*

n_actor_updates
actor gradient steps per iteration.

Type *int*

n_critic_updates
critic gradient steps per iteration.

Type *int*

lam
 λ for TD(λ).

Type *float*

beta
 B for weight scale.

Type *float*

max_weight
 w_{\max} for weight clipping.

Type *float*

use_gpu
GPU device.

Type *d3rlpy.gpu.Device*

scaler
preprocessor.

Type `d3rlpy.preprocessingScaler`

augmentation

augmentation pipeline.

Type `d3rlpy.augmentation.AugmentationPipeline`

n_augmentations

the number of data augmentations to update.

Type `int`

dynamics

dynamics model.

Type `d3rlpy.dynamics.base.DynamicsBase`

impl

algorithm implementation.

Type `d3rlpy.algos.torch.awr_impl.DiscreteAWRImpl`

eval_results_

evaluation results.

Type `dict`

Methods

create_impl (*observation_shape*, *action_size*)

Instantiate implementation objects with the dataset shapes.

This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs=1000*, *save_metrics=True*, *experiment_name=None*, *with_timestamp=True*, *logdir='d3rlpy_logs'*, *verbose=True*, *show_progress=True*, *tensorboard=True*, *eval_episodes=None*, *save_interval=1*, *scorers=None*)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.

- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

fit_online (*env*, *buffer*, *explorer=None*, *n_steps=1000000*, *n_steps_per_epoch=10000*, *update_interval=1*, *update_start_step=0*, *eval_env=None*, *eval_epsilon=0.0*, *save_metrics=True*, *experiment_name=None*, *with_timestamp=True*, *logdir='d3rlpy_logs'*, *verbose=True*, *show_progress=True*, *tensorboard=True*)

Start training loop of online deep reinforcement learning.

This method is a convenient alias to `d3rlpy.online.iterators.train`.

Parameters

- **env** (*gym.Env*) – gym-like environment.
- **buffer** (*d3rlpy.online.buffers.Buffer*) – replay buffer.
- **explorer** (*d3rlpy.online.explorers.Explorer*) – action explorer.
- **n_steps** (*int*) – the number of total steps to train.
- **n_steps_per_epoch** (*int*) – the number of steps per epoch.
- **update_interval** (*int*) – the number of steps per update.
- **update_start_step** (*int*) – the steps before starting updates.
- **eval_env** (*gym.Env*) – gym-like environment. If *None*, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ -greedy factor during evaluation.
- **save_metrics** (*bool*) – flag to record metrics. If *False*, the log directory is not created and the model parameters are not saved.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be `{class name}_online_{timestamp}`.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)

classmethod from_json (*fname*, *use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')
```

(continues on next page)

(continued from previous page)

```
# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.**Return type** *d3rlpy.base.LearnableBase***get_params** (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters **deep** (*bool*) – flag to deeply copy objects such as *impl*.**Returns** attribute values in dictionary.**Return type** *dict***load_model** (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters **fname** (*str*) – source file path.**predict** (*x*)

Returns greedy actions.

```
# 100 observations with shape of (10,)
x = np.random.random((100, 10))

actions = algo.predict(x)
# actions.shape == (100, action size) for continuous control
# actions.shape == (100,) for discrete control
```

Parameters **x** (*numpy.ndarray*) – observations**Returns** greedy actions**Return type** *numpy.ndarray*

predict_value (*x*, **args*, ***kwargs*)

Returns predicted state values.

Parameters *x* (*numpy.ndarray*) – observations.

Returns predicted state values.

Return type *numpy.ndarray*

sample_action (*x*)

Returns sampled actions.

The sampled actions are identical to the output of *predict* method if the policy is deterministic.

Parameters *x* (*numpy.ndarray*) – observations.

Returns sampled actions.

Return type *numpy.ndarray*

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters *fname* (*str*) – destination file path.

save_policy (*fname*, *as_onnx=False*)

Save the greedy-policy computational graph as TorchScript or ONNX.

```
# save as TorchScript
algo.save_policy('policy.pt')

# save as ONNX
algo.save_policy('policy.onnx', as_onnx=True)
```

The artifacts saved with this method will work without d3rlpy. This method is especially useful to deploy the learned policy to production environments or embedding systems.

See also

- https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html (for Python).
- https://pytorch.org/tutorials/advanced/cpp_export.html (for C++).
- <https://onnx.ai> (for ONNX)

Parameters

- *fname* (*str*) – destination file path.
- *as_onnx* (*bool*) – flag to save as ONNX format.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ***params* – arbitrary inputs to set as attributes.

Returns itself.

Return type `d3rlpy.algos.base.AlgoBase`

update (*epoch*, *itr*, *batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (`d3rlpy.dataset.TransitionMiniBatch`) – mini-batch data.

Returns loss values.

Return type `list`

3.2 Q Functions

d3rlpy provides various Q functions including state-of-the-arts, which are internally used in algorithm objects. You can switch Q functions by passing *q_func_type* argument at algorithm initialization.

```
from d3rlpy.algos import CQL

cql = CQL(q_func_type='qr') # use Quantile Regression Q function
```

The default Q function is *mean* approximator, which estimates expected scalar action-values. However, in recent advancements of deep reinforcement learning, the new type of action-value approximators has been proposed, which is called *distributional* Q functions.

Unlike the *mean* approximator, the *distributional* Q functions estimate distribution of action-values. This *distributional* approaches have shown consistently much stronger performance than the *mean* approximator.

Here is a list of available Q functions in the order of performance ascendingly. Currently, as a trade-off between performance and computational complexity, the higher performance requires the more expensive computational costs.

Table 3: available Q functions

q_func_type	reference
mean (default)	N/A
qr	Quantile Regression
iqn	Implicit Quantile Network
fqf (experimental)	Fully-parametrized Quantile Function

3.3 MDPDataset

d3rlpy provides useful dataset structure for data-driven deep reinforcement learning. In supervised learning, the training script iterates input data X and label data Y . However, in reinforcement learning, mini-batches consist with sets of $(s_t, a_t, r_{t+1}, s_{t+1})$ and episode terminal flags. Converting a set of observations, actions, rewards and terminal flags into this tuples is boring and requires some codings.

Therefore, d3rlpy provides *MDPDataset* class which enables you to handle reinforcement learning datasets without any efforts.

```

from d3rlpy.dataset import MDPDataset

# 1000 steps of observations with shape of (100,)
observations = np.random.random((1000, 100))
# 1000 steps of actions with shape of (4,)
actions = np.random.random((1000, 4))
# 1000 steps of rewards
rewards = np.random.random(1000)
# 1000 steps of terminal flags
terminals = np.random.randint(2, size=1000)

dataset = MDPDataset(observations, actions, rewards, terminals)

# automatically splitted into d3rlpy.dataset.Episode objects
dataset.episodes

# each episode is also splitted into d3rlpy.dataset.Transition objects
episode = dataset.episodes[0]
episode[0].observation
episode[0].action
episode[0].next_reward
episode[0].next_observation
episode[0].terminal

# d3rlpy.dataset.Transition object has pointers to previous and next
# transitions like linked list.
transition = episode[0]
while transition.next_transition:
    transition = transition.next_transition

# save as HDF5
dataset.dump('dataset.h5')

# load from HDF5
new_dataset = MDPDataset.load('dataset.h5')

```

<code>d3rlpy.dataset.MDPDataset</code>	Markov-Decision Process Dataset class.
<code>d3rlpy.dataset.Episode</code>	Episode class.
<code>d3rlpy.dataset.Transition</code>	Transition class.
<code>d3rlpy.dataset.TransitionMiniBatch</code>	mini-batch of Transition objects.

3.3.1 d3rlpy.dataset.MDPDataset

class `d3rlpy.dataset.MDPDataset` (*observations, actions, rewards, terminals, discrete_action=False*)

Markov-Decision Process Dataset class.

MDPDataset is deisnged for reinforcement learning datasets to use them like supervised learning datasets.

```

from d3rlpy.dataset import MDPDataset

# 1000 steps of observations with shape of (100,)
observations = np.random.random((1000, 100))
# 1000 steps of actions with shape of (4,)
actions = np.random.random((1000, 4))

```

(continues on next page)

(continued from previous page)

```
# 1000 steps of rewards
rewards = np.random.random(1000)
# 1000 steps of terminal flags
terminals = np.random.randint(2, size=1000)

dataset = MDPDataset(observations, actions, rewards, terminals)
```

The MDPDataset object automatically splits the given data into list of `d3rlpy.dataset.Episode` objects. Furthermore, the MDPDataset object behaves like a list in order to use with scikit-learn utilities.

```
# returns the number of episodes
len(dataset)

# access to the first episode
episode = dataset[0]

# iterate through all episodes
for episode in dataset:
    pass
```

Parameters

- **observations** (`numpy.ndarray`) – N-D array. If the observation is a vector, the shape should be $(N, \text{dim_observation})$. If the observations is an image, the shape should be (N, C, H, W) .
- **actions** (`numpy.ndarray`) – N-D array. If the actions-space is continuous, the shape should be $(N, \text{dim_action})$. If the action-space is discrete, the shape should be $(N,)$.
- **rewards** (`numpy.ndarray`) – array of scalar rewards.
- **terminals** (`numpy.ndarray`) – array of binary terminal flags.
- **discrete_action** (`bool`) – flag to use the given actions as discrete action-space actions.

Methods

`__getitem__` (*index*)

`__len__` ()

`__iter__` ()

append (*observations, actions, rewards, terminals*)

Appends new data.

Parameters

- **observations** (`numpy.ndarray`) – N-D array.
- **actions** (`numpy.ndarray`) – actions.
- **rewards** (`numpy.ndarray`) – rewards.
- **terminals** (`numpy.ndarray`) – terminals.

build_episodes ()

Builds episode objects.

This method will be internally called when accessing the episodes property at the first time.

clip_reward (*low=None, high=None*)

Clips rewards in the given range.

Parameters

- **low** (*float*) – minimum value. If None, clipping is not performed on lower edge.
- **high** (*float*) – maximum value. If None, clipping is not performed on upper edge.

compute_stats ()

Computes statistics of the dataset.

```
stats = dataset.compute_stats()

# return statistics
stats['return']['mean']
stats['return']['std']
stats['return']['min']
stats['return']['max']

# reward statistics
stats['reward']['mean']
stats['reward']['std']
stats['reward']['min']
stats['reward']['max']

# action (only with continuous control actions)
stats['action']['mean']
stats['action']['std']
stats['action']['min']
stats['action']['max']

# observation (only with numpy.ndarray observations)
stats['observation']['mean']
stats['observation']['std']
stats['observation']['min']
stats['observation']['max']
```

Returns statistics of the dataset.

Return type dict

dump (*fname*)

Saves dataset as HDF5.

Parameters **fname** (*str*) – file path.

extend (*dataset*)

Extend dataset by another dataset.

Parameters **dataset** (`d3rlpy.dataset.MDPDataset`) – dataset.

get_action_size ()

Returns dimension of action-space.

If *discrete_action=True*, the return value will be the maximum index +1 in the give actions.

Returns dimension of action-space.

Return type int

get_observation_shape()

Returns observation shape.

Returns observation shape.

Return type `tuple`

is_action_discrete()

Returns *discrete_action* flag.

Returns *discrete_action* flag.

Return type `bool`

classmethod load(fname)

Loads dataset from HDF5.

```
import numpy as np
from d3rlpy.dataset import MDPDataset

dataset = MDPDataset(np.random.random(10, 4),
                     np.random.random(10, 2),
                     np.random.random(10),
                     np.random.randint(2, size=10))

# save as HDF5
dataset.dump('dataset.h5')

# load from HDF5
new_dataset = MDPDataset.load('dataset.h5')
```

Parameters *fname* (*str*) – file path.

size()

Returns the number of episodes in the dataset.

Returns the number of episodes.

Return type `int`

Attributes

actions

Returns the actions.

Returns array of actions.

Return type `numpy.ndarray`

episodes

Returns the episodes.

Returns list of `d3rlpy.dataset.Episode` objects.

Return type `list(d3rlpy.dataset.Episode)`

observations

Returns the observations.

Returns array of observations.

Return type `numpy.ndarray`

rewards

Returns the rewards.

Returns array of rewards

Return type `numpy.ndarray`

terminals

Returns the terminal flags.

Returns array of terminal flags.

Return type `numpy.ndarray`

3.3.2 d3rlpy.dataset.Episode

class `d3rlpy.dataset.Episode` (*observation_shape*, *action_size*, *observations*, *actions*, *rewards*)
Episode class.

This class is designed to hold data collected in a single episode.

Episode object automatically splits data into list of `d3rlpy.dataset.Transition` objects. Also Episode object behaves like a list object for ease of access to transitions.

```
# return the number of transitions
len(episode)

# access to the first transition
transitions = episode[0]

# iterate through all transitions
for transition in episode:
    pass
```

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.
- **observations** (*numpy.ndarray*) – observations.
- **actions** (*numpy.ndarray*) – actions.
- **rewards** (*numpy.ndarray*) – scalar rewards.
- **terminals** (*numpy.ndarray*) – binary terminal flags.

Methods

`__getitem__` (*index*)

`__len__` ()

`__iter__` ()

build_transitions ()

Builds transition objects.

This method will be internally called when accessing the transitions property at the first time.

compute_return()
Computes sum of rewards.

$$R = \sum_{i=1} r_i$$

Returns episode return.

Return type float

get_action_size()
Returns dimension of action-space.

Returns dimension of action-space.

Return type int

get_observation_shape()
Returns observation shape.

Returns observation shape.

Return type tuple

size()
Returns the number of transitions.

Returns the number of transitions.

Return type int

Attributes

actions
Returns the actions.

Returns array of actions.

Return type numpy.ndarray

observations
Returns the observations.

Returns array of observations.

Return type numpy.ndarray

rewards
Returns the rewards.

Returns array of rewards.

Return type numpy.ndarray

transitions
Returns the transitions.

Returns list of *d3rlpy.dataset.Transition* objects.

Return type list(*d3rlpy.dataset.Transition*)

3.3.3 d3rlpy.dataset.Transition

class d3rlpy.dataset.Transition

Transition class.

This class is designed to hold data between two time steps, which is usually used as inputs of loss calculation in reinforcement learning.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.
- **observation** (*numpy.ndarray*) – observation at t .
- **action** (*numpy.ndarray* or *int*) – action at t .
- **reward** (*float*) – reward at t .
- **next_observation** (*numpy.ndarray*) – observation at $t+1$.
- **next_action** (*numpy.ndarray* or *int*) – action at $t+1$.
- **next_reward** (*float*) – reward at $t+1$.
- **terminal** (*int*) – terminal flag at $t+1$.
- **prev_transition** (*d3rlpy.dataset.Transition*) – pointer to the previous transition.
- **next_transition** (*d3rlpy.dataset.Transition*) – pointer to the next transition.

Methods

clear_links()

Clears links to the next and previous transitions.

This method is necessary to call when freeing this instance by GC.

get_action_size()

Returns dimension of action-space.

Returns dimension of action-space.

Return type *int*

get_observation_shape()

Returns observation shape.

Returns observation shape.

Return type *tuple*

Attributes

action

Returns action at t .

Returns action at t .

Return type (*numpy.ndarray* or *int*)

next_action

Returns action at $t+1$.

Returns action at $t+1$.

Return type (`numpy.ndarray` or `int`)

next_observation

Returns observation at $t+1$.

Returns observation at $t+1$.

Return type `numpy.ndarray` or `torch.Tensor`

next_reward

Returns reward at $t+1$.

Returns reward at $t+1$.

Return type `float`

next_transition

Returns pointer to the next transition.

If this is the last transition, this method should return `None`.

Returns next transition.

Return type `d3rlpy.dataset.Transition`

observation

Returns observation at t .

Returns observation at t .

Return type `numpy.ndarray` or `torch.Tensor`

prev_transition

Returns pointer to the previous transition.

If this is the first transition, this method should return `None`.

Returns previous transition.

Return type `d3rlpy.dataset.Transition`

reward

Returns reward at t .

Returns reward at t .

Return type `float`

terminal

Returns terminal flag at $t+1$.

Returns terminal flag at $t+1$.

Return type `int`

3.3.4 `d3rlpy.dataset.TransitionMiniBatch`

class `d3rlpy.dataset.TransitionMiniBatch`
mini-batch of `Transition` objects.

This class is designed to hold `d3rlpy.dataset.Transition` objects for being passed to algorithms during fitting.

If the observation is image, you can stack arbitrary frames via `n_frames`.

```
transition.observation.shape == (3, 84, 84)

batch_size = len(transitions)

# stack 4 frames
batch = TransitionMiniBatch(transitions, n_frames=4)

# 4 frames x 3 channels
batch.observations.shape == (batch_size, 12, 84, 84)
```

This is implemented by tracing previous transitions through `prev_transition` property.

Parameters

- **transitions** (`list(d3rlpy.dataset.Transition)`) – mini-batch of transitions.
- **n_frames** (`int`) – the number of frames to stack for image observation.

Methods

size()

Returns size of mini-batch.

Returns mini-batch size.

Return type `int`

Attributes

actions

Returns mini-batch of actions at t .

Returns actions at t .

Return type `numpy.ndarray`

next_actions

Returns mini-batch of actions at $t+1$.

Returns actions at $t+1$.

Return type `numpy.ndarray`

next_observations

Returns mini-batch of observations at $t+1$.

Returns observations at $t+1$.

Return type `numpy.ndarray` or `torch.Tensor`

next_rewards

Returns mini-batch of rewards at $t+1$.

Returns rewards at $t+1$.

Return type `numpy.ndarray`

observations

Returns mini-batch of observations at t .

Returns observations at t .

Return type `numpy.ndarray` or `torch.Tensor`

rewards

Returns mini-batch of rewards at t .

Returns rewards at t .

Return type `numpy.ndarray`

terminals

Returns mini-batch of terminal flags at $t+1$.

Returns terminal flags at $t+1$.

Return type `numpy.ndarray`

transitions

Returns transitions.

Returns list of transitions.

Return type `d3rlpy.dataset.Transition`

3.4 Datasets

d3rlpy provides datasets for experimenting data-driven deep reinforcement learning algorithms.

<code>d3rlpy.datasets.get_cartpole</code>	Returns cartpole dataset and environment.
<code>d3rlpy.datasets.get_pendulum</code>	Returns pendulum dataset and environment.
<code>d3rlpy.datasets.get_pybullet</code>	Returns pybullet dataset and environment.
<code>d3rlpy.datasets.get_atari</code>	Returns atari dataset and environment.

3.4.1 d3rlpy.datasets.get_cartpole

`d3rlpy.datasets.get_cartpole()`

Returns cartpole dataset and environment.

The dataset is automatically downloaded to `d3rlpy_data/cartpole.pkl` if it does not exist.

Returns tuple of `d3rlpy.dataset.MDPDataset` and gym environment.

Return type `tuple`

3.4.2 d3rlpy.datasets.get_pendulum

`d3rlpy.datasets.get_pendulum()`

Returns pendulum dataset and environment.

The dataset is automatically downloaded to `d3rlpy_data/pendulum.pkl` if it does not exist.

Returns tuple of `d3rlpy.dataset.MDPDataset` and gym environment.

Return type `tuple`

3.4.3 d3rlpy.datasets.get_pybullet

`d3rlpy.datasets.get_pybullet(env_name)`

Returns pybullet dataset and environment.

