

---

**d3rlpy**

**Takuma Seno**

**Jul 25, 2021**



# TUTORIALS

<b>1 Getting Started</b>	<b>3</b>
1.1 Install . . . . .	3
1.2 Prepare Dataset . . . . .	3
1.3 Setup Algorithm . . . . .	4
1.4 Setup Metrics . . . . .	4
1.5 Start Training . . . . .	5
1.6 Save and Load . . . . .	6
<b>2 Jupyter Notebooks</b>	<b>7</b>
<b>3 API Reference</b>	<b>9</b>
3.1 Algorithms . . . . .	9
3.2 Q Functions . . . . .	280
3.3 MDPDataset . . . . .	286
3.4 Datasets . . . . .	297
3.5 Preprocessing . . . . .	301
3.6 Optimizers . . . . .	314
3.7 Network Architectures . . . . .	318
3.8 Metrics . . . . .	325
3.9 Off-Policy Evaluation . . . . .	333
3.10 Save and Load . . . . .	355
3.11 Logging . . . . .	358
3.12 scikit-learn compatibility . . . . .	358
3.13 Online Training . . . . .	361
3.14 (experimental) Model-based Algorithms . . . . .	368
3.15 Stable-Baselines3 Wrapper . . . . .	376
<b>4 Command Line Interface</b>	<b>379</b>
4.1 plot . . . . .	379
4.2 plot-all . . . . .	380
4.3 export . . . . .	381
4.4 record . . . . .	381
4.5 play . . . . .	382
<b>5 Installation</b>	<b>383</b>
5.1 Recommended Platforms . . . . .	383
5.2 Install d3rlpy . . . . .	383
<b>6 Tips</b>	<b>385</b>
6.1 Reproducibility . . . . .	385
6.2 Create your own dataset . . . . .	385

6.3	Learning from image observation . . . . .	386
6.4	Improve performance beyond the original paper . . . . .	387
<b>7</b>	<b>Paper Reproductions</b>	<b>389</b>
7.1	Offline . . . . .	389
7.2	Online . . . . .	391
<b>8</b>	<b>License</b>	<b>393</b>
<b>9</b>	<b>Indices and tables</b>	<b>395</b>
	<b>Python Module Index</b>	<b>397</b>
	<b>Index</b>	<b>399</b>

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

```
$ pip install d3rlpy
```

d3rlpy provides state-of-the-art offline 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 their papers via several tweaks.



## 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
```

See more information at [Installation](#).

---

**Note:** If core dump error occurs in this tutorial, please try [Install from source](#).

---

---

**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 [MDPDataset](#).

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
from d3rlpy.datasets import get_d4rl # D4RL 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)  
  
# initialize neural networks with the given observation shape and action size.  
# this is not necessary when you directly call fit or fit_online method.  
dqn.build_with_dataset(dataset)
```

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_factory=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]
```

(continues on next page)

(continued from previous page)

```
# 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.build_with_dataset(dataset)
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  
**TWO**

---

## JUPYTER NOTEBOOKS

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



## API REFERENCE

### 3.1 Algorithms

d3rlpy provides state-of-the-art offline 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.CRR</code>	Critic Regularized Regression 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.
<code>d3rlpy.algos.PLAS</code>	Policy in Latent Action Space algorithm.
<code>d3rlpy.algos.PLASWithPerturbation</code>	Policy in Latent Action Space algorithm with perturbation layer.
<code>d3rlpy.algos.TD3PlusBC</code>	TD3+BC algorithm.
<code>d3rlpy.algos.MOPO</code>	Model-based Offline Policy Optimization.
<code>d3rlpy.algos.COMBO</code>	Conservative Offline Model-Based Optimization.
<code>d3rlpy.algos.RandomPolicy</code>	Random Policy for continuous control algorithm.

#### `d3rlpy.algos.BC`

```
class d3rlpy.algos.BC(*, learning_rate=0.001,
                      optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam', betas=(0.9,
                      0.999), eps=1e-08, weight_decay=0, amsgrad=False), encoder_factory='default',
                      batch_size=100, n_frames=1, use_gpu=False, scaler=None, action_scaler=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.models.optimizers.OptimizerFactory`) – optimizer factory.
- **encoder\_factory** (`d3rlpy.models.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.preprocessing.Scaler` or `str`) – preprocessor. The available options are ['pixel', 'min\_max', 'standard'].
- **action\_scaler** (`d3rlpy.preprocessing.ActionScaler` or `str`) – action scaler. The available options are ['min\_max'].
- **impl** (`d3rlpy.algos.torch.bc_impl.BCImpl`) – implementation of the algorithm.
- **kwargs** (`Any`) –

### Methods

#### `build_with_dataset(dataset)`

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

#### `build_with_env(env)`

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

#### `collect(env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True)`

Collects data via interaction with environment.

If buffer is not given, ReplayBuffer will be internally created.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_steps** (`int`) – the number of total steps to train.
- **show\_progress** (`bool`) – flag to show progress bar for iterations.

- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.  
truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

**copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