The dataset is provided through d4rl-pybullet. See more details including available dataset from its GitHub page.

```
from d3rlpy.datasets import get_pybullet

dataset, env = get_pybullet('hopper-bullet-mixed-v0')
```

References

- <https://github.com/takuseno/d4rl-pybullet>

Parameters `env_name` (*str*) – environment id of d4rl-pybullet dataset.

Returns tuple of `d3rlpy.dataset.MDPDataset` and gym environment.

Return type tuple

3.4.4 d3rlpy.datasets.get_atari

`d3rlpy.datasets.get_atari(env_name)`

Returns atari dataset and environment.

The dataset is provided through d4rl-atari. See more details including available dataset from its GitHub page.

```
from d3rlpy.datasets import get_atari

dataset, env = get_atari('breakout-mixed-v0')
```

References

- <https://github.com/takuseno/d4rl-atari>

Parameters `env_name` (*str*) – environment id of d4rl-atari dataset.

Returns tuple of `d3rlpy.dataset.MDPDataset` and gym environment.

Return type tuple

3.5 Preprocessing

d3rlpy provides several preprocessors tightly incorporated with algorithms. Each preprocessor is implemented with PyTorch operation, which will be included in the model exported by `save_policy` method.

```
from d3rlpy.algos import CQL
from d3rlpy.dataset import MDPDataset

dataset = MDPDataset(...)
```

(continues on next page)

(continued from previous page)

```
# choose from ['pixel', 'min_max', 'standard'] or None
cql = CQL(scaler='standard')

# scaler is fitted from the given episodes
cql.fit(dataset.episodes)

# preprocessing is included in TorchScript
cql.save_policy('policy.pt')

# you don't need to take care of preprocessing at production
policy = torch.jit.load('policy.pt')
action = policy(unpreprocessed_x)
```

You can also initialize scalers by yourself.

```
from d3rlpy.preprocessing import StandardScaler

scaler = StandardScaler(mean=..., std=...)

cql = CQL(scaler=scaler)
```

<code>d3rlpy.preprocessing.PixelScaler</code>	Pixel normalization preprocessing.
<code>d3rlpy.preprocessing.MinMaxScaler</code>	Min-Max normalization preprocessing.
<code>d3rlpy.preprocessing.StandardScaler</code>	Standardization preprocessing.

3.5.1 d3rlpy.preprocessing.PixelScaler

class d3rlpy.preprocessing.PixelScaler
Pixel normalization preprocessing.

$$x' = x/255$$

```
from d3rlpy.dataset import MDPDataset
from d3rlpy.algos import CQL

dataset = MDPDataset(observations, actions, rewards, terminals)

# initialize algorithm with PixelScaler
cql = CQL(scaler='pixel')

cql.fit(dataset.episodes)
```

Methods

fit (*episodes*)

get_params (*deep=False*)
Returns scaling parameters.

PixelScaler returns empty dictionary.

Parameters *deep* (*bool*) – flag to deeply copy objects.

Returns empty dictionary.

Return type `dict`

get_type()

Returns a scaler type.

Returns scaler type.

Return type `str`

reverse_transform(x)

Returns reversely transformed observations.

Parameters \mathbf{x} (`torch.Tensor`) – normalized observation tensor.

Returns unnormalized pixel observation tensor.

Return type `torch.Tensor`

transform(x)

Returns normalized pixel observations.

Parameters \mathbf{x} (`torch.Tensor`) – pixel observation tensor.

Returns normalized pixel observation tensor.

Return type `torch.Tensor`

Attributes

TYPE = `'pixel'`

3.5.2 d3rlpy.preprocessing.MinMaxScaler

class `d3rlpy.preprocessing.MinMaxScaler` (*dataset=None, maximum=None, minimum=None*)
Min-Max normalization preprocessing.

$$x' = (x - \min x) / (\max x - \min x)$$

```
from d3rlpy.dataset import MDPDataset
from d3rlpy.algos import CQL

dataset = MDPDataset(observations, actions, rewards, terminals)

# initialize algorithm with MinMaxScaler
cql = CQL(scaler='min_max')

# scaler is initialized from the given episodes
cql.fit(dataset.episodes)
```

You can also initialize with `d3rlpy.dataset.MDPDataset` object or manually.

```
from d3rlpy.preprocessing import MinMaxScaler

# initialize with dataset
scaler = MinMaxScaler(dataset)
```

(continues on next page)

```
# initialize manually
minimum = observations.min(axis=0)
maximum = observations.max(axis=0)
scaler = MinMaxScaler(minimum=minimum, maximum=maximum)

cql = CQL(scaler=scaler)
```

Parameters

- **dataset** (`d3rlpy.dataset.MDPDataset`) – dataset object.
- **min** (`numpy.ndarray`) – minimum values at each entry.
- **max** (`numpy.ndarray`) – maximum values at each entry.

minimum

minimum values at each entry.

Type `numpy.ndarray`

maximum

maximum values at each entry.

Type `numpy.ndarray`

Methods

fit (*episodes*)

Fits minimum and maximum from list of episodes.

Parameters **episodes** (`list` (`d3rlpy.dataset.Episode`)) – list of episodes.

get_params (*deep=False*)

Returns scaling parameters.

Parameters **deep** (`bool`) – flag to deeply copy objects.

Returns *maximum* and *minimum*.

Return type `dict`

get_type ()

Returns a scaler type.

Returns scaler type.

Return type `str`

reverse_transform (*x*)

Returns reversely transformed observations.

Parameters **x** (`torch.Tensor`) – normalized observation tensor.

Returns unnormalized observation tensor.

Return type `torch.Tensor`

transform (*x*)

Returns normalized observation tensor.

Parameters **x** (`torch.Tensor`) – observation tensor.

Returns normalized observation tensor.

Return type torch.Tensor

Attributes

TYPE = 'min_max'

3.5.3 d3rlpy.preprocessing.StandardScaler

class d3rlpy.preprocessing.StandardScaler (dataset=None, mean=None, std=None)
Standardization preprocessing.

$$x' = (x - \mu) / \sigma$$

```
from d3rlpy.dataset import MDPDataset
from d3rlpy.algos import CQL

dataset = MDPDataset(observations, actions, rewards, terminals)

# initialize algorithm with StandardScaler
cql = CQL(scaler='standard')

# scaler is initialized from the given episodes
cql.fit(dataset.episodes)
```

You can initialize with `d3rlpy.dataset.MDPDataset` object or manually.

```
from d3rlpy.preprocessing import StandardScaler

# initialize with dataset
scaler = StandardScaler(dataset)

# initialize manually
mean = observations.mean(axis=0)
std = observations.std(axis=0)
scaler = StandardScaler(mean=mean, std=std)

cql = CQL(scaler=scaler)
```

Parameters

- **dataset** (d3rlpy.dataset.MDPDataset) – dataset object.
- **mean** (numpy.ndarray) – mean values at each entry.
- **std** (numpy.ndarray) – standard deviation at each entry.

mean

mean values at each entry.

Type numpy.ndarray

std

standard deviation values at each entry.

Type numpy.ndarray

Methods

fit (*episodes*)

Fits mean and standard deviation from list of episodes.

Parameters **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes.

get_params (*deep=False*)

Returns scaling parameters.

Parameters **deep** (*bool*) – flag to deeply copy objects.

Returns *mean* and *std*.

Return type `dict`

get_type ()

Returns a scaler type.

Returns scaler type.

Return type `str`

reverse_transform (*x*)

Returns reversely transformed observation tensor.

Parameters **x** (*torch.Tensor*) – standardized observation tensor.

Returns unstandardized observation tensor.

Return type `torch.Tensor`

transform (*x*)

Returns standardized observation tensor.

Parameters **x** (*torch.Tensor*) – observation tensor.

Returns standardized observation tensor.

Return type `torch.Tensor`

Attributes

TYPE = `'standard'`

3.6 Optimizers

d3rlpy provides `OptimizerFactory` that gives you flexible control over optimizers. `OptimizerFactory` takes PyTorch's optimizer class and its arguments to initialize, which you can check more [here](#).

```
from torch.optim import Adam
from d3rlpy.algos import DQN
from d3rlpy.optimizers import OptimizerFactory

# modify weight decay
optim_factory = OptimizerFactory(Adam, weight_decay=1e-4)

# set OptimizerFactory
dqn = DQN(optim_factory=optim_factory)
```

There are also convenient aliases.

```
from d3rlpy.optimizers import AdamFactory

# alias for Adam optimizer
optim_factory = AdamFactory(weight_decay=1e-4)

dqn = DQN(optim_factory=optim_factory)
```

<code>d3rlpy.optimizers.OptimizerFactory</code>	A factory class that creates an optimizer object in a lazy way.
<code>d3rlpy.optimizers.SGDFactory</code>	An alias for SGD optimizer.
<code>d3rlpy.optimizers.AdamFactory</code>	An alias for Adam optimizer.
<code>d3rlpy.optimizers.RMSpropFactory</code>	An alias for RMSprop optimizer.

3.6.1 d3rlpy.optimizers.OptimizerFactory

class d3rlpy.optimizers.OptimizerFactory(*optim_cls*, ***kwargs*)

A factory class that creates an optimizer object in a lazy way.

The optimizers in algorithms can be configured through this factory class.

```
from torch.optim import Adam
from d3rlpy.optimizers import OptimizerFactory
from d3rlpy.algos import DQN

factory = OptimizerFactory(Adam, eps=0.001)

dqn = DQN(optim_factory=factory)
```

Parameters

- **optim_cls** (*type or str*) – An optimizer class.
- **kwargs** (*any*) – arbitrary keyword-arguments.

optim_cls

An optimizer class.

Type *type*

optim_kwargs

given parameters for an optimizer.

Type *dict*

Methods

create (*params*, *lr*)

Returns an optimizer object.

Parameters

- **params** (*list*) – a list of PyTorch parameters.
- **lr** (*float*) – learning rate.

Returns an optimizer object.

Return type torch.optim.Optimizer

get_params (*deep=False*)
Returns optimizer parameters.

3.6.2 d3rlpy.optimizers.SGDFactory

class d3rlpy.optimizers.SGDFactory (*momentum=0, dampening=0, weight_decay=0, nesterov=False, **kwargs*)

An alias for SGD optimizer.

```
from d3rlpy.optimizers import SGDFactory

factory = SGDFactory(weight_decay=1e-4)
```

Parameters

- **momentum** (*float*) – momentum factor.
- **dampening** (*float*) – dampening for momentum.
- **weight_decay** (*float*) – weight decay (L2 penalty).
- **nesterov** (*bool*) – flag to enable Nesterov momentum.

optim_cls
torch.optim.SGD class.

Type *type*

optim_kwargs
given parameters for an optimizer.

Type *dict*

Methods

create (*params, lr*)
Returns an optimizer object.

Parameters

- **params** (*list*) – a list of PyTorch parameters.
- **lr** (*float*) – learning rate.

Returns an optimizer object.

Return type torch.optim.Optimizer

get_params (*deep=False*)
Returns optimizer parameters.

3.6.3 d3rlpy.optimizers.AdamFactory

class d3rlpy.optimizers.**AdamFactory** (*betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False, **kwargs*)

An alias for Adam optimizer.

```
from d3rlpy.optimizers import AdamFactory

factory = AdamFactory(weight_decay=1e-4)
```

Parameters

- **betas** (*tuple*) – coefficients used for computing running averages of gradient and its square.
- **eps** (*float*) – term added to the denominator to improve numerical stability.
- **weight_decay** (*float*) – weight decay (L2 penalty).
- **amsgrad** (*bool*) – flag to use the AMSGrad variant of this algorithm.

optim_cls

torch.optim.Adam class.

Type *type*

optim_kwargs

given parameters for an optimizer.

Type *dict*

Methods

create (*params, lr*)

Returns an optimizer object.

Parameters

- **params** (*list*) – a list of PyTorch parameters.
- **lr** (*float*) – learning rate.

Returns an optimizer object.

Return type torch.optim.Optimizer

get_params (*deep=False*)

Returns optimizer parameters.

3.6.4 d3rlpy.optimizers.RMSpropFactory

class d3rlpy.optimizers.**RMSpropFactory** (*alpha=0.99, eps=1e-08, weight_decay=0, momentum=0, centered=True, **kwargs*)

An alias for RMSprop optimizer.

```
from d3rlpy.optimizers import RMSpropFactory

factory = RMSpropFactory(weight_decay=1e-4)
```

Parameters

- **alpha** (*float*) – smoothing constant.
- **eps** (*float*) – term added to the denominator to improve numerical stability.
- **weight_decay** (*float*) – weight decay (L2 penalty).
- **momentum** (*float*) – momentum factor.
- **centered** (*bool*) – flag to compute the centered RMSProp, the gradient is normalized by an estimation of its variance.

optim_cls

`torch.optim.RMSprop` class.

Type *type*

optim_kwargs

given parameters for an optimizer.

Type *dict*

Methods**create** (*params, lr*)

Returns an optimizer object.

Parameters

- **params** (*list*) – a list of PyTorch parameters.
- **lr** (*float*) – learning rate.

Returns an optimizer object.

Return type `torch.optim.Optimizer`

get_params (*deep=False*)

Returns optimizer parameters.

3.7 Network Architectures

In d3rlpy, the neural network architecture is automatically selected based on observation shape. If the observation is image, the algorithm uses the Nature DQN-based encoder at each function. Otherwise, the standard MLP architecture that consists with two linear layers with 256 hidden units.

Furthermore, d3rlpy provides `EncoderFactory` that gives you flexible control over this neural network architectures.

```
from d3rlpy.algos import DQN
from d3rlpy.encoders import VectorEncoderFactory

# encoder factory
encoder_factory = VectorEncoderFactory(hidden_units=[300, 400],
                                       activation='tanh')

# set OptimizerFactory
dqn = DQN(encoder_factory=encoder_factory)
```

You can also build your own encoder factory.

```
import torch
import torch.nn as nn

from d3rlpy.encoders import EncoderFactory

# your own neural network
class CustomEncoder(nn.Module):
    def __init__(self, observation_shape, feature_size):
        self.feature_size = feature_size
        self.fc1 = nn.Linear(observation_shape[0], 64)
        self.fc2 = nn.Linear(64, feature_size)

    def forward(self, x):
        h = torch.relu(self.fc1(x))
        h = torch.relu(self.fc2(h))
        return h

    # THIS IS IMPORTANT!
    def get_feature_size(self):
        return self.feature_size

# your own encoder factory
class CustomEncoderFactory(EncoderFactory):
    TYPE = 'custom' # this is necessary

    def __init__(self, feature_size):
        self.feature_size = feature_size

    def create(self, observation_shape, action_size=None, discrete_action=False):
        return CustomEncoder(observation_shape, self.feature_size)

    def get_params(self, deep=False):
        return {
            'feature_size': self.feature_size
        }

dqn = DQN(encoder_factory=CustomEncoderFactory(feature_size=64))
```

You can also share the factory across functions as below.

```
class CustomEncoderWithAction(nn.Module):
    def __init__(self, observation_shape, action_size, feature_size):
        self.feature_size = feature_size
        self.fc1 = nn.Linear(observation_shape[0] + action_size, 64)
        self.fc2 = nn.Linear(64, feature_size)

    def forward(self, x, action): # action is also given
        h = torch.cat([x, action], dim=1)
        h = torch.relu(self.fc1(h))
        h = torch.relu(self.fc2(h))
        return h

    def get_feature_size(self):
        return self.feature_size

class CustomEncoderFactory(EncoderFactory):
```

(continues on next page)

(continued from previous page)

```

TYPE = 'custom' # this is necessary

def __init__(self, feature_size):
    self.feature_size = feature_size

def create(self, observation_shape, action_size=None, discrete_action=False):
    # branch based on if ``action_size`` is given.
    if action_size is None:
        return CustomEncoder(observation_shape, self.feature_size)
    else:
        return CustomEncoderWithAction(observation_shape,
                                         action_size,
                                         self.feature_size)

def get_params(self, deep=False):
    return {
        'feature_size': self.feature_size
    }

from d3rlpy.algos import SAC

factory = CustomEncoderFactory(feature_size=64)

sac = SAC(actor_encoder_factory=factory, critic_encoder_factory=factory)

```

If you want `from_json` method to load the algorithm configuration including your encoder configuration, you need to register your encoder factory.

```

from d3rlpy.encoders import register_encoder_factory

# register your own encoder factory
register_encoder_factory(CustomEncoderFactory)

# load algorithm from json
dqn = DQN.from_json('<path-to-json>/params.json')

```

Once you register your encoder factory, you can specify it via `TYPE` value.

```
dqn = DQN(encoder_factory='custom')
```

<code>d3rlpy.encoders.</code>	Default encoder factory class.
<code>DefaultEncoderFactory</code>	
<code>d3rlpy.encoders.PixelEncoderFactory</code>	Pixel encoder factory class.
<code>d3rlpy.encoders.VectorEncoderFactory</code>	Vector encoder factory class.

3.7.1 d3rlpy.encoders.DefaultEncoderFactory

class `d3rlpy.encoders.DefaultEncoderFactory` (*use_batch_norm=False*)
 Default encoder factory class.

This encoder factory returns an encoder based on observation shape.

Parameters `use_batch_norm` (*bool*) – flag to insert batch normalization layers.

use_batch_norm

flag to insert batch normalization layers.

Type `bool`

Methods

create (*observation_shape*, *action_size=None*, *discrete_action=False*)

Returns PyTorch's encoder module.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – action size. If None, the encoder does not take action as input.
- **discrete_action** (*bool*) – flag if action-space is discrete.

Returns an encoder object.

Return type `torch.nn.Module`

get_params (*deep=False*)

Returns encoder parameters.

Parameters **deep** (*bool*) – flag to deeply copy the parameters.

Returns encoder parameters.

Return type `dict`

get_type ()

Returns encoder type.

Returns encoder type.

Return type `str`

Attributes

TYPE = `'default'`

3.7.2 d3rlpy.encoders.PixelEncoderFactory

class `d3rlpy.encoders.PixelEncoderFactory` (*filters=None*, *feature_size=512*, *activation='relu'*, *use_batch_norm=False*)

Pixel encoder factory class.

This is the default encoder factory for image observation.

Parameters

- **filters** (*list*) – list of tuples consisting with (*filter_size*, *kernel_size*, *stride*). If None, Nature DQN-based architecture is used.
- **feature_size** (*int*) – the last linear layer size.
- **activation** (*str*) – activation function name.
- **use_batch_norm** (*bool*) – flag to insert batch normalization layers.

filters

list of tuples consisting with (*filter_size*, *kernel_size*, *stride*).

Type `list`

feature_size

the last linear layer size.

Type `int`

activation

activation function name.

Type `str`

use_batch_norm

flag to insert batch normalization layers.

Type `bool`

Methods

create (*observation_shape*, *action_size=None*, *discrete_action=False*)

Returns PyTorch's encoder module.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – action size. If None, the encoder does not take action as input.
- **discrete_action** (*bool*) – flag if action-space is discrete.

Returns an encoder object.

Return type `torch.nn.Module`

get_params (*deep=False*)

Returns encoder parameters.

Parameters **deep** (*bool*) – flag to deeply copy the parameters.

Returns encoder parameters.

Return type `dict`

get_type ()

Returns encoder type.

Returns encoder type.

Return type `str`

Attributes

TYPE = `'pixel'`

3.7.3 d3rlpy.encoders.VectorEncoderFactory

class `d3rlpy.encoders.VectorEncoderFactory` (*hidden_units=None*, *activation='relu'*,
use_batch_norm=False)

Vector encoder factory class.

This is the default encoder factory for vector observation.

Parameters

- **hidden_units** (*list*) – list of hidden unit sizes. If `None`, the standard architecture with `[256, 256]` is used.
- **activation** (*str*) – activation function name.
- **use_batch_norm** (*bool*) – flag to insert batch normalization layers.

hidden_units

list of hidden unit sizes.

Type *list***activation**

activation function name.

Type *str***use_batch_norm**

flag to insert batch normalization layers.

Type *bool***Methods****create** (*observation_shape, action_size=None, discrete_action=False*)

Returns PyTorch's encoder module.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – action size. If `None`, the encoder does not take action as input.
- **discrete_action** (*bool*) – flag if action-space is discrete.

Returns an encoder object.**Return type** `torch.nn.Module`**get_params** (*deep=False*)

Returns encoder parameters.

Parameters **deep** (*bool*) – flag to deeply copy the parameters.**Returns** encoder parameters.**Return type** *dict***get_type** ()

Returns encoder type.

Returns encoder type.**Return type** *str***Attributes****TYPE** = `'vector'`

3.8 Data Augmentation

d3rlpy provides data augmentation techniques tightly integrated with reinforcement learning algorithms.

1. Kostrikov et al., Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels.
2. Laskin et al., Reinforcement Learning with Augmented Data.

Efficient data augmentation potentially boosts algorithm performance significantly.

```
from d3rlpy.algos import DiscreteCQL

# choose data augmentation types
cql = DiscreteCQL(augmentation=['random_shift', 'intensity'],
                  n_augmentations=2)
```

You can also tune data augmentation parameters by yourself.

```
from d3rlpy.augmentation.image import RandomShift

random_shift = RandomShift(shift_size=10)

cql = DiscreteCQL(augmentation=[random_shift, 'intensity'],
                  n_augmentations=2)
```

3.8.1 Image Observation

<code>d3rlpy.augmentation.image.RandomShift</code>	Random shift augmentation.
<code>d3rlpy.augmentation.image.Cutout</code>	Cutout augmentation.
<code>d3rlpy.augmentation.image.HorizontalFlip</code>	Horizontal flip augmentation.
<code>d3rlpy.augmentation.image.VerticalFlip</code>	Vertical flip augmentation.
<code>d3rlpy.augmentation.image.RandomRotation</code>	Random rotation augmentation.
<code>d3rlpy.augmentation.image.Intensity</code>	Intensity augmentation.
<code>d3rlpy.augmentation.image.ColorJitter</code>	Color Jitter augmentation.

`d3rlpy.augmentation.image.RandomShift`

```
class d3rlpy.augmentation.image.RandomShift (shift_size=4)
    Random shift augmentation.
```

References

- Kostrikov et al., Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels.

Parameters `shift_size` (*int*) – size to shift image.

shift_size
size to shift image.

Type `int`

Methods

get_params (*deep=False*)

Returns augmentation parameters.

Parameters **deep** (*bool*) – flag to deeply copy objects.

Returns augmentation parameters.

Return type `dict`

get_type ()

Returns augmentation type.

Returns augmentation type.

Return type `str`

transform (*x*)

Returns shifted images.

Parameters **x** (*torch.Tensor*) – observation tensor.

Returns processed observation tensor.

Return type `torch.Tensor`

Attributes

TYPE = `'random_shift'`

d3rlpy.augmentation.image.Cutout

class `d3rlpy.augmentation.image.Cutout` (*probability=0.5*)

Cutout augmentation.

References

- Kostrikov et al., Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels.

Parameters **probability** (*float*) – probability to cutout.

probability

probability to cutout.

Type `float`

Methods

get_params (*deep=False*)

Returns augmentation parameters.

Parameters **deep** (*bool*) – flag to deeply copy objects.

Returns augmentation parameters.

Return type `dict`

get_type()

Returns augmentation type.

Returns augmentation type.

Return type `str`

transform(x)

Returns observation performed Cutout.

Parameters **x** (`torch.Tensor`) – observation tensor.

Returns processed observation tensor.

Return type `torch.Tensor`

Attributes

TYPE = `'cutout'`

d3rlpy.augmentation.image.HorizontalFlip

class d3rlpy.augmentation.image.**HorizontalFlip**(*probability=0.1*)

Horizontal flip augmentation.

References

- [Kostrikov et al., Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels.](#)

Parameters **probability** (`float`) – probability to flip horizontally.

probability

probability to flip horizontally.

Type `float`

Methods

get_params(*deep=False*)

Returns augmentation parameters.

Parameters **deep** (`bool`) – flag to deeply copy objects.

Returns augmentation parameters.

Return type `dict`

get_type()

Returns augmentation type.

Returns augmentation type.

Return type `str`

transform (*x*)

Returns horizontally flipped image.

Parameters *x* (*torch.Tensor*) – observation tensor.

Returns processed observation tensor.

Return type *torch.Tensor*

Attributes

TYPE = 'horizontal_flip'

d3rlpy.augmentation.image.VerticalFlip

class d3rlpy.augmentation.image.**VerticalFlip** (*probability=0.1*)

Vertical flip augmentation.

References

- Kostrikov et al., Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels.

Parameters *probability* (*float*) – probability to flip vertically.

probability

probability to flip vertically.

Type *float*

Methods

get_params (*deep=False*)

Returns augmentation parameters.

Parameters *deep* (*bool*) – flag to deeply copy objects.

Returns augmentation parameters.

Return type *dict*

get_type ()

Returns augmentation type.

Returns augmentation type.

Return type *str*

transform (*x*)

Returns vertically flipped image.

Parameters *x* (*torch.Tensor*) – observation tensor.

Returns processed observation tensor.

Return type *torch.Tensor*

Attributes

TYPE = 'vertical_flip'

d3rlpy.augmentation.image.RandomRotation

class d3rlpy.augmentation.image.**RandomRotation** (*degree=5.0*)
Random rotation augmentation.

References

- [Kostrikov et al., Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels.](#)

Parameters **degree** (*float*) – range of degrees to rotate image.

degree
range of degrees to rotate image.
Type *float*

Methods

get_params (*deep=False*)
Returns augmentation parameters.
Parameters **deep** (*bool*) – flag to deeply copy objects.
Returns augmentation parameters.
Return type *dict*

get_type ()
Returns augmentation type.
Returns augmentation type.
Return type *str*

transform (*x*)
Returns rotated image.
Parameters **x** (*torch.Tensor*) – observation tensor.
Returns processed observation tensor.
Return type *torch.Tensor*

Attributes

TYPE = 'random_rotation'

d3rlpy.augmentation.image.Intensity

class d3rlpy.augmentation.image.**Intensity** (*scale=0.1*)
Intensity augmentation.

$$x' = x + n$$

where $n \sim N(0, scale)$.

References

- Kostrikov et al., Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels.

Parameters *scale* (*float*) – scale of multiplier.

scale
scale of multiplier.

Type *float*

Methods

get_params (*deep=False*)
Returns augmentation parameters.

Parameters *deep* (*bool*) – flag to deeply copy objects.

Returns augmentation parameters.

Return type *dict*

get_type ()
Returns augmentation type.

Returns augmentation type.

Return type *str*

transform (*x*)
Returns multiplied image.

Parameters *x* (*torch.Tensor*) – observation tensor.

Returns processed observation tensor.

Return type *torch.Tensor*

Attributes

TYPE = *'intensity'*

d3rlpy.augmentation.image.ColorJitter

class d3rlpy.augmentation.image.ColorJitter (*brightness=(0.6, 1.4), contrast=(0.6, 1.4), saturation=(0.6, 1.4), hue=(-0.5, 0.5)*)

Color Jitter augmentation.

This augmentation modifies the given images in the HSV channel spaces as well as a contrast change. This augmentation will be useful with the real world images.

References

- [Laskin et al., Reinforcement Learning with Augmented Data.](#)

Parameters

- **brightness** (*tuple*) – brightness scale range.
- **contrast** (*tuple*) – contrast scale range.
- **saturation** (*tuple*) – saturation scale range.
- **hue** (*tuple*) – hue scale range.

brightness

brightness scale range.

Type *tuple*

contrast

contrast scale range.

Type *tuple*

saturation

saturation scale range.

Type *tuple*

hue

hue scale range.

Type *tuple*

Methods

get_params (*deep=False*)

Returns augmentation parameters.

Parameters **deep** (*bool*) – flag to deeply copy objects.

Returns augmentation parameters.

Return type *dict*

get_type ()

Returns augmentation type.

Returns augmentation type.

Return type *str*

transform (*x*)

Returns jittered images.

Parameters *x* (*torch.Tensor*) – observation tensor.

Returns processed observation tensor.

Return type *torch.Tensor*

Attributes

TYPE = 'color_jitter'

3.8.2 Vector Observation

<i>d3rlpy.augmentation.vector.SingleAmplitudeScaling</i>	Single Amplitude Scaling augmentation.
<i>d3rlpy.augmentation.vector.MultipleAmplitudeScaling</i>	Multiple Amplitude Scaling augmentation.

d3rlpy.augmentation.vector.SingleAmplitudeScaling

class d3rlpy.augmentation.vector.**SingleAmplitudeScaling** (*minimum=0.8, maximum=1.2*) *maxi-*

Single Amplitude Scaling augmentation.

$$x' = x + z$$

where $z \sim \text{Unif}(\text{minimum}, \text{maximum})$.

References

- Laskin et al., Reinforcement Learning with Augmented Data.

Parameters

- **minimum** (*float*) – minimum amplitude scale.
- **maximum** (*float*) – maximum amplitude scale.

minimum

minimum amplitude scale.

Type *float*

maximum

maximum amplitude scale.

Type *float*

Methods

get_params (*deep=False*)

Returns augmentation parameters.

Parameters `deep` (*bool*) – flag to deeply copy objects.

Returns augmentation parameters.

Return type `dict`

get_type()

Returns augmentation type.

Returns augmentation type.

Return type `str`

transform(*x*)

Returns scaled observation.

Parameters `x` (*torch.Tensor*) – observation tensor.

Returns processed observation tensor.

Return type `torch.Tensor`

Attributes

`TYPE = 'single_amplitude_scaling'`

d3rlpy.augmentation.vector.MultipleAmplitudeScaling

class d3rlpy.augmentation.vector.**MultipleAmplitudeScaling**(*minimum=0.8, maximum=1.2*)

Multiple Amplitude Scaling augmentation.

$$x' = x + z$$

where $z \sim \text{Unif}(\text{minimum}, \text{maximum})$ and z is a vector with different amplitude scale on each.

References

- [Laskin et al., Reinforcement Learning with Augmented Data.](#)

Parameters

- **minimum** (*float*) – minimum amplitude scale.
- **maximum** (*float*) – maximum amplitude scale.

minimum

minimum amplitude scale.

Type `float`

maximum

maximum amplitude scale.

Type `float`

Methods

get_params (*deep=False*)

Returns augmentation parameters.

Parameters *deep* (*bool*) – flag to deeply copy objects.

Returns augmentation parameters.

Return type *dict*

get_type ()

Returns augmentation type.

Returns augmentation type.

Return type *str*

transform (*x*)

Returns scaled observation.

Parameters *x* (*torch.Tensor*) – observation tensor.

Returns processed observation tensor.

Return type *torch.Tensor*

Attributes

TYPE = 'multiple_amplitude_scaling'

3.9 Metrics

d3rlpy provides scoring functions without compromising scikit-learn compatibility. You can evaluate many metrics with test episodes during training.

```
from d3rlpy.datasets import get_cartpole
from d3rlpy.algos import DQN
from d3rlpy.metrics.scorer import td_error_scorer
from d3rlpy.metrics.scorer import average_value_estimation_scorer
from d3rlpy.metrics.scorer import evaluate_on_environment
from sklearn.model_selection import train_test_split

dataset, env = get_cartpole()

train_episodes, test_episodes = train_test_split(dataset)

dqn = DQN()

dqn.fit(train_episodes,
        eval_episodes=test_episodes,
        scorers={
            'td_error': td_error_scorer,
            'value_scale': average_value_estimation_scorer,
            'environment': evaluate_on_environment(env)
        })
```

You can also use them with scikit-learn utilities.

```

from sklearn.model_selection import cross_validate

scores = cross_validate(dqn,
                        dataset,
                        scoring={
                            'td_error': td_error_scorer,
                            'environment': evaluate_on_environment(env)
                        })

```

3.9.1 Algorithms

<code>d3rlpy.metrics.scorer.td_error_scorer</code>	Returns average TD error (in negative scale).
<code>d3rlpy.metrics.scorer.discounted_sum_of_advantage_scorer</code>	Returns average of discounted sum of advantage (in negative scale).
<code>d3rlpy.metrics.scorer.average_value_estimation_scorer</code>	Returns average value estimation (in negative scale).
<code>d3rlpy.metrics.scorer.value_estimation_std_scorer</code>	Returns standard deviation of value estimation (in negative scale).
<code>d3rlpy.metrics.scorer.initial_state_value_estimation_scorer</code>	Returns mean estimated action-values at the initial states.
<code>d3rlpy.metrics.scorer.soft_opc_scorer</code>	Returns Soft Off-Policy Classification metrics.
<code>d3rlpy.metrics.scorer.continuous_action_diff_scorer</code>	Returns squared difference of actions between algorithm and dataset.
<code>d3rlpy.metrics.scorer.discrete_action_match_scorer</code>	Returns percentage of identical actions between algorithm and dataset.
<code>d3rlpy.metrics.scorer.evaluate_on_environment</code>	Returns scorer function of evaluation on environment.
<code>d3rlpy.metrics.comparer.compare_continuous_action_diff</code>	Returns scorer function of action difference between algorithms.
<code>d3rlpy.metrics.comparer.compare_discrete_action_match</code>	Returns scorer function of action matches between algorithms.

d3rlpy.metrics.scorer.td_error_scorer

`d3rlpy.metrics.scorer.td_error_scorer` (*algo*, *episodes*, *window_size*=1024)

Returns average TD error (in negative scale).

This metrics suggests how Q functions overfit to training sets. If the TD error is large, the Q functions are overfitting.

$$\mathbb{E}_{s_t, a_t, r_{t+1}, s_{t+1} \sim D} [Q_{\theta}(s_t, a_t) - (r_t + \gamma \max_a Q_{\theta}(s_{t+1}, a))^2]$$

Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **episodes** (`list` (`d3rlpy.dataset.Episode`)) – list of episodes.
- **window_size** (`int`) – mini-batch size to compute.

Returns negative average TD error.

Return type `float`

d3rlpy.metrics.scorer.discounted_sum_of_advantage_scorer

`d3rlpy.metrics.scorer.discounted_sum_of_advantage_scorer` (*algo*, *episodes*, *window_size=1024*)

Returns average of discounted sum of advantage (in negative scale).

This metrics suggests how the greedy-policy selects different actions in action-value space. If the sum of advantage is small, the policy selects actions with larger estimated action-values.

$$\mathbb{E}_{s_t, a_t \sim D} \left[\sum_{t'=t} \gamma^{t'-t} A(s_{t'}, a_{t'}) \right]$$

where $A(s_t, a_t) = Q_\theta(s_t, a_t) - \max_a Q_\theta(s_t, a)$.

References

- [Murphy., A generalization error for Q-Learning.](#)

Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **episodes** (`list` (`d3rlpy.dataset.Episode`)) – list of episodes.
- **window_size** (`int`) – mini-batch size to compute.

Returns negative average of discounted sum of advantage.

Return type `float`

d3rlpy.metrics.scorer.average_value_estimation_scorer

`d3rlpy.metrics.scorer.average_value_estimation_scorer` (*algo*, *episodes*, *window_size=1024*)

Returns average value estimation (in negative scale).

This metrics suggests the scale for estimation of Q functions. If average value estimation is too large, the Q functions overestimate action-values, which possibly makes training failed.

$$\mathbb{E}_{s_t \sim D} [\max_a Q_\theta(s_t, a)]$$

Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **episodes** (`list` (`d3rlpy.dataset.Episode`)) – list of episodes.
- **window_size** (`int`) – mini-batch size to compute.

Returns negative average value estimation.

Return type `float`

d3rlpy.metrics.scorer.value_estimation_std_scorer

`d3rlpy.metrics.scorer.value_estimation_std_scorer` (*algo*, *episodes*, *window_size=1024*)

Returns standard deviation of value estimation (in negative scale).

This metrics suggests how confident Q functions are for the given episodes. This metrics will be more accurate with *bootstrap* enabled and the larger *n_critics* at algorithm. If standard deviation of value estimation is large, the Q functions are overfitting to the training set.

$$\mathbb{E}_{s_t \sim D, a \sim \arg\max_a Q_\theta(s_t, a)} [Q_{\text{std}}(s_t, a)]$$

where $Q_{\text{std}}(s, a)$ is a standard deviation of action-value estimation over ensemble functions.

Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **episodes** (`list(d3rlpy.dataset.Episode)`) – list of episodes.
- **window_size** (`int`) – mini-batch size to compute.

Returns negative standard deviation.

Return type `float`

d3rlpy.metrics.scorer.initial_state_value_estimation_scorer

`d3rlpy.metrics.scorer.initial_state_value_estimation_scorer` (*algo, episodes, window_size=1024*)

Returns mean estimated action-values at the initial states.

This metrics suggests how much return the trained policy would get from the initial states by deploying the policy to the states. If the estimated value is large, the trained policy is expected to get higher returns.

$$\mathbb{E}_{s_0 \sim D} [Q(s_0, \pi(s_0))]$$

References

- Paine et al., Hyperparameter Selection for Offline Reinforcement Learning

Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **episodes** (`list(d3rlpy.dataset.Episode)`) – list of episodes.
- **window_size** (`int`) – mini-batch size to compute.

Returns mean action-value estimation at the initial states.

Return type `float`

d3rlpy.metrics.scorer.soft_opc_scorer

`d3rlpy.metrics.scorer.soft_opc_scorer` (*return_threshold*)

Returns Soft Off-Policy Classification metrics.

This function returns scorer function, which is suitable to the standard scikit-learn scorer function style. The metrics of the scorer function is evaluating gaps of action-value estimation between the success episodes and the all episodes. If the learned Q-function is optimal, action-values in success episodes are expected to be higher than the others. The success episode is defined as an episode with a return above the given threshold.

$$\mathbb{E}_{s, a \sim D_{\text{success}}} [Q(s, a)] - \mathbb{E}_{s, a \sim D} [Q(s, a)]$$


```

from d3rlpy.datasets import get_cartpole
from d3rlpy.algos import DQN
from d3rlpy.metrics.scorer import soft_opc_scorer
from sklearn.model_selection import train_test_split

dataset, _ = get_cartpole()
train_episodes, test_episodes = train_test_split(dataset, test_size=0.2)

scorer = soft_opc_scorer(return_threshold=180)

dqn = DQN()
dqn.fit(train_episodes,
        eval_episodes=test_episodes,
        scorers={'soft_opc': scorer})

```

References

- Irpan et al., Off-Policy Evaluation via Off-Policy Classification.

Parameters `return_threshold` (*float*) – threshold of success episodes.

Returns scorer function.

Return type callable

d3rlpy.metrics.scorer.continuous_action_diff_scorer

`d3rlpy.metrics.scorer.continuous_action_diff_scorer` (*algo*, *episodes*, *window_size=1024*)

Returns squared difference of actions between algorithm and dataset.