```
fit(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
     experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
     show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
     shuffle=True, callback=None)
```

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when n\_steps is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with eval\_episodes.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step), which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                 n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                 save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                 logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                 timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

### Parameters

- **env** (*d3rlpy.envs.batch.BatchEnv*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.BatchBuffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.

- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (*Optional[gym.core.Env]*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -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.
- **save\_interval** (*int*) – the number of epochs before saving models.
- **experiment\_name** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (*bool*) – flag to turn terminal flag False when TimeLimit.truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

#### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when n\_steps is None.
- **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** (*Optional[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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional*[`List[d3rlpy.dataset.Episode]`]) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (*Optional*[`Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]`]) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (*Optional*[`Callable[[d3rlpy.base.LearnableBase, int, int], None]`]) – callable function that takes (`algo`, `epoch`, `total_step`), which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** `Generator[Tuple[int, Dict[str, float]], None, None]`

**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** (*Optional*[`Union[bool, int, d3rlpy.gpu.Device]`]) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** `d3rlpy.base.LearnableBase`

**generate\_new\_data**(`transitions`)

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** `transitions` (`List[d3rlpy.dataset.Transition]`) – list of transitions.

**Returns** list of new transitions.

**Return type** Optional[List[d3rlpy.dataset.Transition]]

#### get\_action\_type()

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

#### 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[str, Any]

#### load\_model(fname)

Load neural network parameters.

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

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

**Return type** None

#### 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` (Union[numpy.ndarray, List[Any]]) – observations

**Returns** greedy actions

**Return type** numpy.ndarray

#### predict\_value(x, action, with\_std=False)

value prediction is not supported by BC algorithms.

##### Parameters

- `x` (Union[numpy.ndarray, List[Any]]) –

- **action**(Union[numpy.ndarray, List[Any]]) –
- **with\_std**(bool) –

**Return type** numpy.ndarray

**sample\_action**(*x*)

sampling action is not supported by BC algorithm.

**Parameters** **x**(Union[numpy.ndarray, List[Any]]) –

**Return type** None

**save\_model**(*fname*)

Saves neural network parameters.

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

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

**Return type** None

**save\_params**(*logger*)

Saves configurations as params.json.

**Parameters** **logger**(d3rlpy.logger.D3RLPyLogger) – logger object.

**Return type** None

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** None

**set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** **logger**(d3rlpy.logger.D3RLPyLogger) – logger object.

**Return type** None

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

**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` (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

## Attributes

**action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** `Optional[ActionScaler]`

**action\_size**

Action size.

**Returns** action size.

**Return type** `Optional[int]`

**active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

**batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** `int`

**gamma**

Discount factor.

**Returns** discount factor.

**Return type** float

**grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

**impl**

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

**n\_frames**

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

**n\_steps**

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## d3rlpy.algos.DDPG

```
class d3rlpy.algos.DDPG(*, actor_learning_rate=0.0003, critic_learning_rate=0.0003,
                        actor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                        betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                        critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                        betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                        actor_encoder_factory='default', critic_encoder_factory='default',
                        q_func_factory='mean', batch_size=100, n_frames=1, n_steps=1, gamma=0.99,
                        tau=0.005, n_critics=1, target_reduction_type='min', use_gpu=False, scaler=None,
                        action_scaler=None, reward_scaler=None, impl=None, **kwargs)
```

Deep Deterministic Policy Gradients algorithm.

DDPG is an actor-critic algorithm that trains a Q function parametrized with  $\theta$  and a policy function parametrized with  $\phi$ .

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

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

where  $\theta'$  and  $\phi$  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.models.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **actor\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the actor.
- **critic\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the critic.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.

- **tau** (`float`) – target network synchronization coefficient.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **target\_reduction\_type** (`str`) – ensemble reduction method at target value estimation. The available options are ['min', 'max', 'mean', 'mix', 'none'].
- **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']
- **action\_scaler** (`d3rlpy.preprocessing.ActionScaler` or `str`) – action preprocessor. The available options are ['min\_max'].
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler` or `str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.algos.torch.ddpg_impl.DDPGImpl`) – algorithm implementation.
- **kwargs** (`Any`) –

## Methods

**build\_with\_dataset**(`dataset`)

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

**build\_with\_env**(`env`)

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

**collect**(`env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True`)

Collects data via interaction with environment.

If buffer is not given, ReplayBuffer will be internally created.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_steps** (`int`) – the number of total steps to train.
- **show\_progress** (`bool`) – flag to show progress bar for iterations.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag `False` when TimeLimit truncated flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

**copy\_policy\_from**(`algo`)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(`algo`)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

```
fit(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
     experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
     show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
     shuffle=True, callback=None)
```

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.

- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (`Optional[List[d3rlpy.dataset.Episode]]`) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (`Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]`) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (`Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** `List[Tuple[int, Dict[str, float]]]`

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                  n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                  save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                  logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                  timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.BatchBuffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (`Optional[gym.core.Env]`) – gym-like environment. If `None`, evaluation is skipped.

- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when TimeLimit.truncated flag is True, which is designed to incorporate with gym.wrappers.TimeLimit.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes (algo, epoch, total\_step) , which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

#### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag `False` when `TimeLimit`.  
truncated flag is `True`, which is designed to incorporate with `gym.wrappers`.  
`TimeLimit`.
- **callback** (*Optional*[`Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]`]) – callable function that takes (`algo, epoch, total_step`) , which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,  
experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,  
show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,  
shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):  
    my_plot(metrics)  
    algo.save_model(my_path)
```

#### Parameters

- **dataset** (*Union*[`List[d3rlpy.dataset.Episode]`, `d3rlpy.dataset.MDPDataset`]) – list of episodes to train.
- **n\_epochs** (*Optional*[`int`]) – the number of epochs to train.
- **n\_steps** (*Optional*[`int`]) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.

- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

#### **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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

#### **generate\_new\_data(transitions)**

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** Optional[List[d3rlpy.dataset.Transition]]

#### **get\_action\_type()**

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

**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[str, Any]`

**load\_model(fname)**

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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)
```

(continues on next page)

(continued from previous page)

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

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

**Parameters**

- **x** (*Union[`numpy.ndarray`, `List[Any]`]*) – observations
- **action** (*Union[`numpy.ndarray`, `List[Any]`]*) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (*Union[`numpy.ndarray`, `List[Any]`]*) – 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.**Return type** `None`**save\_params(*logger*)**Saves configurations as `params.json`.**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.**Return type** `None`**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

### `set_active_logger(logger)`

Set active D3RLPyLogger object

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

### `set_grad_step(grad_step)`

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

### `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` (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

### `update(batch)`

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

## Attributes

### `action_scaler`

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

### `action_size`

Action size.

**Returns** action size.

**Return type** Optional[int]

### `active_logger`

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

### `batch_size`

Batch size to train.

**Returns** batch size.

**Return type** int

### `gamma`

Discount factor.

**Returns** discount factor.

**Return type** float

### `grad_step`

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

### `impl`

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

### `n_frames`

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

### `n_steps`

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

**d3rlpy.algos.TD3**

```
class d3rlpy.algos.TD3(*, actor_learning_rate=0.0003, critic_learning_rate=0.0003,
                      actor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                      betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                      critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                      betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                      actor_encoder_factory='default', critic_encoder_factory='default',
                      q_func_factory='mean', batch_size=100, n_frames=1, n_steps=1, gamma=0.99,
                      tau=0.005, n_critics=2, target_reduction_type='min', target_smoothing_sigma=0.2,
                      target_smoothing_clip=0.5, update_actor_interval=2, use_gpu=False, scaler=None,
                      action_scaler=None, reward_scaler=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.models.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **actor\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the actor.
- **critic\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the critic.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **tau** (`float`) – target network synchronization coefficient.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **target\_reduction\_type** (`str`) – ensemble reduction method at target value estimation. The available options are ['min', 'max', 'mean', 'mix', 'none'].
- **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.
- **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'].
- **action\_scaler** (`d3rlpy.preprocessing.ActionScaler or str`) – action preprocessor. The available options are ['min\_max'].
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler or str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.algos.torch.td3_impl.TD3Impl`) – algorithm implementation.
- **kwargs** (`Any`) –

## Methods

### `build_with_dataset(dataset)`

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset (d3rlpy.dataset.MDPDataset)` – dataset.

**Return type** `None`

### `build_with_env(env)`

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env (gym.core.Env)` – gym-like environment.

**Return type** `None`

### `collect(env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True)`

Collects data via interaction with environment.

If `buffer` is not given, `ReplayBuffer` will be internally created.

**Parameters**

- `env (gym.core.Env)` – gym-like environment.
- `buffer (Optional[d3rlpy.online.buffers.Buffer])` – replay buffer.
- `explorer (Optional[d3rlpy.online.explorers.Explorer])` – action explorer.
- `n_steps (int)` – the number of total steps to train.
- `show_progress (bool)` – flag to show progress bar for iterations.
- `timelimit_aware (bool)` – flag to turn `terminal` flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

### `copy_policy_from(algo)`

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo (d3rlpy.algos.base.AlgoBase)` – algorithm object.

**Return type** `None`

### `copy_q_function_from(algo)`

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

**fit**(`dataset`, `n_epochs=None`, `n_steps=None`, `n_steps_per_epoch=10000`, `save_metrics=True`, `experiment_name=None`, `with_timestamp=True`, `logdir='d3rlpy_logs'`, `verbose=True`, `show_progress=True`, `tensorboard_dir=None`, `eval_episodes=None`, `save_interval=1`, `scorers=None`, `shuffle=True`, `callback=None`)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.

- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                  n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                  save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                  logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                  timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (*d3rlpy.envs.batch.BatchEnv*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.BatchBuffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_epochs** (*int*) – the number of epochs to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (*int*) – the number of updates per epoch.
- **eval\_interval** (*int*) – the number of epochs before evaluation.
- **eval\_env** (*Optional[gym.core.Env]*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -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.
- **save\_interval** (*int*) – the number of epochs before saving models.
- **experiment\_name** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.

- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when TimeLimit. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when TimeLimit. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each iteration algo methods and properties can be changed or queried.**

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when n\_steps is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with eval\_episodes.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

```
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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

### generate\_new\_data(*transitions*)

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** *Optional[List[d3rlpy.dataset.Transition]]*

### get\_action\_type()

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

### 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[str, Any]

### load\_model(*fname*)

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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` (`Union[numpy.ndarray, List[Any]]`) – observations
- `action` (`Union[numpy.ndarray, List[Any]]`) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (*Union[`numpy.ndarray`, `List[Any]`]*) – 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.

**Return type** `None`

**save\_params(*logger*)**

Saves configurations as params.json.

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger(*logger*)**

Set active D3RLPyLogger object

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**set\_grad\_step(*grad\_step*)**

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

**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` (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update(*batch*)**

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

**Attributes****action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** `Optional[ActionScaler]`

**action\_size**

Action size.

**Returns** action size.

**Return type** `Optional[int]`

**active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

**batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** `int`

**gamma**

Discount factor.

**Returns** discount factor.

**Return type** float

**grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

**impl**

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

**n\_frames**

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

**n\_steps**

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## 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.models.optimizers.AdamFactory(optim_cls='Adam',
                      betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                      critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                      betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                      temp_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                      betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                      actor_encoder_factory='default', critic_encoder_factory='default',
                      q_func_factory='mean', batch_size=100, n_frames=1, n_steps=1, gamma=0.99,
                      tau=0.005, n_critics=2, target_reduction_type='min', initial_temperature=1.0,
                      use_gpu=False, scaler=None, action_scaler=None, reward_scaler=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.

$$\begin{aligned} L(\theta_i) &= \mathbb{E}_{s_t, a_t, r_{t+1}, s_{t+1} \sim D, a_{t+1} \sim \pi_\phi(\cdot | s_{t+1})} \left[ (y - Q_{\theta_i}(s_t, a_t))^2 \right] \\ y &= r_{t+1} + \gamma \left( \min_j Q_{\theta_j}(s_{t+1}, a_{t+1}) - \alpha \log (\pi_\phi(a_{t+1} | s_{t+1})) \right) \\ J(\phi) &= \mathbb{E}_{s_t \sim D, a_t \sim \pi_\phi(\cdot | s_t)} \left[ \alpha \log(\pi_\phi(a_t | s_t)) - \min_i Q_{\theta_i}(s_t, \pi_\phi(a_t | s_t)) \right] \end{aligned}$$

The temperature parameter  $\alpha$  is also automatically adjustable.

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

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.models.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **temp\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the temperature.

- **actor\_encoder\_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the actor.
- **critic\_encoder\_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the critic.
- **q\_func\_factory** (*d3rlpy.models.q\_functions.QFunctionFactory or str*) – Q function factory.
- **batch\_size** (*int*) – mini-batch size.
- **n\_frames** (*int*) – the number of frames to stack for image observation.
- **n\_steps** (*int*) – N-step TD calculation.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficient.
- **n\_critics** (*int*) – the number of Q functions for ensemble.
- **target\_reduction\_type** (*str*) – ensemble reduction method at target value estimation. The available options are ['min', 'max', 'mean', 'mix', 'none'].
- **initial\_temperature** (*float*) – initial temperature value.
- **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'].
- **action\_scaler** (*d3rlpy.preprocessing.ActionScaler or str*) – action preprocessor. The available options are ['min\_max'].
- **reward\_scaler** (*d3rlpy.preprocessing.RewardScaler or str*) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (*d3rlpy.algos.torch.sac\_impl.SACImpl*) – algorithm implementation.
- **kwargs** (*Any*) –

## Methods

**build\_with\_dataset**(*dataset*)

Instantiate implementation object with MDPDataset object.

**Parameters** **dataset** (*d3rlpy.dataset.MDPDataset*) – dataset.

**Return type** *None*

**build\_with\_env**(*env*)

Instantiate implementation object with OpenAI Gym object.

**Parameters** **env** (*gym.core.Env*) – gym-like environment.

**Return type** *None*

**collect**(*env, buffer=None, explorer=None, n\_steps=1000000, show\_progress=True, timelimit\_aware=True*)

Collects data via interaction with environment.

If buffer is not given, ReplayBuffer will be internally created.

**Parameters**

- **env** (*gym.core.Env*) – gym-like environment.

- **buffer** (*Optional[d3rlpy.online.buffers.Buffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_steps** (*int*) – the number of total steps to train.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.
- **timelimit\_aware** (*bool*) – flag to turn terminal flag `False` when `TimeLimit`.  
truncated flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

**copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (*d3rlpy.algos.base.AlgoBase*) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (*d3rlpy.algos.base.AlgoBase*) – algorithm object.

**Return type** `None`

**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** (*Sequence[int]*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**Return type** `None`

```
fit(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
     experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
     show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
     shuffle=True, callback=None)
Trains with the given dataset.
```

```
algo.fit(episodes, n_steps=1000000)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when n\_steps is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with eval\_episodes.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step), which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                 n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                 save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                 logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                 timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.BatchBuffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

#### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (*Optional* [`gym.core.Env`]) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (*Optional* [`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\_dir** (*Optional* [`str`]) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn `terminal` flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional* [`Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]`]) – callable function that takes `(algo, epoch, total_step)` , which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,  
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,  
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,  
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):  
    my_plot(metrics)  
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union* [`List[d3rlpy.dataset.Episode]`, `d3rlpy.dataset.MDPDataset`]) – list of episodes to train.
- **n\_epochs** (*Optional* [`int`]) – the number of epochs to train.
- **n\_steps** (*Optional* [`int`]) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is None.

- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional*[`List[d3rlpy.dataset.Episode]`]) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (*Optional*[`Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]`]) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (*Optional*[`Callable[[d3rlpy.base.LearnableBase, int, int], None]`]) – callable function that takes `(algo, epoch, total_step)` , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

**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** (*Optional*[`Union[bool, int, d3rlpy.gpu.Device]`]) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** `d3rlpy.base.LearnableBase`

**generate\_new\_data(transitions)**

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** `transitions` (`List[d3rlpy.dataset.Transition]`) – list of transitions.

**Returns** list of new transitions.

**Return type** `Optional[List[d3rlpy.dataset.Transition]]`

**get\_action\_type()**

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** `d3rlpy.constants.ActionSpace`

**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[str, Any]`

**load\_model(fname)**

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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** (`Union[numpy.ndarray, List[Any]]`) – observations
- **action** (`Union[numpy.ndarray, List[Any]]`) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (`Union[numpy.ndarray, List[Any]]`) – 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.

**Return type** `None`

**save\_params**(*logger*)

Saves configurations as `params.json`.

**Parameters** **logger** (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** **grad\_step** (*int*) – total gradient step counter.

**Return type** `None`

**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** (*Any*) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** Dict[str, float]

## Attributes

### action\_scaler

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

### action\_size

Action size.

**Returns** action size.

**Return type** Optional[int]

### active\_logger

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

### batch\_size

Batch size to train.

**Returns** batch size.

**Return type** int

### gamma

Discount factor.

**Returns** discount factor.

**Return type** float

### grad\_step

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

### impl

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

### n\_frames

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

### n\_steps

N-step TD backup.

**Returns** N-step TD backup.

**Return type** `int`

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** `Optional[Sequence[int]]`

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** `Optional[RewardScaler]`

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** `Optional[Scaler]`

## d3rlpy.algos.BCQ

```
class d3rlpy.algos.BCQ(*, actor_learning_rate=0.001, critic_learning_rate=0.001,
                      imitator_learning_rate=0.001,
                      actor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                      betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                      critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                      betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                      imitator_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                      betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                      actor_encoder_factory='default', critic_encoder_factory='default',
                      imitator_encoder_factory='default', q_func_factory='mean', batch_size=100,
                      n_frames=1, n_steps=1, gamma=0.99, tau=0.005, n_critics=2,
                      update_actor_interval=1, lam=0.75, n_action_samples=100, action_flexibility=0.05,
                      rl_start_step=0, latent_size=32, beta=0.5, use_gpu=False, scaler=None,
                      action_scaler=None, reward_scaler=None, impl=None, **kwargs)
```

Batch-Constrained Q-learning algorithm.

BCQ is the very first practical data-driven deep reinforcement learning lgorithm. 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  $E_\omega$  and  $D_\omega$  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  $\Phi$  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.models.optimizers.OptimizerFactory*) – optimizer factory for the actor.
- **critic\_optim\_factory** (*d3rlpy.models.optimizers.OptimizerFactory*) – optimizer factory for the critic.
- **imitator\_optim\_factory** (*d3rlpy.models.optimizers.OptimizerFactory*) – optimizer factory for the conditional VAE.
- **actor\_encoder\_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the actor.
- **critic\_encoder\_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the critic.
- **imitator\_encoder\_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the conditional VAE.
- **q\_func\_factory** (*d3rlpy.models.q\_functions.QFunctionFactory or str*) – Q function factory.
- **batch\_size** (*int*) – mini-batch size.
- **n\_frames** (*int*) – the number of frames to stack for image observation.
- **n\_steps** (*int*) – N-step TD calculation.

- **gamma** (`float`) – discount factor.
- **tau** (`float`) – target network synchronization coefficient.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **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  $\Phi$ .
- **rl\_start\_step** (`int`) – step 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.
- **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'].
- **action\_scaler** (`d3rlpy.preprocessing.ActionScaler or str`) – action preprocessor. The available options are ['min\_max'].
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler or str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.algos.torch.bcq_impl.BCQImpl`) – algorithm implementation.
- **kwargs** (`Any`) –

## Methods

**build\_with\_dataset**(`dataset`)

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

**build\_with\_env**(`env`)

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

**collect**(`env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True`)

Collects data via interaction with environment.

If `buffer` is not given, `ReplayBuffer` will be internally created.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_steps** (`int`) – the number of total steps to train.

- **show\_progress** (`bool`) – flag to show progress bar for iterations.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag `False` when `TimeLimit`.  
truncated flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