This metrics suggests how different the greedy-policy is from the given episodes in continuous action-space. If the given episodes are near-optimal, the small action difference would be better.

$$\mathbb{E}_{s_t, a_t \sim D} [(a_t - \pi_\phi(s_t))^2]$$

Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes.
- **window_size** (*int*) – mini-batch size to compute.

Returns negative squared action difference.

Return type *float*

d3rlpy.metrics.scorer.discrete_action_match_scorer

`d3rlpy.metrics.scorer.discrete_action_match_scorer` (*algo*, *episodes*, *window_size=1024*)

Returns percentage of identical actions between algorithm and dataset.

This metrics suggests how different the greedy-policy is from the given episodes in discrete action-space. If the given episodes are near-optimal, the large percentage would be better.

$$\frac{1}{N} \sum^N \mathbb{I} \{a_t = \operatorname{argmax}_a Q_\theta(s_t, a)\}$$

Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **episodes** (`list(d3rlpy.dataset.Episode)`) – list of episodes.
- **window_size** (`int`) – mini-batch size to compute.

Returns percentage of identical actions.

Return type `float`

d3rlpy.metrics.scorer.evaluate_on_environment

`d3rlpy.metrics.scorer.evaluate_on_environment(env, n_trials=10, epsilon=0.0, render=False)`

Returns scorer function of evaluation on environment.

This function returns scorer function, which is suitable to the standard scikit-learn scorer function style. The metrics of the scorer function is ideal metrics to evaluate the resulted policies.

```
import gym

from d3rlpy.algos import DQN
from d3rlpy.metrics.scorer import evaluate_on_environment

env = gym.make('CartPole-v0')

scorer = evaluate_on_environment(env)

cql = CQL()

mean_episode_return = scorer(cql)
```

Parameters

- **env** (`gym.Env`) – gym-styled environment.
- **n_trials** (`int`) – the number of trials.
- **epsilon** (`float`) – noise factor for epsilon-greedy policy.
- **render** (`bool`) – flag to render environment.

Returns scorer function.

Return type callable

d3rlpy.metrics.comparer.compare_continuous_action_diff

`d3rlpy.metrics.comparer.compare_continuous_action_diff(base_algo, win-
dow_size=1024)`

Returns scorer function of action difference between algorithms.

This metrics suggests how different the two algorithms are in continuous action-space. If the algorithm to compare with is near-optimal, the small action difference would be better.

$$\mathbb{E}_{s_t \sim D}[(\pi_{\phi_1}(s_t) - \pi_{\phi_2}(s_t))^2]$$

```

from d3rlpy.algos import CQL
from d3rlpy.metrics.comparer import compare_continuous_action_diff

cql1 = CQL()
cql2 = CQL()

scorer = compare_continuous_action_diff(cql1)

squared_action_diff = scorer(cql2, ...)

```

Parameters

- **base_algo** (*d3rlpy.algos.base.AlgoBase*) – algorithm to compare with.
- **window_size** (*int*) – mini-batch size to compute.

Returns scorer function.

Return type callable

d3rlpy.metrics.comparer.compare_discrete_action_match

`d3rlpy.metrics.comparer.compare_discrete_action_match` (*base_algo*, *window_size=1024*)

Returns scorer function of action matches between algorithms.

This metrics suggests how different the two algorithms are in discrete action-space. If the algorithm to compare with is near-optimal, the small action difference would be better.

$$\mathbb{E}_{s_t \sim D} [\| \{ \operatorname{argmax}_a Q_{\theta_1}(s_t, a) = \operatorname{argmax}_a Q_{\theta_2}(s_t, a) \} \|]$$

```

from d3rlpy.algos import DQN
from d3rlpy.metrics.comparer import compare_continuous_action_diff

dqn1 = DQN()
dqn2 = DQN()

scorer = compare_continuous_action_diff(dqn1)

percentage_of_identical_actions = scorer(dqn2, ...)

```

Parameters

- **base_algo** (*d3rlpy.algos.base.AlgoBase*) – algorithm to compare with.
- **window_size** (*int*) – mini-batch size to compute.

Returns scorer function.

Return type callable

3.9.2 Dynamics

<code>d3rlpy.metrics.scorer.dynamics_observation_prediction_error_scorer</code>	Returns MSE of observation prediction (in negative scale).
<code>d3rlpy.metrics.scorer.dynamics_reward_prediction_error_scorer</code>	Returns MSE of reward prediction (in negative scale).
<code>d3rlpy.metrics.scorer.dynamics_prediction_variance_scorer</code>	Returns prediction variance of ensemble dynamics (in negative scale).

`d3rlpy.metrics.scorer.dynamics_observation_prediction_error_scorer`

`d3rlpy.metrics.scorer.dynamics_observation_prediction_error_scorer` (*dynamics*,
episodes,
win-
dow_size=1024)

Returns MSE of observation prediction (in negative scale).

This metrics suggests how dynamics model is generalized to test sets. If the MSE is large, the dynamics model are overfitting.

$$\mathbb{E}_{s_t, a_t, s_{t+1} \sim D} [(s_{t+1} - s')^2]$$

where $s' \sim T(s_t, a_t)$.

Parameters

- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model.
- **episodes** (`list` (`d3rlpy.dataset.Episode`)) – list of episodes.
- **window_size** (`int`) – mini-batch size to compute.

Returns negative mean squared error.

Return type `float`

`d3rlpy.metrics.scorer.dynamics_reward_prediction_error_scorer`

`d3rlpy.metrics.scorer.dynamics_reward_prediction_error_scorer` (*dynamics*,
episodes, *win-*
dow_size=1024)

Returns MSE of reward prediction (in negative scale).

This metrics suggests how dynamics model is generalized to test sets. If the MSE is large, the dynamics model are overfitting.

$$\mathbb{E}_{s_t, a_t, r_{t+1} \sim D} [(r_{t+1} - r')^2]$$

where $r' \sim T(s_t, a_t)$.

Parameters

- **dynamics** (`d3rlpy.dynamics.base.DynamicsBase`) – dynamics model.
- **episodes** (`list` (`d3rlpy.dataset.Episode`)) – list of episodes.
- **window_size** (`int`) – mini-batch size to compute.

Returns negative mean squared error.

Return type `float`

d3rlpy.metrics.scorer.dynamics_prediction_variance_scorer

`d3rlpy.metrics.scorer.dynamics_prediction_variance_scorer` (*dynamics*, *episodes*,
window_size=1024)

Returns prediction variance of ensemble dynamics (in negative scale).

This metrics suggests how dynamics model is confident of test sets. If the variance is large, the dynamics model has large uncertainty.

Parameters

- **dynamics** (*d3rlpy.dynamics.base.DynamicsBase*) – dynamics model.
- **episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes.
- **window_size** (*int*) – mini-batch size to compute.

Returns negative variance.

Return type `float`

3.10 Save and Load

3.10.1 save_model and load_model

```
from d3rlpy.datasets import get_cartpole
from d3rlpy.algos import DQN

dataset, env = get_cartpole()

dqn = DQN()
dqn.fit(dataset.episodes, n_epochs=1)

# save entire model parameters.
dqn.save_model('model.pt')

# load entire model parameters.
dqn.load_model('model.pt')
```

`save_model` method saves all parameters including optimizer states, which is useful when checking all the outputs or re-training from snapshots.

3.10.2 from_json

It is very boring to set the same hyperparameters to initialize algorithms when loading model parameters. In d3rlpy, `params.json` is saved at the beginning of `fit` method, which includes all hyperparameters within the algorithm object. You can recreate algorithm objects from `params.json` via `from_json` method.

```
from d3rlpy.algos import DQN

dqn = DQN.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
dqn.load_model('model.pt')
```

3.10.3 save_policy

`save_policy` method saves the only greedy-policy computation graph as TorchScript or ONNX. When `save_policy` method is called, the greedy-policy graph is constructed and traced via `torch.jit.trace` function.

```
from d3rlpy.datasets import get_cartpole
from d3rlpy.algos import DQN

dataset, env = get_cartpole()

dqn = DQN()
dqn.fit(dataset.episodes, n_epochs=1)

# save greedy-policy as TorchScript
dqn.save_policy('policy.pt')

# save greedy-policy as ONNX
dqn.save_policy('policy.onnx', as_onnx=True)
```

TorchScript

TorchScript is a optimizable graph expression provided by PyTorch. The saved policy can be loaded without any dependencies except PyTorch.

```
import torch

# load greedy-policy only with PyTorch
policy = torch.jit.load('policy.pt')

# returns greedy actions
actions = policy(torch.rand(32, 6))
```

This is especially useful when deploying the trained models to productions. The computation can be faster and you don't need to install d3rlpy. Moreover, TorchScript model can be easily loaded even with C++, which will empower your robotics and embedding system projects.

```
#include <torch/script.h>

int main(int argc, char* argv[]) {
    torch::jit::script::Module module;
    try {
        module = torch::jit::load("policy.pt")
    } catch (const c10::Error& e) {
        return -1;
    }
    return 0;
}
```

You can get more information about TorchScript [here](#).

ONNX

ONNX is an open format built to represent machine learning models. This is also useful when deploying the trained model to productions with various programming languages including Python, C++, JavaScript and more.

The following example is written with `onnxruntime`.

```
import onnxruntime as ort

# load ONNX policy via onnxruntime
ort_session = ort.InferenceSession('policy.onnx')

# observation
observation = np.random.rand(1, 6).astype(np.float32)

# returns greedy action
action = ort_session.run(None, {'input_0': observation})[0]
```

You can get more information about ONNX [here](#).

3.11 Logging

d3rlpy algorithms automatically save model parameters and metrics under *d3rlpy_logs* directory.

```
from d3rlpy.datasets import get_cartpole
from d3rlpy.algos import DQN

dataset, env = get_cartpole()

dqn = DQN()

# metrics and parameters are saved in `d3rlpy_logs/DQN_YYYYMMDDHHmmss`
dqn.fit(dataset.episodes)
```

You can designate the directory.

```
# the directory will be `custom_logs/custom_YYYYMMDDHHmmss`
dqn.fit(dataset.episodes, logdir='custom_logs', experiment_name='custom')
```

If you want to disable all loggings, you can pass *save_metrics=False*.

```
dqn.fit(dataset.episodes, save_metrics=False)
```

3.11.1 TensorBoard

The same information is also automatically saved for tensorboard under *runs* directory. You can interactively visualize training metrics easily.

```
$ pip install tensorboard
$ tensorboard --logdir runs
```

This tensorboard logs can be disabled by passing *tensorboard=False*.

```
dqn.fit(dataset.episodes, tensorboard=False)
```

3.12 scikit-learn compatibility

d3rlpy provides complete scikit-learn compatible APIs.

3.12.1 train_test_split

`d3rlpy.dataset.MDPDataset` is compatible with splitting functions in scikit-learn.

```
from d3rlpy.algos import DQN
from d3rlpy.datasets import get_cartpole
from d3rlpy.metrics.scorer import td_error_scorer
from sklearn.model_selection import train_test_split

dataset, env = get_cartpole()

train_episodes, test_episodes = train_test_split(dataset, test_size=0.2)

dqn = DQN()
dqn.fit(train_episodes,
        eval_episodes=test_episodes,
        n_epochs=1,
        scorers={'td_error': td_error_scorer})
```

3.12.2 cross_validate

cross validation is also easily performed.

```
from d3rlpy.algos import DQN
from d3rlpy.datasets import get_cartpole
from d3rlpy.metrics import td_error_scorer
from sklearn.model_selection import cross_validate

dataset, env = get_cartpole()

dqn = DQN()

scores = cross_validate(dqn,
                        dataset,
                        scoring={'td_error': td_error_scorer},
                        fit_params={'n_epochs': 1})
```

3.12.3 GridSearchCV

You can also perform grid search to find good hyperparameters.

```
from d3rlpy.algos import DQN
from d3rlpy.datasets import get_cartpole
from d3rlpy.metrics import td_error_scorer
from sklearn.model_selection import GridSearchCV

dataset, env = get_cartpole()

dqn = DQN()

gscv = GridSearchCV(estimator=dqn,
                    param_grid={'learning_rate': [1e-4, 3e-4, 1e-3]},
                    scoring={'td_error': td_error_scorer},
                    refit=False)
```

(continues on next page)

(continued from previous page)

```
gscv.fit(dataset.episodes, n_epochs=1)
```

3.12.4 parallel execution with multiple GPUs

Some scikit-learn utilities provide `n_jobs` option, which enable fitting process to run in parallel to boost productivity. Ideally, if you have multiple GPUs, the multiple processes use different GPUs for computational efficiency.

d3rlpy provides special device assignment mechanism to realize this.

```
from d3rlpy.algos import DQN
from d3rlpy.datasets import get_cartpole
from d3rlpy.metrics import td_error_scorer
from d3rlpy.context import parallel
from sklearn.model_selection import cross_validate

dataset, env = get_cartpole()

# enable GPU
dqn = DQN(use_gpu=True)

# automatically assign different GPUs for the 4 processes.
with parallel():
    scores = cross_validate(dqn,
                           dataset,
                           scoring={'td_error': td_error_scorer},
                           fit_params={'n_epochs': 1},
                           n_jobs=4)
```

If `use_gpu=True` is passed, d3rlpy internally manages GPU device id via `d3rlpy.gpu.Device` object. This object is designed for scikit-learn's multi-process implementation that makes deep copies of the estimator object before dispatching. The `Device` object will increment its device id when deeply copied under the parallel context.

```
import copy
from d3rlpy.context import parallel
from d3rlpy.gpu import Device

device = Device(0)
# device.get_id() == 0

new_device = copy.deepcopy(device)
# new_device.get_id() == 0

with parallel():
    new_device = copy.deepcopy(device)
    # new_device.get_id() == 1
    # device.get_id() == 1

    new_device = copy.deepcopy(device)
    # if you have only 2 GPUs, it goes back to 0.
    # new_device.get_id() == 0
    # device.get_id() == 0

from d3rlpy.algos import DQN
```

(continues on next page)

```
dqn = DQN(use_gpu=Device(0)) # assign id=0
dqn = DQN(use_gpu=Device(1)) # assign id=1
```

3.13 Online Training

d3rlpy provides not only offline training, but also online training utilities. Despite being designed for offline training algorithms, d3rlpy is flexible enough to be trained in an online manner with a few more utilities.

```
import gym

from d3rlpy.algos import DQN
from d3rlpy.online.buffers import ReplayBuffer
from d3rlpy.online.explorers import LinearDecayEpsilonGreedy

# setup environment
env = gym.make('CartPole-v0')
eval_env = gym.make('CartPole-v0')

# setup algorithm
dqn = DQN(batch_size=32,
          learning_rate=2.5e-4,
          target_update_interval=100,
          use_gpu=True)

# setup replay buffer
buffer = ReplayBuffer(maxlen=1000000, env=env)

# setup explorers
explorer = LinearDecayEpsilonGreedy(start_epsilon=1.0,
                                    end_epsilon=0.1,
                                    duration=10000)

# start training
dqn.fit_online(env,
              buffer,
              explorer=explorer, # you don't need this with probablistic policy_
↳ algorithms
              eval_env=eval_env,
              n_epochs=30,
              n_steps_per_epoch=1000,
              n_updates_per_epoch=100)
```

3.13.1 Replay Buffer

`d3rlpy.online.buffers.ReplayBuffer`

Standard Replay Buffer.

d3rlpy.online.buffers.ReplayBuffer

class d3rlpy.online.buffers.**ReplayBuffer** (*maxlen, env, episodes=None*)
 Standard Replay Buffer.

Parameters

- **maxlen** (*int*) – the maximum number of data length.
- **env** (*gym.Env*) – gym-like environment to extract shape information.
- **episodes** (*list* (*d3rlpy.dataset.Episode*)) – list of episodes to initialize buffer

prev_observation

previously appended observation.

Type *numpy.ndarray*

prev_action

previously appended action.

Type *numpy.ndarray* or *int*

prev_reward

previously appended reward.

Type *float*

prev_transition

previously appended transition.

Type *d3rlpy.dataset.Transition*

transitions

queue of transitions.

Type *d3rlpy.online.buffers.TransitionQueue*

observation_shape

observation shape.

Type *tuple*

action_size

action size.

Type *int*

Methods

__len__ ()

append (*observation, action, reward, terminal*)

Append observation, action, reward and terminal flag to buffer.

If the terminal flag is True, Monte-Carlo returns will be computed with an entire episode and the whole transitions will be appended.

Parameters

- **observation** (*numpy.ndarray*) – observation.
- **action** (*numpy.ndarray* or *int*) – action.
- **reward** (*float*) – reward.
- **terminal** (*bool* or *float*) – terminal flag.

append_episode (*episode*)

Append Episode object to buffer.

Parameters **episode** (*d3rlpy.dataset.Episode*) – episode.

sample (*batch_size*, *n_frames*=1)

Returns sampled mini-batch of transitions.

If observation is image, you can stack arbitrary frames via *n_frames*.

```
buffer.observation_shape == (3, 84, 84)

# stack 4 frames
batch = buffer.sample(batch_size=32, n_frames=4)

batch.observations.shape == (32, 12, 84, 84)
```

Parameters

- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.

Returns mini-batch.

Return type *d3rlpy.dataset.TransitionMiniBatch*

size ()

Returns the number of appended elements in buffer.

Returns the number of elements in buffer.

Return type *int*

3.13.2 Explorers

<i>d3rlpy.online.explorers.</i> <i>LinearDecayEpsilonGreedy</i>	ϵ -greedy explorer with linear decay schedule.
<i>d3rlpy.online.explorers.NormalNoise</i>	Normal noise explorer.

d3rlpy.online.explorers.LinearDecayEpsilonGreedy

class *d3rlpy.online.explorers.LinearDecayEpsilonGreedy* (*start_epsilon*=1.0,
end_epsilon=0.1, *duration*=1000000)

ϵ -greedy explorer with linear decay schedule.

Parameters

- **start_epsilon** (*float*) – the beginning ϵ .
- **end_epsilon** (*float*) – the end ϵ .
- **duration** (*int*) – the scheduling duration.

start_epsilon

the beginning ϵ .

Type *float*

end_epsilon

the end ϵ .

Type *float*

duration
the scheduling duration.
Type `int`

Methods

compute_epsilon (*step*)
Returns decayed ϵ .

Returns ϵ .

Return type `float`

sample (*algo*, *x*, *step*)
Returns ϵ -greedy action.

Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **x** (`numpy.ndarray`) – observation.
- **step** (`int`) – current environment step.

Returns ϵ -greedy action.

Return type `int`

d3rlpy.online.explorers.NormalNoise

class `d3rlpy.online.explorers.NormalNoise` (*mean=0.0*, *std=0.1*)
Normal noise explorer.

Parameters

- **mean** (`float`) – mean.
- **std** (`float`) – standard deviation.

mean
mean.

Type `float`

std
standard deviation.

Type `float`

Methods

sample (*algo*, *x*, **args*)
Returns action with noise injection.

Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **x** (`numpy.ndarray`) – observation.

Returns action with noise injection.

Return type `numpy.ndarray`

3.13.3 Iterators

`d3rlpy.online.iterators.train`

Start training loop of online deep reinforcement learning.

`d3rlpy.online.iterators.train`

```
d3rlpy.online.iterators.train(env, algo, buffer, explorer=None, n_steps=1000000,
                              n_steps_per_epoch=10000, update_interval=1, update_start_step=0,
                              eval_env=None, eval_epsilon=0.05, save_metrics=True,
                              experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs',
                              verbose=True, show_progress=True, tensorboard=True)
```

Start training loop of online deep reinforcement learning.

Parameters

- **env** (`gym.Env`) – gym-like environment.
- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm.
- **buffer** (`d3rlpy.online.buffers.Buffer`) – replay buffer.
- **explorer** (`d3rlpy.online.explorers.Explorer`) – action explorer.
- **n_steps** (`int`) – the number of total steps to train.
- **n_steps_per_epoch** (`int`) – the number of steps per epoch.
- **update_interval** (`int`) – the number of steps per update.
- **update_start_step** (`int`) – the steps before starting updates.
- **eval_env** (`gym.Env`) – gym-like environment. If `None`, evaluation is skipped.
- **eval_epsilon** (`float`) – ϵ -greedy factor during evaluation.
- **save_metrics** (`bool`) – flag to record metrics. If `False`, the log directory is not created and the model parameters are not saved.
- **experiment_name** (`str`) – experiment name for logging. If not passed, the directory name will be `{class name}_online_{timestamp}`.
- **with_timestamp** (`bool`) – flag to add timestamp string to the last of directory name.
- **logdir** (`str`) – root directory name to save logs.
- **verbose** (`bool`) – flag to show logged information on stdout.
- **show_progress** (`bool`) – flag to show progress bar for iterations.
- **tensorboard** (`bool`) – flag to save logged information in tensorboard (additional to the csv data)

3.14 Model-based Data Augmentation

d3rlpy provides model-based reinforcement learning algorithms. In d3rlpy, model-based algorithms are viewed as data augmentation techniques, which can boost performance potentially beyond the model-free algorithms.

```
from d3rlpy.datasets import get_pendulum
from d3rlpy.dynamics import MOPO
from d3rlpy.metrics.scorer import dynamics_observation_prediction_error_scorer
from d3rlpy.metrics.scorer import dynamics_reward_prediction_error_scorer
from d3rlpy.metrics.scorer import dynamics_prediction_variance_scorer
from sklearn.model_selection import train_test_split

dataset, _ = get_pendulum()

train_episodes, test_episodes = train_test_split(dataset)

mopo = MOPO(learning_rate=1e-4, use_gpu=True)

# same as algorithms
mopo.fit(train_episodes,
        eval_episodes=test_episodes,
        n_epochs=100,
        scorers={
            'observation_error': dynamics_observation_prediction_error_scorer,
            'reward_error': dynamics_reward_prediction_error_scorer,
            'variance': dynamics_prediction_variance_scorer,
        })
```

Pick the best model based on evaluation metrics.

```
from d3rlpy.dynamics import MOPO
from d3rlpy.algos import CQL

# load trained dynamics model
mopo = MOPO.from_json('<path-to-params.json>/params.json')
mopo.load_model('<path-to-model>/model_xx.pt')
mopo.n_transitions = 400 # tunable parameter
mopo.horizon = 5 # tunable parameter
mopo.lam = 1.0 # tunable parameter

# give mopo as dynamics argument.
cql = CQL(dynamics=mopo)
```

If you pass a dynamics model to algorithms, new transitions are generated at the beginning of every epoch.

d3rlpy.dynamics.mopo.MOPO

Model-based Offline Policy Optimization.

3.14.1 d3rlpy.dynamics.mopo.MOPO

```
class d3rlpy.dynamics.mopo.MOPO(*,
                                learning_rate=0.001,
                                op-
                                tim_factory=<d3rlpy.optimizers.AdamFactory object>,
                                encoder_factory='default', batch_size=100, n_frames=1,
                                n_ensembles=5, n_transitions=400, horizon=5, lam=1.0,
                                discrete_action=False, scaler=None, use_gpu=False,
                                impl=None, **kwargs)
```

Model-based Offline Policy Optimization.

MOPO is a model-based RL approach for offline policy optimization. MOPO leverages the probabilistic ensemble dynamics model to generate new dynamics data with uncertainty penalties.

The ensemble dynamics model consists of N probabilistic models $\{T_{\theta_i}\}_{i=1}^N$. At each epoch, new transitions are generated via randomly picked dynamics model T_{θ} .

$$s_{t+1}, r_{t+1} \sim T_{\theta}(s_t, a_t)$$

where $s_t \sim D$ for the first step, otherwise s_t is the previous generated observation, and $a_t \sim \pi(\cdot|s_t)$. The generated r_{t+1} would be far from the ground truth if the actions sampled from the policy function is out-of-distribution. Thus, the uncertainty penalty regularizes this bias.

$$r_{t+1}^{\sim} = r_{t+1} - \lambda \max_{i=1}^N \|\Sigma_i(s_t, a_t)\|$$

where $\Sigma(s_t, a_t)$ is the estimated variance.

Finally, the generated transitions $(s_t, a_t, r_{t+1}^{\sim}, s_{t+1})$ are appended to dataset D .

This generation process starts with randomly sampled $n_transitions$ transitions till $horizon$ steps.

Note: Currently, MOPO only supports vector observations.

References

- Yu et al., MOPO: Model-based Offline Policy Optimization.

Parameters

- **learning_rate** (*float*) – learning rate for dynamics model.
- **optim_factory** (*d3rlpy.optimizers.OptimizerFactory*) – optimizer factory.
- **encoder_factory** (*d3rlpy.encoders.EncoderFactory* or *str*) – encoder factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **n_ensembles** (*int*) – the number of dynamics model for ensemble.
- **n_transitions** (*int*) – the number of parallel trajectories to generate.
- **horizon** (*int*) – the number of steps to generate.
- **lam** (*float*) – λ for uncertainty penalties.
- **discrete_action** (*bool*) – flag to take discrete actions.

- **scaler** (*d3rlpy.preprocessing.scalers.Scaler* or *str*) – preprocessor. The available options are [*'pixel'*, *'min_max'*, *'standard'*].
- **use_gpu** (*bool* or *d3rlpy.gpu.Device*) – flag to use GPU or device.
- **impl** (*d3rlpy.dynamics.base.DynamicsImplBase*) – dynamics implementation.

learning_rate

learning rate for dynamics model.

Type *float*

optim_factory

optimizer factory.

Type *d3rlpy.optimizers.OptimizerFactory*

encoder_factory

encoder factory.

Type *d3rlpy.encoders.EncoderFactory*

batch_size

mini-batch size.

Type *int*

n_frames

the number of frames to stack for image observation.

Type *int*

n_ensembles

the number of dynamics model for ensemble.

Type *int*

n_transitions

the number of parallel trajectories to generate.

Type *int*

horizon

the number of steps to generate.

Type *int*

lam

λ for uncertainty penalties.

Type *float*

discrete_action

flag to take discrete actions.

Type *bool*

scaler

preprocessor.

Type *d3rlpy.preprocessing.scalers.Scaler*

use_gpu

flag to use GPU or device.

Type `d3rlpy.gpu.Device`

impl
dynamics implementation.

Type `d3rlpy.dynamics.base.DynamicsImplBase`

eval_results_
evaluation results.

Type `dict`

Methods

create_impl (*observation_shape*, *action_size*)
Instantiate implementation objects with the dataset shapes.

This method will be used internally when *fit* method is called.

Parameters

- **observation_shape** (*tuple*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

fit (*episodes*, *n_epochs*=1000, *save_metrics*=True, *experiment_name*=None, *with_timestamp*=True, *logdir*='d3rlpy_logs', *verbose*=True, *show_progress*=True, *tensorboard*=True, *eval_episodes*=None, *save_interval*=1, *scorers*=None)
Trains with the given dataset.

```
algo.fit(episodes)
```

Parameters

- **episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to train.
- **n_epochs** (*int*) – the number of epochs to train.
- **save_metrics** (*bool*) – flag to record metrics in files. If False, the log directory is not created and the model parameters are not saved during training.
- **experiment_name** (*str*) – experiment name for logging. If not passed, the directory name will be *{class name}_{timestamp}*.
- **with_timestamp** (*bool*) – flag to add timestamp string to the last of directory name.
- **logdir** (*str*) – root directory name to save logs.
- **verbose** (*bool*) – flag to show logged information on stdout.
- **show_progress** (*bool*) – flag to show progress bar for iterations.
- **tensorboard** (*bool*) – flag to save logged information in tensorboard (additional to the csv data)
- **eval_episodes** (*list* (`d3rlpy.dataset.Episode`)) – list of episodes to test.
- **save_interval** (*int*) – interval to save parameters.
- **scorers** (*list* (*callable*)) – list of scorer functions used with *eval_episodes*.

classmethod `from_json` (*fname*, *use_gpu=False*)

Returns algorithm configured with json file.

The Json file should be the one saved during fitting.

```
from d3rlpy.algos import Algo

# create algorithm with saved configuration
algo = Algo.from_json('d3rlpy_logs/<path-to-json>/params.json')

# ready to load
algo.load_model('d3rlpy_logs/<path-to-model>/model_100.pt')

# ready to predict
algo.predict(...)
```

Parameters

- **fname** (*str*) – file path to *params.json*.
- **use_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.

Returns algorithm.

Return type *d3rlpy.base.LearnableBase*

generate (*algo*, *transitions*)

Returns new transitions for data augmentation.

Parameters

- **algo** (*d3rlpy.algos.base.AlgoBase*) – algorithm.
- **transitions** (*list* (*d3rlpy.dataset.Transition*)) – list of transitions.

Returns list of generated transitions.

Return type *list*(*d3rlpy.dataset.Transition*)

get_params (*deep=True*)

Returns the all attributes.

This method returns the all attributes including ones in subclasses. Some of scikit-learn utilities will use this method.

```
params = algo.get_params(deep=True)

# the returned values can be used to instantiate the new object.
algo2 = AlgoBase(**params)
```

Parameters **deep** (*bool*) – flag to deeply copy objects such as *impl*.

Returns attribute values in dictionary.

Return type *dict*

load_model (*fname*)

Load neural network parameters.

```
algo.load_model('model.pt')
```

Parameters **fname** (*str*) – source file path.

predict (*x*, *action*, *with_variance=False*)

Returns predicted observation and reward.

Parameters

- **x** (*numpy.ndarray*) – observation
- **action** (*numpy.ndarray*) – action
- **with_variance** (*bool*) – flag to return prediction variance.

Returns tuple of predicted observation and reward.

Return type *tuple*

save_model (*fname*)

Saves neural network parameters.

```
algo.save_model('model.pt')
```

Parameters **fname** (*str*) – destination file path.

set_params (***params*)

Sets the given arguments to the attributes if they exist.

This method sets the given values to the attributes including ones in subclasses. If the values that don't exist as attributes are passed, they are ignored. Some of scikit-learn utilities will use this method.

```
algo.set_params(batch_size=100)
```

Parameters ****params** – arbitrary inputs to set as attributes.

Returns itself.

Return type `d3rlpy.algos.base.AlgoBase`

update (*epoch*, *total_step*, *batch*)

Update parameters with mini-batch of data.

Parameters

- **epoch** (*int*) – the current number of epochs.
- **total_step** (*int*) – the current number of total iterations.
- **batch** (`d3rlpy.dataset.TransitionMiniBatch`) – mini-batch data.

Returns loss values.

Return type *list*

4.1 Recommended Platforms

d3rlpy supports Linux, macOS and also Windows.

4.2 Install d3rlpy

4.2.1 Install via PyPI

pip is a recommended way to install d3rlpy:

```
$ pip install d3rlpy
```

4.2.2 Install via Anaconda

d3rlpy is also available on *conda-forge*:

```
$ conda install -c conda-forge d3rlpy
```

4.2.3 Install from source

You can also install via GitHub repository:

```
$ git clone https://github.com/takuseno/d3rlpy
$ cd d3rlpy
$ pip install Cython numpy # if you have not installed them.
$ pip install -e .
```


MIT License

Copyright (c) 2020 Takuma Seno

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 6

Indices and tables

- `genindex`
- `modindex`
- `search`

d

- `d3rlpy`, 9
- `d3rlpy.algos`, 9
- `d3rlpy.augmentation`, 152
- `d3rlpy.dataset`, 126
- `d3rlpy.datasets`, 136
- `d3rlpy.dynamics`, 179
- `d3rlpy.encoders`, 146
- `d3rlpy.metrics`, 161
- `d3rlpy.models.torch.q_functions`, 126
- `d3rlpy.online`, 174
- `d3rlpy.optimizers`, 142
- `d3rlpy.preprocessing`, 137

Symbols

`__getitem__()` (*d3rlpy.dataset.Episode method*), 131
`__getitem__()` (*d3rlpy.dataset.MDPDataset method*), 128
`__iter__()` (*d3rlpy.dataset.Episode method*), 131
`__iter__()` (*d3rlpy.dataset.MDPDataset method*), 128
`__len__()` (*d3rlpy.dataset.Episode method*), 131
`__len__()` (*d3rlpy.dataset.MDPDataset method*), 128
`__len__()` (*d3rlpy.online.buffers.ReplayBuffer method*), 175

A

`action` (*d3rlpy.dataset.Transition attribute*), 133
`action_flexibility` (*d3rlpy.algos.BCQ attribute*), 41
`action_flexibility` (*d3rlpy.algos.DiscreteBCQ attribute*), 107
`action_size` (*d3rlpy.online.buffers.ReplayBuffer attribute*), 175
`actions` (*d3rlpy.dataset.Episode attribute*), 132
`actions` (*d3rlpy.dataset.MDPDataset attribute*), 130
`actions` (*d3rlpy.dataset.TransitionMiniBatch attribute*), 135
`activation` (*d3rlpy.encoders.PixelEncoderFactory attribute*), 150
`activation` (*d3rlpy.encoders.VectorEncoderFactory attribute*), 151
`actor_encoder_factory` (*d3rlpy.algos.AWAC attribute*), 73
`actor_encoder_factory` (*d3rlpy.algos.AWR attribute*), 66
`actor_encoder_factory` (*d3rlpy.algos.BCQ attribute*), 40
`actor_encoder_factory` (*d3rlpy.algos.BEAR attribute*), 49
`actor_encoder_factory` (*d3rlpy.algos.CQL attribute*), 58
`actor_encoder_factory` (*d3rlpy.algos.DDPG attribute*), 16

`actor_encoder_factory` (*d3rlpy.algos.DiscreteAWR attribute*), 121
`actor_encoder_factory` (*d3rlpy.algos.DiscreteSAC attribute*), 99
`actor_encoder_factory` (*d3rlpy.algos.SAC attribute*), 32
`actor_encoder_factory` (*d3rlpy.algos.TD3 attribute*), 24
`actor_learning_rate` (*d3rlpy.algos.AWAC attribute*), 72
`actor_learning_rate` (*d3rlpy.algos.AWR attribute*), 65
`actor_learning_rate` (*d3rlpy.algos.BCQ attribute*), 40
`actor_learning_rate` (*d3rlpy.algos.BEAR attribute*), 49
`actor_learning_rate` (*d3rlpy.algos.CQL attribute*), 57
`actor_learning_rate` (*d3rlpy.algos.DDPG attribute*), 16
`actor_learning_rate` (*d3rlpy.algos.DiscreteAWR attribute*), 120
`actor_learning_rate` (*d3rlpy.algos.DiscreteSAC attribute*), 99
`actor_learning_rate` (*d3rlpy.algos.SAC attribute*), 31
`actor_learning_rate` (*d3rlpy.algos.TD3 attribute*), 24
`actor_optim_factory` (*d3rlpy.algos.AWAC attribute*), 73
`actor_optim_factory` (*d3rlpy.algos.AWR attribute*), 66
`actor_optim_factory` (*d3rlpy.algos.BCQ attribute*), 40
`actor_optim_factory` (*d3rlpy.algos.BEAR attribute*), 49
`actor_optim_factory` (*d3rlpy.algos.CQL attribute*), 58
`actor_optim_factory` (*d3rlpy.algos.DDPG attribute*), 16

tribute), 16
 actor_optim_factory (*d3rlpy.algos.DiscreteAWR attribute*), 120
 actor_optim_factory (*d3rlpy.algos.DiscreteSAC attribute*), 99
 actor_optim_factory (*d3rlpy.algos.SAC attribute*), 32
 actor_optim_factory (*d3rlpy.algos.TD3 attribute*), 24
 AdamFactory (*class in d3rlpy.optimizers*), 145
 alpha_learning_rate (*d3rlpy.algos.BEAR attribute*), 49
 alpha_learning_rate (*d3rlpy.algos.CQL attribute*), 58
 alpha_optim_factory (*d3rlpy.algos.BEAR attribute*), 49
 alpha_optim_factory (*d3rlpy.algos.CQL attribute*), 58
 alpha_threshold (*d3rlpy.algos.BEAR attribute*), 50
 alpha_threshold (*d3rlpy.algos.CQL attribute*), 59
 append() (*d3rlpy.dataset.MDPDataset method*), 128
 append() (*d3rlpy.online.buffers.ReplayBuffer method*), 175
 append_episode() (*d3rlpy.online.buffers.ReplayBuffer method*), 175
 augmentation (*d3rlpy.algos.AWAC attribute*), 74
 augmentation (*d3rlpy.algos.AWR attribute*), 67
 augmentation (*d3rlpy.algos.BC attribute*), 11
 augmentation (*d3rlpy.algos.BCQ attribute*), 42
 augmentation (*d3rlpy.algos.BEAR attribute*), 51
 augmentation (*d3rlpy.algos.CQL attribute*), 59
 augmentation (*d3rlpy.algos.DDPG attribute*), 17
 augmentation (*d3rlpy.algos.DiscreteAWR attribute*), 122
 augmentation (*d3rlpy.algos.DiscreteBC attribute*), 80
 augmentation (*d3rlpy.algos.DiscreteBCQ attribute*), 108
 augmentation (*d3rlpy.algos.DiscreteCQL attribute*), 114
 augmentation (*d3rlpy.algos.DiscreteSAC attribute*), 100
 augmentation (*d3rlpy.algos.DoubleDQN attribute*), 93
 augmentation (*d3rlpy.algos.DQN attribute*), 86
 augmentation (*d3rlpy.algos.SAC attribute*), 33
 augmentation (*d3rlpy.algos.TD3 attribute*), 25
 average_value_estimation_scorer() (*in module d3rlpy.metrics.scorer*), 163
 AWAC (*class in d3rlpy.algos*), 71
 AWR (*class in d3rlpy.algos*), 64