#### `copy_policy_from(algo)`

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

#### `copy_q_function_from(algo)`

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

#### `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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

```
fit(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,  
     experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,  
     show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,  
     shuffle=True, callback=None)  
Trains with the given dataset.
```

```
algo.fit(episodes, n_steps=1000000)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when n\_steps is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with eval\_episodes.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step), which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,  
                  n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,  
                  save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,  
                  logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,  
                  timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.BatchBuffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

#### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (*Optional* [`gym.core.Env`]) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (*Optional* [`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\_dir** (*Optional* [`str`]) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional* [`Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]`]) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union* [`List[d3rlpy.dataset.Episode]`, `d3rlpy.dataset.MDPDataset`]) – list of episodes to train.
- **n\_epochs** (*Optional* [`int`]) – the number of epochs to train.
- **n\_steps** (*Optional* [`int`]) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is None.

- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional*[`List[d3rlpy.dataset.Episode]`]) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (*Optional*[`Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]`]) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (*Optional*[`Callable[[d3rlpy.base.LearnableBase, int, int], None]`]) – callable function that takes `(algo, epoch, total_step)` , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

#### 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** (*Optional*[`Union[bool, int, d3rlpy.gpu.Device]`]) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** `d3rlpy.base.LearnableBase`

**generate\_new\_data(transitions)**

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** `transitions` (`List[d3rlpy.dataset.Transition]`) – list of transitions.

**Returns** list of new transitions.

**Return type** `Optional[List[d3rlpy.dataset.Transition]]`

**get\_action\_type()**

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** `d3rlpy.constants.ActionSpace`

**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[str, Any]`

**load\_model(fname)**

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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** (`Union[numpy.ndarray, List[Any]]`) – observations
- **action** (`Union[numpy.ndarray, List[Any]]`) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, numpy.ndarray]]`

**sample\_action**(*x*)

BCQ does not support sampling action.

**Parameters** `x` (`Union[numpy.ndarray, List[Any]]`) –

**Return type** `numpy.ndarray`

**save\_model**(*fname*)

Saves neural network parameters.

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

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

**Return type** `None`

**save\_params**(*logger*)

Saves configurations as `params.json`.

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** **logger** (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** **grad\_step** (`int`) – total gradient step counter.

**Return type** `None`

**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** (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

---

**Return type** Dict[str, float]

## Attributes

### action\_scaler

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

### action\_size

Action size.

**Returns** action size.

**Return type** Optional[int]

### active\_logger

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

### batch\_size

Batch size to train.

**Returns** batch size.

**Return type** int

### gamma

Discount factor.

**Returns** discount factor.

**Return type** float

### grad\_step

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

### impl

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

### n\_frames

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

### n\_steps

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int  
**observation\_shape**  
Observation shape.  
**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**  
Preprocessing reward scaler.  
**Returns** preprocessing reward scaler.  
**Return type** Optional[RewardScaler]

**scaler**  
Preprocessing scaler.  
**Returns** preprocessing scaler.  
**Return type** Optional[Scaler]

## d3rlpy.algos.BEAR

```
class d3rlpy.algos.BEAR(*, actor_learning_rate=0.0001, critic_learning_rate=0.0003,
                        imitator_learning_rate=0.0003, temp_learning_rate=0.0001,
                        alpha_learning_rate=0.001,
                        actor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                        betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                        critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                        betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                        imitator_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                        betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                        temp_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                        betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                        alpha_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                        betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                        actor_encoder_factory='default', critic_encoder_factory='default',
                        imitator_encoder_factory='default', q_func_factory='mean', batch_size=256,
                        n_frames=1, n_steps=1, gamma=0.99, tau=0.005, n_critics=2,
                        initial_temperature=1.0, initial_alpha=1.0, alpha_threshold=0.05, lam=0.75,
                        n_action_samples=100, n_target_samples=10, n_mmd_action_samples=4,
                        mmd_kernel='laplacian', mmd_sigma=20.0, vae_kl_weight=0.5,
                        warmup_steps=40000, use_gpu=False, scaler=None, action_scaler=None,
                        reward_scaler=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  $k(x, y) = \exp((x - y)^2 / (2\sigma^2))$ .

$\alpha$  is also adjustable through dual gradient descent where  $\alpha$  becomes smaller if MMD is smaller than the threshold  $\epsilon$ .

## 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  $\alpha$ .
- **actor\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **imitator\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the behavior policy.
- **temp\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the temperature.
- **alpha\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for  $\alpha$ .
- **actor\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the actor.
- **critic\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the critic.
- **imitator\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the behavior policy.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **tau** (`float`) – target network synchronization coefficient.
- **n\_critics** (`int`) – the number of Q functions for ensemble.

- **initial\_temperature** (`float`) – initial temperature value.
- **initial\_alpha** (`float`) – initial  $\alpha$  value.
- **alpha\_threshold** (`float`) – threshold value described as  $\epsilon$ .
- **lam** (`float`) – weight for critic ensemble.
- **n\_action\_samples** (`int`) – the number of action samples to compute the best action.
- **n\_target\_samples** (`int`) – the number of action samples to compute BCQ-like target value.
- **n\_mmd\_action\_samples** (`int`) – the number of action samples to compute MMD.
- **mmd\_kernel** (`str`) – MMD kernel function. The available options are ['gaussian', 'laplacian'].
- **mmd\_sigma** (`float`) –  $\sigma$  for gaussian kernel in MMD calculation.
- **vae\_kl\_weight** (`float`) – constant weight to scale KL term for behavior policy training.
- **warmup\_steps** (`int`) – the number of steps to warmup the policy function.
- **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'].
- **action\_scaler** (`d3rlpy.preprocessing.ActionScaler` or `str`) – action preprocessor. The available options are ['min\_max'].
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler` or `str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.algos.torch.bear_impl.BEARImpl`) – algorithm implementation.
- **kwargs** (`Any`) –

## Methods

**build\_with\_dataset**(`dataset`)

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

**build\_with\_env**(`env`)

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

**collect**(`env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True`)

Collects data via interaction with environment.