B

batch_size (*d3rlpy.algos.AWAC attribute*), 73
 batch_size (*d3rlpy.algos.AWR attribute*), 66
 batch_size (*d3rlpy.algos.BC attribute*), 10
 batch_size (*d3rlpy.algos.BCQ attribute*), 40
 batch_size (*d3rlpy.algos.BEAR attribute*), 50
 batch_size (*d3rlpy.algos.CQL attribute*), 58
 batch_size (*d3rlpy.algos.DDPG attribute*), 16
 batch_size (*d3rlpy.algos.DiscreteAWR attribute*), 121
 batch_size (*d3rlpy.algos.DiscreteBC attribute*), 80
 batch_size (*d3rlpy.algos.DiscreteBCQ attribute*), 107
 batch_size (*d3rlpy.algos.DiscreteCQL attribute*), 114
 batch_size (*d3rlpy.algos.DiscreteSAC attribute*), 100
 batch_size (*d3rlpy.algos.DoubleDQN attribute*), 92
 batch_size (*d3rlpy.algos.DQN attribute*), 85
 batch_size (*d3rlpy.algos.SAC attribute*), 32
 batch_size (*d3rlpy.algos.TD3 attribute*), 24
 batch_size (*d3rlpy.dynamics.mopo.MOPO attribute*), 181
 batch_size_per_update (*d3rlpy.algos.AWR attribute*), 66
 batch_size_per_update (*d3rlpy.algos.DiscreteAWR attribute*), 121
 BC (*class in d3rlpy.algos*), 9
 BCQ (*class in d3rlpy.algos*), 38
 BEAR (*class in d3rlpy.algos*), 47
 beta (*d3rlpy.algos.AWR attribute*), 66
 beta (*d3rlpy.algos.BCQ attribute*), 41
 beta (*d3rlpy.algos.DiscreteAWR attribute*), 121
 beta (*d3rlpy.algos.DiscreteBC attribute*), 80
 beta (*d3rlpy.algos.DiscreteBCQ attribute*), 107
 bootstrap (*d3rlpy.algos.AWAC attribute*), 73
 bootstrap (*d3rlpy.algos.BCQ attribute*), 41
 bootstrap (*d3rlpy.algos.BEAR attribute*), 50
 bootstrap (*d3rlpy.algos.CQL attribute*), 59
 bootstrap (*d3rlpy.algos.DDPG attribute*), 17
 bootstrap (*d3rlpy.algos.DiscreteBCQ attribute*), 107
 bootstrap (*d3rlpy.algos.DiscreteCQL attribute*), 114
 bootstrap (*d3rlpy.algos.DiscreteSAC attribute*), 100
 bootstrap (*d3rlpy.algos.DoubleDQN attribute*), 92
 bootstrap (*d3rlpy.algos.DQN attribute*), 86
 bootstrap (*d3rlpy.algos.SAC attribute*), 32
 bootstrap (*d3rlpy.algos.TD3 attribute*), 24
 brightness (*d3rlpy.augmentation.image.ColorJitter attribute*), 158
 build_episodes() (*d3rlpy.dataset.MDPDataset method*), 128
 build_transitions() (*d3rlpy.dataset.Episode method*), 131

C

clear_links() (*d3rlpy.dataset.Transition method*), 133
 clip_reward() (*d3rlpy.dataset.MDPDataset method*), 129

ColorJitter (class in d3rlpy.augmentation.image), 158
 compare_continuous_action_diff() (in module d3rlpy.metrics.comparer), 166
 compare_discrete_action_match() (in module d3rlpy.metrics.comparer), 167
 compute_epsilon() (d3rlpy.online.explorers.LinearDecayEpsilonGreedy method), 177
 compute_return() (d3rlpy.dataset.Episode method), 131
 compute_stats() (d3rlpy.dataset.MDPDataset method), 129
 continuous_action_diff_scorer() (in module d3rlpy.metrics.scorer), 165
 contrast (d3rlpy.augmentation.image.ColorJitter attribute), 158
 CQL (class in d3rlpy.algos), 56
 create() (d3rlpy.encoders.DefaultEncoderFactory method), 149
 create() (d3rlpy.encoders.PixelEncoderFactory method), 150
 create() (d3rlpy.encoders.VectorEncoderFactory method), 151
 create() (d3rlpy.optimizers.AdamFactory method), 145
 create() (d3rlpy.optimizers.OptimizerFactory method), 143
 create() (d3rlpy.optimizers.RMSpropFactory method), 146
 create() (d3rlpy.optimizers.SGDFactory method), 144
 create_impl() (d3rlpy.algos.AWAC method), 74
 create_impl() (d3rlpy.algos.AWR method), 67
 create_impl() (d3rlpy.algos.BC method), 11
 create_impl() (d3rlpy.algos.BCQ method), 42
 create_impl() (d3rlpy.algos.BEAR method), 51
 create_impl() (d3rlpy.algos.CQL method), 60
 create_impl() (d3rlpy.algos.DDPG method), 18
 create_impl() (d3rlpy.algos.DiscreteAWR method), 122
 create_impl() (d3rlpy.algos.DiscreteBC method), 81
 create_impl() (d3rlpy.algos.DiscreteBCQ method), 108
 create_impl() (d3rlpy.algos.DiscreteCQL method), 115
 create_impl() (d3rlpy.algos.DiscreteSAC method), 101
 create_impl() (d3rlpy.algos.DoubleDQN method), 93
 create_impl() (d3rlpy.algos.DQN method), 87
 create_impl() (d3rlpy.algos.SAC method), 33
 create_impl() (d3rlpy.algos.TD3 method), 26
 create_impl() (d3rlpy.dynamics.mopo.MOPO method), 182
 critic_encoder_factory (d3rlpy.algos.AWAC attribute), 73
 critic_encoder_factory (d3rlpy.algos.AWR attribute), 66
 critic_encoder_factory (d3rlpy.algos.BCQ attribute), 40
 critic_encoder_factory (d3rlpy.algos.BEAR attribute), 49
 critic_encoder_factory (d3rlpy.algos.CQL attribute), 58
 critic_encoder_factory (d3rlpy.algos.DDPG attribute), 16
 critic_encoder_factory (d3rlpy.algos.DiscreteAWR attribute), 121
 critic_encoder_factory (d3rlpy.algos.DiscreteSAC attribute), 99
 critic_encoder_factory (d3rlpy.algos.SAC attribute), 32
 critic_encoder_factory (d3rlpy.algos.TD3 attribute), 24
 critic_learning_rate (d3rlpy.algos.AWAC attribute), 72
 critic_learning_rate (d3rlpy.algos.AWR attribute), 65
 critic_learning_rate (d3rlpy.algos.BCQ attribute), 40
 critic_learning_rate (d3rlpy.algos.BEAR attribute), 49
 critic_learning_rate (d3rlpy.algos.CQL attribute), 57
 critic_learning_rate (d3rlpy.algos.DDPG attribute), 16
 critic_learning_rate (d3rlpy.algos.DiscreteAWR attribute), 120
 critic_learning_rate (d3rlpy.algos.DiscreteSAC attribute), 99
 critic_learning_rate (d3rlpy.algos.SAC attribute), 31
 critic_learning_rate (d3rlpy.algos.TD3 attribute), 24
 critic_optim_factory (d3rlpy.algos.AWAC attribute), 73
 critic_optim_factory (d3rlpy.algos.AWR attribute), 66
 critic_optim_factory (d3rlpy.algos.BCQ attribute), 40
 critic_optim_factory (d3rlpy.algos.BEAR attribute), 49
 critic_optim_factory (d3rlpy.algos.CQL attribute), 58
 critic_optim_factory (d3rlpy.algos.DDPG attribute), 16

critic_optim_factory
 (*d3rlpy.algos.DiscreteAWR attribute*), 120
critic_optim_factory (*d3rlpy.algos.DiscreteSAC attribute*), 99
critic_optim_factory (*d3rlpy.algos.SAC attribute*), 32
critic_optim_factory (*d3rlpy.algos.TD3 attribute*), 24
Cutout (*class in d3rlpy.augmentation.image*), 153

D

d3rlpy (*module*), 9
d3rlpy.algos (*module*), 9
d3rlpy.augmentation (*module*), 152
d3rlpy.dataset (*module*), 126
d3rlpy.datasets (*module*), 136
d3rlpy.dynamics (*module*), 179
d3rlpy.encoders (*module*), 146
d3rlpy.metrics (*module*), 161
d3rlpy.models.torch.q_functions (*module*), 126
d3rlpy.online (*module*), 174
d3rlpy.optimizers (*module*), 142
d3rlpy.preprocessing (*module*), 137
DDPG (*class in d3rlpy.algos*), 15
DefaultEncoderFactory (*class in d3rlpy.encoders*), 148
degree (*d3rlpy.augmentation.image.RandomRotation attribute*), 156
discounted_sum_of_advantage_scorer() (*in module d3rlpy.metrics.scorer*), 163
discrete_action (*d3rlpy.dynamics.mopo.MOPO attribute*), 181
discrete_action_match_scorer() (*in module d3rlpy.metrics.scorer*), 165
DiscreteAWR (*class in d3rlpy.algos*), 119
DiscreteBC (*class in d3rlpy.algos*), 79
DiscreteBCQ (*class in d3rlpy.algos*), 105
DiscreteCQL (*class in d3rlpy.algos*), 112
DiscreteSAC (*class in d3rlpy.algos*), 98
DoubleDQN (*class in d3rlpy.algos*), 91
DQN (*class in d3rlpy.algos*), 84
dump() (*d3rlpy.dataset.MDPDataset method*), 129
duration (*d3rlpy.online.explorers.LinearDecayEpsilonGreedy attribute*), 176
dynamics (*d3rlpy.algos.AWAC attribute*), 74
dynamics (*d3rlpy.algos.AWR attribute*), 67
dynamics (*d3rlpy.algos.BC attribute*), 11
dynamics (*d3rlpy.algos.BCQ attribute*), 42
dynamics (*d3rlpy.algos.BEAR attribute*), 51
dynamics (*d3rlpy.algos.CQL attribute*), 59
dynamics (*d3rlpy.algos.DDPG attribute*), 17
dynamics (*d3rlpy.algos.DiscreteAWR attribute*), 122
dynamics (*d3rlpy.algos.DiscreteBC attribute*), 80

dynamics (*d3rlpy.algos.DiscreteBCQ attribute*), 108
dynamics (*d3rlpy.algos.DiscreteCQL attribute*), 114
dynamics (*d3rlpy.algos.DiscreteSAC attribute*), 100
dynamics (*d3rlpy.algos.DoubleDQN attribute*), 93
dynamics (*d3rlpy.algos.DQN attribute*), 86
dynamics (*d3rlpy.algos.SAC attribute*), 33
dynamics (*d3rlpy.algos.TD3 attribute*), 25
dynamics_observation_prediction_error_scorer() (*in module d3rlpy.metrics.scorer*), 168
dynamics_prediction_variance_scorer() (*in module d3rlpy.metrics.scorer*), 169
dynamics_reward_prediction_error_scorer() (*in module d3rlpy.metrics.scorer*), 168

E

encoder_factory (*d3rlpy.algos.BC attribute*), 10
encoder_factory (*d3rlpy.algos.DiscreteBC attribute*), 80
encoder_factory (*d3rlpy.algos.DiscreteBCQ attribute*), 107
encoder_factory (*d3rlpy.algos.DiscreteCQL attribute*), 113
encoder_factory (*d3rlpy.algos.DoubleDQN attribute*), 92
encoder_factory (*d3rlpy.algos.DQN attribute*), 85
encoder_factory (*d3rlpy.dynamics.mopo.MOPO attribute*), 181
end_epsilon (*d3rlpy.online.explorers.LinearDecayEpsilonGreedy attribute*), 176
Episode (*class in d3rlpy.dataset*), 131
episodes (*d3rlpy.dataset.MDPDataset attribute*), 130
eval_results_ (*d3rlpy.algos.AWR attribute*), 67
eval_results_ (*d3rlpy.algos.BC attribute*), 11
eval_results_ (*d3rlpy.algos.BCQ attribute*), 42
eval_results_ (*d3rlpy.algos.BEAR attribute*), 51
eval_results_ (*d3rlpy.algos.CQL attribute*), 60
eval_results_ (*d3rlpy.algos.DDPG attribute*), 18
eval_results_ (*d3rlpy.algos.DiscreteAWR attribute*), 122
eval_results_ (*d3rlpy.algos.DiscreteBC attribute*), 80
eval_results_ (*d3rlpy.algos.DiscreteBCQ attribute*), 108
eval_results_ (*d3rlpy.algos.DiscreteCQL attribute*), 114
eval_results_ (*d3rlpy.algos.DiscreteSAC attribute*), 101
eval_results_ (*d3rlpy.algos.DQN attribute*), 86
eval_results_ (*d3rlpy.algos.SAC attribute*), 33
eval_results_ (*d3rlpy.algos.TD3 attribute*), 25
eval_results_ (*d3rlpy.dynamics.mopo.MOPO attribute*), 182
evaluate_on_environment() (*in module d3rlpy.metrics.scorer*), 166

`extend()` (*d3rlpy.dataset.MDPDataset method*), 129

F

`feature_size` (*d3rlpy.encoders.PixelEncoderFactory attribute*), 150

`filters` (*d3rlpy.encoders.PixelEncoderFactory attribute*), 149

`fit()` (*d3rlpy.algos.AWAC method*), 74

`fit()` (*d3rlpy.algos.AWR method*), 67

`fit()` (*d3rlpy.algos.BC method*), 11

`fit()` (*d3rlpy.algos.BCQ method*), 42

`fit()` (*d3rlpy.algos.BEAR method*), 51

`fit()` (*d3rlpy.algos.CQL method*), 60

`fit()` (*d3rlpy.algos.DDPG method*), 18

`fit()` (*d3rlpy.algos.DiscreteAWR method*), 122

`fit()` (*d3rlpy.algos.DiscreteBC method*), 81

`fit()` (*d3rlpy.algos.DiscreteBCQ method*), 108

`fit()` (*d3rlpy.algos.DiscreteCQL method*), 115

`fit()` (*d3rlpy.algos.DiscreteSAC method*), 101

`fit()` (*d3rlpy.algos.DoubleDQN method*), 93

`fit()` (*d3rlpy.algos.DQN method*), 87

`fit()` (*d3rlpy.algos.SAC method*), 33

`fit()` (*d3rlpy.algos.TD3 method*), 26

`fit()` (*d3rlpy.dynamics.mopo.MOPO method*), 182

`fit()` (*d3rlpy.preprocessing.MinMaxScaler method*), 140

`fit()` (*d3rlpy.preprocessing.PixelScaler method*), 138

`fit()` (*d3rlpy.preprocessing.StandardScaler method*), 142

`fit_online()` (*d3rlpy.algos.AWAC method*), 75

`fit_online()` (*d3rlpy.algos.AWR method*), 68

`fit_online()` (*d3rlpy.algos.BC method*), 12

`fit_online()` (*d3rlpy.algos.BCQ method*), 43

`fit_online()` (*d3rlpy.algos.BEAR method*), 52

`fit_online()` (*d3rlpy.algos.CQL method*), 60

`fit_online()` (*d3rlpy.algos.DDPG method*), 18

`fit_online()` (*d3rlpy.algos.DiscreteAWR method*), 123

`fit_online()` (*d3rlpy.algos.DiscreteBC method*), 81

`fit_online()` (*d3rlpy.algos.DiscreteBCQ method*), 109

`fit_online()` (*d3rlpy.algos.DiscreteCQL method*), 115

`fit_online()` (*d3rlpy.algos.DiscreteSAC method*), 101

`fit_online()` (*d3rlpy.algos.DoubleDQN method*), 94

`fit_online()` (*d3rlpy.algos.DQN method*), 87

`fit_online()` (*d3rlpy.algos.SAC method*), 34

`fit_online()` (*d3rlpy.algos.TD3 method*), 26

`from_json()` (*d3rlpy.algos.AWAC class method*), 76

`from_json()` (*d3rlpy.algos.AWR class method*), 68

`from_json()` (*d3rlpy.algos.BC class method*), 12

`from_json()` (*d3rlpy.algos.BCQ class method*), 43

`from_json()` (*d3rlpy.algos.BEAR class method*), 53

`from_json()` (*d3rlpy.algos.CQL class method*), 61

`from_json()` (*d3rlpy.algos.DDPG class method*), 19

`from_json()` (*d3rlpy.algos.DiscreteAWR class method*), 123

`from_json()` (*d3rlpy.algos.DiscreteBC class method*), 82

`from_json()` (*d3rlpy.algos.DiscreteBCQ class method*), 109

`from_json()` (*d3rlpy.algos.DiscreteCQL class method*), 116

`from_json()` (*d3rlpy.algos.DiscreteSAC class method*), 102

`from_json()` (*d3rlpy.algos.DoubleDQN class method*), 95

`from_json()` (*d3rlpy.algos.DQN class method*), 88

`from_json()` (*d3rlpy.algos.SAC class method*), 35

`from_json()` (*d3rlpy.algos.TD3 class method*), 27

`from_json()` (*d3rlpy.dynamics.mopo.MOPO class method*), 182

G

`gamma` (*d3rlpy.algos.AWAC attribute*), 73

`gamma` (*d3rlpy.algos.AWR attribute*), 66

`gamma` (*d3rlpy.algos.BCQ attribute*), 41

`gamma` (*d3rlpy.algos.BEAR attribute*), 50

`gamma` (*d3rlpy.algos.CQL attribute*), 58

`gamma` (*d3rlpy.algos.DDPG attribute*), 17

`gamma` (*d3rlpy.algos.DiscreteAWR attribute*), 121

`gamma` (*d3rlpy.algos.DiscreteBCQ attribute*), 107

`gamma` (*d3rlpy.algos.DiscreteCQL attribute*), 114

`gamma` (*d3rlpy.algos.DiscreteSAC attribute*), 100

`gamma` (*d3rlpy.algos.DoubleDQN attribute*), 92

`gamma` (*d3rlpy.algos.DQN attribute*), 86

`gamma` (*d3rlpy.algos.SAC attribute*), 32

`gamma` (*d3rlpy.algos.TD3 attribute*), 24

`generate()` (*d3rlpy.dynamics.mopo.MOPO method*), 183

`get_action_size()` (*d3rlpy.dataset.Episode method*), 132

`get_action_size()` (*d3rlpy.dataset.MDPDataset method*), 129

`get_action_size()` (*d3rlpy.dataset.Transition method*), 133

`get_atari()` (*in module d3rlpy.datasets*), 137

`get_cartpole()` (*in module d3rlpy.datasets*), 136

`get_observation_shape()` (*d3rlpy.dataset.Episode method*), 132

`get_observation_shape()` (*d3rlpy.dataset.MDPDataset method*), 129

`get_observation_shape()` (*d3rlpy.dataset.Transition method*), 133

`get_params()` (*d3rlpy.algos.AWAC method*), 76

`get_params()` (*d3rlpy.algos.AWR method*), 69

`get_params()` (*d3rlpy.algos.BC method*), 13

`get_params()` (*d3rlpy.algos.BCQ method*), 44
`get_params()` (*d3rlpy.algos.BEAR method*), 53
`get_params()` (*d3rlpy.algos.CQL method*), 62
`get_params()` (*d3rlpy.algos.DDPG method*), 20
`get_params()` (*d3rlpy.algos.DiscreteAWR method*), 124
`get_params()` (*d3rlpy.algos.DiscreteBC method*), 82
`get_params()` (*d3rlpy.algos.DiscreteBCQ method*), 110
`get_params()` (*d3rlpy.algos.DiscreteCQL method*), 116
`get_params()` (*d3rlpy.algos.DiscreteSAC method*), 103
`get_params()` (*d3rlpy.algos.DoubleDQN method*), 95
`get_params()` (*d3rlpy.algos.DQN method*), 88
`get_params()` (*d3rlpy.algos.SAC method*), 35
`get_params()` (*d3rlpy.algos.TD3 method*), 27
`get_params()` (*d3rlpy.augmentation.image.ColorJitter method*), 158
`get_params()` (*d3rlpy.augmentation.image.Cutout method*), 153
`get_params()` (*d3rlpy.augmentation.image.HorizontalFlip method*), 154
`get_params()` (*d3rlpy.augmentation.image.Intensity method*), 157
`get_params()` (*d3rlpy.augmentation.image.RandomRotation method*), 156
`get_params()` (*d3rlpy.augmentation.image.RandomShift method*), 153
`get_params()` (*d3rlpy.augmentation.image.VerticalFlip method*), 155
`get_params()` (*d3rlpy.augmentation.vector.MultipleAmplitudeScaling method*), 161
`get_params()` (*d3rlpy.augmentation.vector.SingleAmplitudeScaling method*), 159
`get_params()` (*d3rlpy.dynamics.mopo.MOPO method*), 183
`get_params()` (*d3rlpy.encoders.DefaultEncoderFactory method*), 149
`get_params()` (*d3rlpy.encoders.PixelEncoderFactory method*), 150
`get_params()` (*d3rlpy.encoders.VectorEncoderFactory method*), 151
`get_params()` (*d3rlpy.optimizers.AdamFactory method*), 145
`get_params()` (*d3rlpy.optimizers.OptimizerFactory method*), 144
`get_params()` (*d3rlpy.optimizers.RMSpropFactory method*), 146
`get_params()` (*d3rlpy.optimizers.SGDFactory method*), 144
`get_params()` (*d3rlpy.preprocessing.MinMaxScaler method*), 140
`get_params()` (*d3rlpy.preprocessing.PixelScaler method*), 138
`get_params()` (*d3rlpy.preprocessing.StandardScaler method*), 142
`get_pendulum()` (*in module d3rlpy.datasets*), 136
`get_pybullet()` (*in module d3rlpy.datasets*), 137
`get_type()` (*d3rlpy.augmentation.image.ColorJitter method*), 158
`get_type()` (*d3rlpy.augmentation.image.Cutout method*), 154
`get_type()` (*d3rlpy.augmentation.image.HorizontalFlip method*), 154
`get_type()` (*d3rlpy.augmentation.image.Intensity method*), 157
`get_type()` (*d3rlpy.augmentation.image.RandomRotation method*), 156
`get_type()` (*d3rlpy.augmentation.image.RandomShift method*), 153
`get_type()` (*d3rlpy.augmentation.image.VerticalFlip method*), 155
`get_type()` (*d3rlpy.augmentation.vector.MultipleAmplitudeScaling method*), 161
`get_type()` (*d3rlpy.augmentation.vector.SingleAmplitudeScaling method*), 160
`get_type()` (*d3rlpy.encoders.DefaultEncoderFactory method*), 149
`get_type()` (*d3rlpy.encoders.PixelEncoderFactory method*), 150
`get_type()` (*d3rlpy.encoders.VectorEncoderFactory method*), 151
`get_type()` (*d3rlpy.preprocessing.MinMaxScaler method*), 140
`get_type()` (*d3rlpy.preprocessing.PixelScaler method*), 139
`get_type()` (*d3rlpy.preprocessing.StandardScaler method*), 142

H

`hidden_units` (*d3rlpy.encoders.VectorEncoderFactory attribute*), 151
`horizon` (*d3rlpy.dynamics.mopo.MOPO attribute*), 181
`HorizontalFlip` (*class in d3rlpy.augmentation.image*), 154
`hue` (*d3rlpy.augmentation.image.ColorJitter attribute*), 158

I

`imitator_encoder_factory` (*d3rlpy.algos.BCQ attribute*), 40
`imitator_encoder_factory` (*d3rlpy.algos.BEAR attribute*), 49
`imitator_learning_rate` (*d3rlpy.algos.BCQ attribute*), 40
`imitator_learning_rate` (*d3rlpy.algos.BEAR attribute*), 49

- imitator_optim_factory (*d3rlpy.algos.BCQ attribute*), 40
 imitator_optim_factory (*d3rlpy.algos.BEAR attribute*), 49
 impl (*d3rlpy.algos.AWAC attribute*), 74
 impl (*d3rlpy.algos.AWR attribute*), 67
 impl (*d3rlpy.algos.BC attribute*), 11
 impl (*d3rlpy.algos.BCQ attribute*), 42
 impl (*d3rlpy.algos.BEAR attribute*), 51
 impl (*d3rlpy.algos.CQL attribute*), 59
 impl (*d3rlpy.algos.DDPG attribute*), 17
 impl (*d3rlpy.algos.DiscreteAWR attribute*), 122
 impl (*d3rlpy.algos.DiscreteBC attribute*), 80
 impl (*d3rlpy.algos.DiscreteBCQ attribute*), 108
 impl (*d3rlpy.algos.DiscreteCQL attribute*), 114
 impl (*d3rlpy.algos.DiscreteSAC attribute*), 100
 impl (*d3rlpy.algos.DoubleDQN attribute*), 93
 impl (*d3rlpy.algos.DQN attribute*), 86
 impl (*d3rlpy.algos.SAC attribute*), 33
 impl (*d3rlpy.algos.TD3 attribute*), 25
 impl (*d3rlpy.dynamics.mopo.MOPO attribute*), 182
 initial_alpha (*d3rlpy.algos.BEAR attribute*), 50
 initial_alpha (*d3rlpy.algos.CQL attribute*), 59
 initial_state_value_estimation_scorer() (in module *d3rlpy.metrics.scorer*), 164
 initial_temperature (*d3rlpy.algos.BEAR attribute*), 50
 initial_temperature (*d3rlpy.algos.CQL attribute*), 59
 initial_temperature (*d3rlpy.algos.DiscreteSAC attribute*), 100
 initial_temperature (*d3rlpy.algos.SAC attribute*), 33
 Intensity (class in *d3rlpy.augmentation.image*), 157
 is_action_discrete() (*d3rlpy.dataset.MDPDataset method*), 130
- ## L
- lam (*d3rlpy.algos.AWAC attribute*), 73
 lam (*d3rlpy.algos.AWR attribute*), 66
 lam (*d3rlpy.algos.BCQ attribute*), 41
 lam (*d3rlpy.algos.BEAR attribute*), 50
 lam (*d3rlpy.algos.DiscreteAWR attribute*), 121
 lam (*d3rlpy.dynamics.mopo.MOPO attribute*), 181
 latent_size (*d3rlpy.algos.BCQ attribute*), 41
 learning_rate (*d3rlpy.algos.BC attribute*), 10
 learning_rate (*d3rlpy.algos.DiscreteBC attribute*), 80
 learning_rate (*d3rlpy.algos.DiscreteBCQ attribute*), 106
 learning_rate (*d3rlpy.algos.DiscreteCQL attribute*), 113
 learning_rate (*d3rlpy.algos.DoubleDQN attribute*), 92
 learning_rate (*d3rlpy.algos.DQN attribute*), 85
 learning_rate (*d3rlpy.dynamics.mopo.MOPO attribute*), 181
 LinearDecayEpsilonGreedy (class in *d3rlpy.online.explorers*), 176
 load() (*d3rlpy.dataset.MDPDataset class method*), 130
 load_model() (*d3rlpy.algos.AWAC method*), 76
 load_model() (*d3rlpy.algos.AWR method*), 69
 load_model() (*d3rlpy.algos.BC method*), 13
 load_model() (*d3rlpy.algos.BCQ method*), 44
 load_model() (*d3rlpy.algos.BEAR method*), 53
 load_model() (*d3rlpy.algos.CQL method*), 62
 load_model() (*d3rlpy.algos.DDPG method*), 20
 load_model() (*d3rlpy.algos.DiscreteAWR method*), 124
 load_model() (*d3rlpy.algos.DiscreteBC method*), 83
 load_model() (*d3rlpy.algos.DiscreteBCQ method*), 110
 load_model() (*d3rlpy.algos.DiscreteCQL method*), 117
 load_model() (*d3rlpy.algos.DiscreteSAC method*), 103
 load_model() (*d3rlpy.algos.DoubleDQN method*), 95
 load_model() (*d3rlpy.algos.DQN method*), 89
 load_model() (*d3rlpy.algos.SAC method*), 35
 load_model() (*d3rlpy.algos.TD3 method*), 28
 load_model() (*d3rlpy.dynamics.mopo.MOPO method*), 183
- ## M
- max_weight (*d3rlpy.algos.AWAC attribute*), 73
 max_weight (*d3rlpy.algos.AWR attribute*), 66
 max_weight (*d3rlpy.algos.DiscreteAWR attribute*), 121
 maximum (*d3rlpy.augmentation.vector.MultipleAmplitudeScaling attribute*), 160
 maximum (*d3rlpy.augmentation.vector.SingleAmplitudeScaling attribute*), 159
 maximum (*d3rlpy.preprocessing.MinMaxScaler attribute*), 140
 MDPDataset (class in *d3rlpy.dataset*), 127
 mean (*d3rlpy.online.explorers.NormalNoise attribute*), 177
 mean (*d3rlpy.preprocessing.StandardScaler attribute*), 141
 minimum (*d3rlpy.augmentation.vector.MultipleAmplitudeScaling attribute*), 160
 minimum (*d3rlpy.augmentation.vector.SingleAmplitudeScaling attribute*), 159
 minimum (*d3rlpy.preprocessing.MinMaxScaler attribute*), 140
 MinMaxScaler (class in *d3rlpy.preprocessing*), 139
 mmd_sigma (*d3rlpy.algos.BEAR attribute*), 51
 MOPO (class in *d3rlpy.dynamics.mopo*), 180

MultipleAmplitudeScaling (class in *d3rlpy.augmentation.vector*), 160

N

n_action_samples (*d3rlpy.algos.AWAC* attribute), 73
 n_action_samples (*d3rlpy.algos.BCQ* attribute), 41
 n_action_samples (*d3rlpy.algos.BEAR* attribute), 50
 n_action_samples (*d3rlpy.algos.CQL* attribute), 59
 n_actor_updates (*d3rlpy.algos.AWR* attribute), 66
 n_actor_updates (*d3rlpy.algos.DiscreteAWR* attribute), 121
 n_augmentations (*d3rlpy.algos.AWAC* attribute), 74
 n_augmentations (*d3rlpy.algos.AWR* attribute), 67
 n_augmentations (*d3rlpy.algos.BC* attribute), 11
 n_augmentations (*d3rlpy.algos.BCQ* attribute), 42
 n_augmentations (*d3rlpy.algos.BEAR* attribute), 51
 n_augmentations (*d3rlpy.algos.CQL* attribute), 59
 n_augmentations (*d3rlpy.algos.DDPG* attribute), 17
 n_augmentations (*d3rlpy.algos.DiscreteAWR* attribute), 122
 n_augmentations (*d3rlpy.algos.DiscreteBC* attribute), 80
 n_augmentations (*d3rlpy.algos.DiscreteBCQ* attribute), 108
 n_augmentations (*d3rlpy.algos.DiscreteCQL* attribute), 114
 n_augmentations (*d3rlpy.algos.DiscreteSAC* attribute), 100
 n_augmentations (*d3rlpy.algos.DoubleDQN* attribute), 93
 n_augmentations (*d3rlpy.algos.DQN* attribute), 86
 n_augmentations (*d3rlpy.algos.SAC* attribute), 33
 n_augmentations (*d3rlpy.algos.TD3* attribute), 25
 n_critic_updates (*d3rlpy.algos.AWR* attribute), 66
 n_critic_updates (*d3rlpy.algos.DiscreteAWR* attribute), 121
 n_critics (*d3rlpy.algos.AWAC* attribute), 73
 n_critics (*d3rlpy.algos.BCQ* attribute), 41
 n_critics (*d3rlpy.algos.BEAR* attribute), 50
 n_critics (*d3rlpy.algos.CQL* attribute), 58
 n_critics (*d3rlpy.algos.DDPG* attribute), 17
 n_critics (*d3rlpy.algos.DiscreteBCQ* attribute), 107
 n_critics (*d3rlpy.algos.DiscreteCQL* attribute), 114
 n_critics (*d3rlpy.algos.DiscreteSAC* attribute), 100
 n_critics (*d3rlpy.algos.DoubleDQN* attribute), 92
 n_critics (*d3rlpy.algos.DQN* attribute), 86
 n_critics (*d3rlpy.algos.SAC* attribute), 32
 n_critics (*d3rlpy.algos.TD3* attribute), 24
 n_ensembles (*d3rlpy.dynamics.mopo.MOPO* attribute), 181
 n_frames (*d3rlpy.algos.AWAC* attribute), 73
 n_frames (*d3rlpy.algos.AWR* attribute), 66

n_frames (*d3rlpy.algos.BC* attribute), 10
 n_frames (*d3rlpy.algos.BCQ* attribute), 40
 n_frames (*d3rlpy.algos.BEAR* attribute), 50
 n_frames (*d3rlpy.algos.CQL* attribute), 58
 n_frames (*d3rlpy.algos.DDPG* attribute), 17
 n_frames (*d3rlpy.algos.DiscreteAWR* attribute), 121
 n_frames (*d3rlpy.algos.DiscreteBC* attribute), 80
 n_frames (*d3rlpy.algos.DiscreteBCQ* attribute), 107
 n_frames (*d3rlpy.algos.DiscreteCQL* attribute), 114
 n_frames (*d3rlpy.algos.DiscreteSAC* attribute), 100
 n_frames (*d3rlpy.algos.DoubleDQN* attribute), 92
 n_frames (*d3rlpy.algos.DQN* attribute), 85
 n_frames (*d3rlpy.algos.SAC* attribute), 32
 n_frames (*d3rlpy.algos.TD3* attribute), 24
 n_frames (*d3rlpy.dynamics.mopo.MOPO* attribute), 181
 n_transitions (*d3rlpy.dynamics.mopo.MOPO* attribute), 181
 next_action (*d3rlpy.dataset.Transition* attribute), 133
 next_actions (*d3rlpy.dataset.TransitionMiniBatch* attribute), 135
 next_observation (*d3rlpy.dataset.Transition* attribute), 134
 next_observations (*d3rlpy.dataset.TransitionMiniBatch* attribute), 135
 next_reward (*d3rlpy.dataset.Transition* attribute), 134
 next_rewards (*d3rlpy.dataset.TransitionMiniBatch* attribute), 135
 next_transition (*d3rlpy.dataset.Transition* attribute), 134
 NormalNoise (class in *d3rlpy.online.explorers*), 177

O

observation (*d3rlpy.dataset.Transition* attribute), 134
 observation_shape (*d3rlpy.online.buffers.ReplayBuffer* attribute), 175
 observations (*d3rlpy.dataset.Episode* attribute), 132
 observations (*d3rlpy.dataset.MDPDataset* attribute), 130
 observations (*d3rlpy.dataset.TransitionMiniBatch* attribute), 135
 optim_cls (*d3rlpy.optimizers.AdamFactory* attribute), 145
 optim_cls (*d3rlpy.optimizers.OptimizerFactory* attribute), 143
 optim_cls (*d3rlpy.optimizers.RMSpropFactory* attribute), 146
 optim_cls (*d3rlpy.optimizers.SGDFactory* attribute), 144

optim_factory (*d3rlpy.algos.BC attribute*), 10
 optim_factory (*d3rlpy.algos.DiscreteBC attribute*), 80
 optim_factory (*d3rlpy.algos.DiscreteBCQ attribute*), 106
 optim_factory (*d3rlpy.algos.DiscreteCQL attribute*), 113
 optim_factory (*d3rlpy.algos.DoubleDQN attribute*), 92
 optim_factory (*d3rlpy.algos.DQN attribute*), 85
 optim_factory (*d3rlpy.dynamics.mopo.MOPO attribute*), 181
 optim_kwargs (*d3rlpy.optimizers.AdamFactory attribute*), 145
 optim_kwargs (*d3rlpy.optimizers.OptimizerFactory attribute*), 143
 optim_kwargs (*d3rlpy.optimizers.RMSpropFactory attribute*), 146
 optim_kwargs (*d3rlpy.optimizers.SGDFactory attribute*), 144
 OptimizerFactory (*class in d3rlpy.optimizers*), 143

P

PixelEncoderFactory (*class in d3rlpy.encoders*), 149
 PixelScaler (*class in d3rlpy.preprocessing*), 138
 predict () (*d3rlpy.algos.AWAC method*), 76
 predict () (*d3rlpy.algos.AWR method*), 69
 predict () (*d3rlpy.algos.BC method*), 13
 predict () (*d3rlpy.algos.BCQ method*), 44
 predict () (*d3rlpy.algos.BEAR method*), 54
 predict () (*d3rlpy.algos.CQL method*), 62
 predict () (*d3rlpy.algos.DDPG method*), 20
 predict () (*d3rlpy.algos.DiscreteAWR method*), 124
 predict () (*d3rlpy.algos.DiscreteBC method*), 83
 predict () (*d3rlpy.algos.DiscreteBCQ method*), 110
 predict () (*d3rlpy.algos.DiscreteCQL method*), 117
 predict () (*d3rlpy.algos.DiscreteSAC method*), 103
 predict () (*d3rlpy.algos.DoubleDQN method*), 95
 predict () (*d3rlpy.algos.DQN method*), 89
 predict () (*d3rlpy.algos.SAC method*), 35
 predict () (*d3rlpy.algos.TD3 method*), 28
 predict () (*d3rlpy.dynamics.mopo.MOPO method*), 184
 predict_value () (*d3rlpy.algos.AWAC method*), 77
 predict_value () (*d3rlpy.algos.AWR method*), 70
 predict_value () (*d3rlpy.algos.BC method*), 14
 predict_value () (*d3rlpy.algos.BCQ method*), 45
 predict_value () (*d3rlpy.algos.BEAR method*), 54
 predict_value () (*d3rlpy.algos.CQL method*), 62
 predict_value () (*d3rlpy.algos.DDPG method*), 20
 predict_value () (*d3rlpy.algos.DiscreteAWR method*), 124

predict_value () (*d3rlpy.algos.DiscreteBC method*), 83
 predict_value () (*d3rlpy.algos.DiscreteBCQ method*), 110
 predict_value () (*d3rlpy.algos.DiscreteCQL method*), 117
 predict_value () (*d3rlpy.algos.DiscreteSAC method*), 103
 predict_value () (*d3rlpy.algos.DoubleDQN method*), 96
 predict_value () (*d3rlpy.algos.DQN method*), 89
 predict_value () (*d3rlpy.algos.SAC method*), 36
 predict_value () (*d3rlpy.algos.TD3 method*), 28
 prev_action (*d3rlpy.online.buffers.ReplayBuffer attribute*), 175
 prev_observation (*d3rlpy.online.buffers.ReplayBuffer attribute*), 175
 prev_reward (*d3rlpy.online.buffers.ReplayBuffer attribute*), 175
 prev_transition (*d3rlpy.dataset.Transition attribute*), 134
 prev_transition (*d3rlpy.online.buffers.ReplayBuffer attribute*), 175
 probability (*d3rlpy.augmentation.image.Cutout attribute*), 153
 probability (*d3rlpy.augmentation.image.HorizontalFlip attribute*), 154
 probability (*d3rlpy.augmentation.image.VerticalFlip attribute*), 155