If buffer is not given, ReplayBuffer will be internally created.

**Parameters**

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

- **n\_steps** (`int`) – the number of total steps to train.
- **show\_progress** (`bool`) – flag to show progress bar for iterations.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag `False` when `TimeLimit.truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

#### `copy_policy_from(algo)`

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

#### `copy_q_function_from(algo)`

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

#### `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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

```
fit(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,  
     experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,  
     show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,  
     shuffle=True, callback=None)  
Trains with the given dataset.
```

```
algo.fit(episodes, n_steps=1000000)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when n\_steps is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with eval\_episodes.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step), which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,  
                  n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,  
                  save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,  
                  logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,  
                  timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.BatchBuffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

#### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (*Optional* [`gym.core.Env`]) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (*Optional* [`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\_dir** (*Optional* [`str`]) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn `terminal` flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional* [`Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]`]) – callable function that takes `(algo, epoch, total_step)` , which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,  
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,  
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,  
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):  
    my_plot(metrics)  
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union* [`List[d3rlpy.dataset.Episode]`, `d3rlpy.dataset.MDPDataset`]) – list of episodes to train.
- **n\_epochs** (*Optional* [`int`]) – the number of epochs to train.
- **n\_steps** (*Optional* [`int`]) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is None.

- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional*[`List[d3rlpy.dataset.Episode]`]) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (*Optional*[`Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]`]) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (*Optional*[`Callable[[d3rlpy.base.LearnableBase, int, int], None]`]) – callable function that takes `(algo, epoch, total_step)` , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

**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** (*Optional*[`Union[bool, int, d3rlpy.gpu.Device]`]) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** `d3rlpy.base.LearnableBase`

**generate\_new\_data(transitions)**

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** `transitions` (`List[d3rlpy.dataset.Transition]`) – list of transitions.

**Returns** list of new transitions.

**Return type** `Optional[List[d3rlpy.dataset.Transition]]`

**get\_action\_type()**

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** `d3rlpy.constants.ActionSpace`

**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[str, Any]`

**load\_model(fname)**

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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** (`Union[numpy.ndarray, List[Any]]`) – observations
- **action** (`Union[numpy.ndarray, List[Any]]`) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (`Union[numpy.ndarray, List[Any]]`) – 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.

**Return type** `None`

**save\_params**(*logger*)

Saves configurations as `params.json`.

**Parameters** **logger** (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** **grad\_step** (*int*) – total gradient step counter.

**Return type** `None`

**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** (*Any*) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

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

---

**Returns** dictionary of metrics.

**Return type** Dict[str, float]

## Attributes

### action\_scaler

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

### action\_size

Action size.

**Returns** action size.

**Return type** Optional[int]

### active\_logger

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

### batch\_size

Batch size to train.

**Returns** batch size.

**Return type** int

### gamma

Discount factor.

**Returns** discount factor.

**Return type** float

### grad\_step

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

### impl

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

### n\_frames

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

### n\_steps

N-step TD backup.

**Returns** N-step TD backup.

**Return type** `int`

**observation\_shape**  
Observation shape.

**Returns** observation shape.

**Return type** `Optional[Sequence[int]]`

**reward\_scaler**  
Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** `Optional[RewardScaler]`

**scaler**  
Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** `Optional[Scaler]`

## d3rlpy.algos.CRR