Q

q_func_type (*d3rlpy.algos.AWAC attribute*), 74
 q_func_type (*d3rlpy.algos.BCQ attribute*), 41
 q_func_type (*d3rlpy.algos.BEAR attribute*), 51
 q_func_type (*d3rlpy.algos.CQL attribute*), 59
 q_func_type (*d3rlpy.algos.DDPG attribute*), 17
 q_func_type (*d3rlpy.algos.DiscreteBCQ attribute*), 107
 q_func_type (*d3rlpy.algos.DiscreteCQL attribute*), 114
 q_func_type (*d3rlpy.algos.DiscreteSAC attribute*), 100
 q_func_type (*d3rlpy.algos.DoubleDQN attribute*), 93
 q_func_type (*d3rlpy.algos.DQN attribute*), 86
 q_func_type (*d3rlpy.algos.SAC attribute*), 33
 q_func_type (*d3rlpy.algos.TD3 attribute*), 25

R

RandomRotation (*class in d3rlpy.augmentation.image*), 156
 RandomShift (*class in d3rlpy.augmentation.image*), 152
 reguralizing_rate (*d3rlpy.algos.DDPG attribute*), 17

reguralizing_rate (*d3rlpy.algos.TD3* attribute), 24
 ReplayBuffer (*class in d3rlpy.online.buffers*), 174
 reverse_transform() (*d3rlpy.preprocessing.MinMaxScaler* method), 140
 reverse_transform() (*d3rlpy.preprocessing.PixelScaler* method), 139
 reverse_transform() (*d3rlpy.preprocessing.StandardScaler* method), 142
 reward (*d3rlpy.dataset.Transition* attribute), 134
 rewards (*d3rlpy.dataset.Episode* attribute), 132
 rewards (*d3rlpy.dataset.MDPDataset* attribute), 130
 rewards (*d3rlpy.dataset.TransitionMiniBatch* attribute), 136
 rl_start_epoch (*d3rlpy.algos.BCQ* attribute), 41
 rl_start_epoch (*d3rlpy.algos.BEAR* attribute), 51
 RMSpropFactory (*class in d3rlpy.optimizers*), 145

S

SAC (*class in d3rlpy.algos*), 30
 sample() (*d3rlpy.online.buffers.ReplayBuffer* method), 175
 sample() (*d3rlpy.online.explorers.LinearDecayEpsilonGreedy* method), 177
 sample() (*d3rlpy.online.explorers.NormalNoise* method), 177
 sample_action() (*d3rlpy.algos.AWAC* method), 77
 sample_action() (*d3rlpy.algos.AWR* method), 70
 sample_action() (*d3rlpy.algos.BC* method), 14
 sample_action() (*d3rlpy.algos.BCQ* method), 45
 sample_action() (*d3rlpy.algos.BEAR* method), 54
 sample_action() (*d3rlpy.algos.CQL* method), 63
 sample_action() (*d3rlpy.algos.DDPG* method), 21
 sample_action() (*d3rlpy.algos.DiscreteAWR* method), 125
 sample_action() (*d3rlpy.algos.DiscreteBC* method), 83
 sample_action() (*d3rlpy.algos.DiscreteBCQ* method), 111
 sample_action() (*d3rlpy.algos.DiscreteCQL* method), 118
 sample_action() (*d3rlpy.algos.DiscreteSAC* method), 104
 sample_action() (*d3rlpy.algos.DoubleDQN* method), 96
 sample_action() (*d3rlpy.algos.DQN* method), 90
 sample_action() (*d3rlpy.algos.SAC* method), 36
 sample_action() (*d3rlpy.algos.TD3* method), 29
 saturation (*d3rlpy.augmentation.image.ColorJitter* attribute), 158
 save_model() (*d3rlpy.algos.AWAC* method), 77
 save_model() (*d3rlpy.algos.AWR* method), 70
 save_model() (*d3rlpy.algos.BC* method), 14
 save_model() (*d3rlpy.algos.BCQ* method), 45
 save_model() (*d3rlpy.algos.BEAR* method), 55
 save_model() (*d3rlpy.algos.CQL* method), 63
 save_model() (*d3rlpy.algos.DDPG* method), 21
 save_model() (*d3rlpy.algos.DiscreteAWR* method), 125
 save_model() (*d3rlpy.algos.DiscreteBC* method), 83
 save_model() (*d3rlpy.algos.DiscreteBCQ* method), 111
 save_model() (*d3rlpy.algos.DiscreteCQL* method), 118
 save_model() (*d3rlpy.algos.DiscreteSAC* method), 104
 save_model() (*d3rlpy.algos.DoubleDQN* method), 96
 save_model() (*d3rlpy.algos.DQN* method), 90
 save_model() (*d3rlpy.algos.SAC* method), 36
 save_model() (*d3rlpy.algos.TD3* method), 29
 save_model() (*d3rlpy.dynamics.mopo.MOPO* method), 184
 save_policy() (*d3rlpy.algos.AWAC* method), 78
 save_policy() (*d3rlpy.algos.AWR* method), 70
 save_policy() (*d3rlpy.algos.BC* method), 14
 save_policy() (*d3rlpy.algos.BCQ* method), 45
 save_policy() (*d3rlpy.algos.BEAR* method), 55
 save_policy() (*d3rlpy.algos.CQL* method), 63
 save_policy() (*d3rlpy.algos.DDPG* method), 21
 save_policy() (*d3rlpy.algos.DiscreteAWR* method), 125
 save_policy() (*d3rlpy.algos.DiscreteBC* method), 83
 save_policy() (*d3rlpy.algos.DiscreteBCQ* method), 111
 save_policy() (*d3rlpy.algos.DiscreteCQL* method), 118
 save_policy() (*d3rlpy.algos.DiscreteSAC* method), 104
 save_policy() (*d3rlpy.algos.DoubleDQN* method), 97
 save_policy() (*d3rlpy.algos.DQN* method), 90
 save_policy() (*d3rlpy.algos.SAC* method), 37
 save_policy() (*d3rlpy.algos.TD3* method), 29
 scale (*d3rlpy.augmentation.image.Intensity* attribute), 157
 scaler (*d3rlpy.algos.AWAC* attribute), 74
 scaler (*d3rlpy.algos.AWR* attribute), 67
 scaler (*d3rlpy.algos.BC* attribute), 10
 scaler (*d3rlpy.algos.BCQ* attribute), 42
 scaler (*d3rlpy.algos.BEAR* attribute), 51
 scaler (*d3rlpy.algos.CQL* attribute), 59
 scaler (*d3rlpy.algos.DDPG* attribute), 17
 scaler (*d3rlpy.algos.DiscreteAWR* attribute), 121
 scaler (*d3rlpy.algos.DiscreteBC* attribute), 80

- scaler (*d3rlpy.algos.DiscreteBCQ* attribute), 107
 scaler (*d3rlpy.algos.DiscreteCQL* attribute), 114
 scaler (*d3rlpy.algos.DiscreteSAC* attribute), 100
 scaler (*d3rlpy.algos.DoubleDQN* attribute), 93
 scaler (*d3rlpy.algos.DQN* attribute), 86
 scaler (*d3rlpy.algos.SAC* attribute), 33
 scaler (*d3rlpy.algos.TD3* attribute), 25
 scaler (*d3rlpy.dynamics.mopo.MOPO* attribute), 181
 set_params() (*d3rlpy.algos.AWAC* method), 78
 set_params() (*d3rlpy.algos.AWR* method), 70
 set_params() (*d3rlpy.algos.BC* method), 14
 set_params() (*d3rlpy.algos.BCQ* method), 46
 set_params() (*d3rlpy.algos.BEAR* method), 55
 set_params() (*d3rlpy.algos.CQL* method), 64
 set_params() (*d3rlpy.algos.DDPG* method), 22
 set_params() (*d3rlpy.algos.DiscreteAWR* method), 125
 set_params() (*d3rlpy.algos.DiscreteBC* method), 84
 set_params() (*d3rlpy.algos.DiscreteBCQ* method), 112
 set_params() (*d3rlpy.algos.DiscreteCQL* method), 118
 set_params() (*d3rlpy.algos.DiscreteSAC* method), 105
 set_params() (*d3rlpy.algos.DoubleDQN* method), 97
 set_params() (*d3rlpy.algos.DQN* method), 90
 set_params() (*d3rlpy.algos.SAC* method), 37
 set_params() (*d3rlpy.algos.TD3* method), 29
 set_params() (*d3rlpy.dynamics.mopo.MOPO* method), 184
 SGDFactory (class in *d3rlpy.optimizers*), 144
 share_encoder (*d3rlpy.algos.AWAC* attribute), 73
 share_encoder (*d3rlpy.algos.BCQ* attribute), 41
 share_encoder (*d3rlpy.algos.BEAR* attribute), 50
 share_encoder (*d3rlpy.algos.CQL* attribute), 59
 share_encoder (*d3rlpy.algos.DDPG* attribute), 17
 share_encoder (*d3rlpy.algos.DiscreteBCQ* attribute), 107
 share_encoder (*d3rlpy.algos.DiscreteSAC* attribute), 100
 share_encoder (*d3rlpy.algos.DoubleDQN* attribute), 92
 share_encoder (*d3rlpy.algos.DQN* attribute), 86
 share_encoder (*d3rlpy.algos.SAC* attribute), 32
 share_encoder (*d3rlpy.algos.TD3* attribute), 25
 shift_size (*d3rlpy.augmentation.image.RandomShift* attribute), 152
 SingleAmplitudeScaling (class in *d3rlpy.augmentation.vector*), 159
 size() (*d3rlpy.dataset.Episode* method), 132
 size() (*d3rlpy.dataset.MDPDataset* method), 130
 size() (*d3rlpy.dataset.TransitionMiniBatch* method), 135
 size() (*d3rlpy.online.buffers.ReplayBuffer* method), 176
 soft_opc_scorer() (in module *d3rlpy.metrics.scorer*), 164
 StandardScaler (class in *d3rlpy.preprocessing*), 141
 start_epsilon (*d3rlpy.online.explorers.LinearDecayEpsilonGreedy* attribute), 176
 std (*d3rlpy.online.explorers.NormalNoise* attribute), 177
 std (*d3rlpy.preprocessing.StandardScaler* attribute), 141

T

 target_smoothing_clip (*d3rlpy.algos.TD3* attribute), 25
 target_smoothing_sigma (*d3rlpy.algos.TD3* attribute), 25
 target_update_interval (*d3rlpy.algos.DiscreteBCQ* attribute), 107
 target_update_interval (*d3rlpy.algos.DiscreteCQL* attribute), 114
 target_update_interval (*d3rlpy.algos.DoubleDQN* attribute), 93
 target_update_interval (*d3rlpy.algos.DQN* attribute), 86
 tau (*d3rlpy.algos.AWAC* attribute), 73
 tau (*d3rlpy.algos.BCQ* attribute), 41
 tau (*d3rlpy.algos.BEAR* attribute), 50
 tau (*d3rlpy.algos.CQL* attribute), 58
 tau (*d3rlpy.algos.DDPG* attribute), 17
 tau (*d3rlpy.algos.SAC* attribute), 32
 tau (*d3rlpy.algos.TD3* attribute), 24
 TD3 (class in *d3rlpy.algos*), 22
 td_error_scorer() (in module *d3rlpy.metrics.scorer*), 162
 temp_learning_rate (*d3rlpy.algos.BEAR* attribute), 49
 temp_learning_rate (*d3rlpy.algos.CQL* attribute), 58
 temp_learning_rate (*d3rlpy.algos.DiscreteSAC* attribute), 99
 temp_learning_rate (*d3rlpy.algos.SAC* attribute), 31
 temp_optim_factory (*d3rlpy.algos.BEAR* attribute), 49
 temp_optim_factory (*d3rlpy.algos.CQL* attribute), 58
 temp_optim_factory (*d3rlpy.algos.DiscreteSAC* attribute), 99
 temp_optim_factory (*d3rlpy.algos.SAC* attribute), 32
 terminal (*d3rlpy.dataset.Transition* attribute), 134
 terminals (*d3rlpy.dataset.MDPDataset* attribute), 131

<code>terminals</code> (<i>d3rlpy.dataset.TransitionMiniBatch attribute</i>), 136	<code>TYPE</code> (<i>d3rlpy.encoders.PixelEncoderFactory attribute</i>), 150
<code>train()</code> (<i>in module d3rlpy.online.iterators</i>), 178	<code>TYPE</code> (<i>d3rlpy.encoders.VectorEncoderFactory attribute</i>), 151
<code>transform()</code> (<i>d3rlpy.augmentation.image.ColorJitter method</i>), 158	<code>TYPE</code> (<i>d3rlpy.preprocessing.MinMaxScaler attribute</i>), 141
<code>transform()</code> (<i>d3rlpy.augmentation.image.Cutout method</i>), 154	<code>TYPE</code> (<i>d3rlpy.preprocessing.PixelScaler attribute</i>), 139
<code>transform()</code> (<i>d3rlpy.augmentation.image.HorizontalFlip method</i>), 154	<code>TYPE</code> (<i>d3rlpy.preprocessing.StandardScaler attribute</i>), 142
<code>transform()</code> (<i>d3rlpy.augmentation.image.Intensity method</i>), 157	U
<code>transform()</code> (<i>d3rlpy.augmentation.image.RandomRotation method</i>), 156	
<code>transform()</code> (<i>d3rlpy.augmentation.image.RandomShift method</i>), 153	<code>update()</code> (<i>d3rlpy.algos.AWAC method</i>), 78
<code>transform()</code> (<i>d3rlpy.augmentation.image.VerticalFlip method</i>), 155	<code>update()</code> (<i>d3rlpy.algos.AWR method</i>), 71
<code>transform()</code> (<i>d3rlpy.augmentation.vector.MultipleAmplitudeScaling method</i>), 161	<code>update()</code> (<i>d3rlpy.algos.BC method</i>), 14
<code>transform()</code> (<i>d3rlpy.augmentation.vector.SingleAmplitudeScaling method</i>), 160	<code>update()</code> (<i>d3rlpy.algos.BCQ method</i>), 46
<code>transform()</code> (<i>d3rlpy.preprocessing.MinMaxScaler method</i>), 140	<code>update()</code> (<i>d3rlpy.algos.BEAR method</i>), 55
<code>transform()</code> (<i>d3rlpy.preprocessing.PixelScaler method</i>), 139	<code>update()</code> (<i>d3rlpy.algos.CQL method</i>), 64
<code>transform()</code> (<i>d3rlpy.preprocessing.StandardScaler method</i>), 142	<code>update()</code> (<i>d3rlpy.algos.DDPG method</i>), 22
<code>Transition</code> (<i>class in d3rlpy.dataset</i>), 133	<code>update()</code> (<i>d3rlpy.algos.DiscreteAWR method</i>), 126
<code>TransitionMiniBatch</code> (<i>class in d3rlpy.dataset</i>), 134	<code>update()</code> (<i>d3rlpy.algos.DiscreteBC method</i>), 84
<code>transitions</code> (<i>d3rlpy.dataset.Episode attribute</i>), 132	<code>update()</code> (<i>d3rlpy.algos.DiscreteBCQ method</i>), 112
<code>transitions</code> (<i>d3rlpy.dataset.TransitionMiniBatch attribute</i>), 136	<code>update()</code> (<i>d3rlpy.algos.DiscreteCQL method</i>), 119
<code>transitions</code> (<i>d3rlpy.online.buffers.ReplayBuffer attribute</i>), 175	<code>update()</code> (<i>d3rlpy.algos.DiscreteSAC method</i>), 105
<code>TYPE</code> (<i>d3rlpy.augmentation.image.ColorJitter attribute</i>), 159	<code>update()</code> (<i>d3rlpy.algos.DoubleDQN method</i>), 97
<code>TYPE</code> (<i>d3rlpy.augmentation.image.Cutout attribute</i>), 154	<code>update()</code> (<i>d3rlpy.algos.DQN method</i>), 91
<code>TYPE</code> (<i>d3rlpy.augmentation.image.HorizontalFlip attribute</i>), 155	<code>update()</code> (<i>d3rlpy.algos.SAC method</i>), 37
<code>TYPE</code> (<i>d3rlpy.augmentation.image.Intensity attribute</i>), 157	<code>update()</code> (<i>d3rlpy.algos.TD3 method</i>), 30
<code>TYPE</code> (<i>d3rlpy.augmentation.image.RandomRotation attribute</i>), 156	<code>update()</code> (<i>d3rlpy.dynamics.mopo.MOPO method</i>), 184
<code>TYPE</code> (<i>d3rlpy.augmentation.image.RandomShift attribute</i>), 153	<code>update_actor_interval</code> (<i>d3rlpy.algos.AWAC attribute</i>), 74
<code>TYPE</code> (<i>d3rlpy.augmentation.image.VerticalFlip attribute</i>), 156	<code>update_actor_interval</code> (<i>d3rlpy.algos.BCQ attribute</i>), 41
<code>TYPE</code> (<i>d3rlpy.augmentation.vector.MultipleAmplitudeScaling attribute</i>), 161	<code>update_actor_interval</code> (<i>d3rlpy.algos.BEAR attribute</i>), 50
<code>TYPE</code> (<i>d3rlpy.augmentation.vector.SingleAmplitudeScaling attribute</i>), 160	<code>update_actor_interval</code> (<i>d3rlpy.algos.CQL attribute</i>), 59
<code>TYPE</code> (<i>d3rlpy.encoders.DefaultEncoderFactory attribute</i>), 149	<code>update_actor_interval</code> (<i>d3rlpy.algos.SAC attribute</i>), 32
	<code>update_actor_interval</code> (<i>d3rlpy.algos.TD3 attribute</i>), 25
	<code>use_batch_norm</code> (<i>d3rlpy.encoders.DefaultEncoderFactory attribute</i>), 148
	<code>use_batch_norm</code> (<i>d3rlpy.encoders.PixelEncoderFactory attribute</i>), 150
	<code>use_batch_norm</code> (<i>d3rlpy.encoders.VectorEncoderFactory attribute</i>), 151
	<code>use_gpu</code> (<i>d3rlpy.algos.AWAC attribute</i>), 74
	<code>use_gpu</code> (<i>d3rlpy.algos.AWR attribute</i>), 66
	<code>use_gpu</code> (<i>d3rlpy.algos.BC attribute</i>), 10
	<code>use_gpu</code> (<i>d3rlpy.algos.BCQ attribute</i>), 41
	<code>use_gpu</code> (<i>d3rlpy.algos.BEAR attribute</i>), 51
	<code>use_gpu</code> (<i>d3rlpy.algos.CQL attribute</i>), 59
	<code>use_gpu</code> (<i>d3rlpy.algos.DDPG attribute</i>), 17
	<code>use_gpu</code> (<i>d3rlpy.algos.DiscreteAWR attribute</i>), 121

`use_gpu` (*d3rlpy.algos.DiscreteBC* attribute), 80
`use_gpu` (*d3rlpy.algos.DiscreteBCQ* attribute), 107
`use_gpu` (*d3rlpy.algos.DiscreteCQL* attribute), 114
`use_gpu` (*d3rlpy.algos.DiscreteSAC* attribute), 100
`use_gpu` (*d3rlpy.algos.DoubleDQN* attribute), 93
`use_gpu` (*d3rlpy.algos.DQN* attribute), 86
`use_gpu` (*d3rlpy.algos.SAC* attribute), 33
`use_gpu` (*d3rlpy.algos.TD3* attribute), 25
`use_gpu` (*d3rlpy.dynamics.mopo.MOPO* attribute), 181

V

`value_estimation_std_scorer()` (in module *d3rlpy.metrics.scorer*), 163
`VectorEncoderFactory` (class in *d3rlpy.encoders*), 150
`VerticalFlip` (class in *d3rlpy.augmentation.image*), 155