```
class d3rlpy.algos.CRR(*, actor_learning_rate=0.0003, critic_learning_rate=0.0003,
                      actor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                      betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                      critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                      betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                      actor_encoder_factory='default', critic_encoder_factory='default',
                      q_func_factory='mean', batch_size=100, n_frames=1, n_steps=1, gamma=0.99,
                      beta=1.0, n_action_samples=4, advantage_type='mean', weight_type='exp',
                      max_weight=20.0, n_critics=1, target_update_interval=100,
                      target_reduction_type='min', update_actor_interval=1, use_gpu=False, scaler=None,
                      action_scaler=None, reward_scaler=None, impl=None, **kwargs)
```

Critic Regularized Regression algorithm.

CRR is a simple offline RL method similar to AWAC.

The policy is trained as a supervised regression.

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

where  $f$  is a filter function which has several options. The first option is `binary` function.

$$f := \mathbb{1}[A_\theta(s, a) > 0]$$

The other is `exp` function.

$$f := \exp(A(s, a) / \beta)$$

The  $A(s, a)$  is an average function which also has several options. The first option is `mean`.

$$A(s, a) = Q_\theta(s, a) - \frac{1}{m} \sum_j^m Q(s, a_j)$$

The other one is `max`.

$$A(s, a) = Q_\theta(s, a) - \max_j^m Q(s, a_j)$$

where  $a_j \sim \pi_\phi(s)$ .

In evaluation, the action is determined by Critic Weighted Policy (CWP). In CWP, the several actions are sampled from the policy function, and the final action is re-sampled from the estimated action-value distribution.

## References

- Wang et al., Critic Regularized Regression.

### Parameters

- `actor_learning_rate` (`float`) – learning rate for policy function.
- `critic_learning_rate` (`float`) – learning rate for Q functions.
- `actor_optim_factory` (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- `critic_optim_factory` (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- `actor_encoder_factory` (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the actor.
- `critic_encoder_factory` (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the critic.
- `q_func_factory` (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.
- `batch_size` (`int`) – mini-batch size.
- `n_frames` (`int`) – the number of frames to stack for image observation.
- `n_steps` (`int`) – N-step TD calculation.
- `gamma` (`float`) – discount factor.
- `beta` (`float`) – temperature value defined as  $\beta$  above.
- `n_action_samples` (`int`) – the number of sampled actions to calculate  $A(s, a)$  and for CWP.
- `advantage_type` (`str`) – advantage function type. The available options are `['mean', 'max']`.
- `weight_type` (`str`) – filter function type. The available options are `['binary', 'exp']`.
- `max_weight` (`float`) – maximum weight for cross-entropy loss.
- `n_critics` (`int`) – the number of Q functions for ensemble.
- `target_reduction_type` (`str`) – ensemble reduction method at target value estimation. The available options are `['min', 'max', 'mean', 'mix', 'none']`.
- `update_actor_interval` (`int`) – interval to update policy function.
- `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']
- **action\_scaler** (*d3rlpy.preprocessing.ActionScaler or str*) – action preprocessor. The available options are ['min\_max'].
- **reward\_scaler** (*d3rlpy.preprocessing.RewardScaler or str*) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (*d3rlpy.algos.torch.crr\_impl.CRRImp1*) – algorithm implementation.
- **target\_update\_interval** (*int*) –
- **kwargs** (*Any*) –

## Methods

**build\_with\_dataset**(*dataset*)

Instantiate implementation object with MDPDataset object.

**Parameters** **dataset** (*d3rlpy.dataset.MDPDataset*) – dataset.

**Return type** *None*

**build\_with\_env**(*env*)

Instantiate implementation object with OpenAI Gym object.

**Parameters** **env** (*gym.core.Env*) – gym-like environment.

**Return type** *None*

**collect**(*env, buffer=None, explorer=None, n\_steps=1000000, show\_progress=True, timelimit\_aware=True*)

Collects data via interaction with environment.

If buffer is not given, ReplayBuffer will be internally created.

**Parameters**

- **env** (*gym.core.Env*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.Buffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_steps** (*int*) – the number of total steps to train.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.
- **timelimit\_aware** (*bool*) – flag to turn terminal flag *False* when *TimeLimit*.truncated flag is *True*, which is designed to incorporate with *gym.wrappers.TimeLimit*.

**Returns** replay buffer with the collected data.

**Return type** *d3rlpy.online.buffers.Buffer*

**copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
```

(continues on next page)

(continued from previous page)

```
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(`algo`)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

**fit**(`dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True, experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None, shuffle=True, callback=None`)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.

- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional*[`List[d3rlpy.dataset.Episode]`]) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (*Optional*[`Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]`]) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (*Optional*[`Callable[[d3rlpy.base.LearnableBase, int, int], None]`]) – callable function that takes `(algo, epoch, total_step)` , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** `List[Tuple[int, Dict[str, float]]]`

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                 n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                 save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                 logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                 timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (*Optional*[`d3rlpy.online.buffers.BatchBuffer`]) – replay buffer.
- **explorer** (*Optional*[`d3rlpy.online.explorers.Explorer`]) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (*Optional*[`gym.core.Env`]) – gym-like environment. If `None`, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

#### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn `terminal` flag `False` when `TimeLimit`.`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional*[`Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]`]) – callable function that takes (`algo`, `epoch`, `total_step`) , which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,  
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,  
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,  
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):  
    my_plot(metrics)  
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union*[`List[d3rlpy.dataset.Episode]`, `d3rlpy.dataset.MDPDataset`]) – list of episodes to train.
- **n\_epochs** (*Optional*[`int`]) – the number of epochs to train.
- **n\_steps** (*Optional*[`int`]) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional*[`List[d3rlpy.dataset.Episode]`]) – list of episodes to test.

- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (`Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]`) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (`Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]`) – callable function that takes (`algo`, `epoch`, `total_step`) , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

#### `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** (`Optional[Union[bool, int, d3rlpy.gpu.Device]]`) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** `d3rlpy.base.LearnableBase`

#### `generate_new_data(transitions)`

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** `transitions` (`List[d3rlpy.dataset.Transition]`) – list of transitions.

**Returns** list of new transitions.

**Return type** `Optional[List[d3rlpy.dataset.Transition]]`

#### `get_action_type()`

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** `d3rlpy.constants.ActionSpace`

#### `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[str, Any]`

**load\_model** (`fname`)

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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,)
```

(continues on next page)

(continued from previous page)

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

**Parameters**

- **x** (`Union[numpy.ndarray, List\[Any\]]`) – observations
- **action** (`Union[numpy.ndarray, List\[Any\]]`) – 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** `Union[numpy.ndarray, Tuple\[numpy.ndarray, numpy.ndarray\]\]`**sample\_action(x)**

Returns sampled actions.

The sampled actions are identical to the output of `predict` method if the policy is deterministic.**Parameters** **x** (`Union[numpy.ndarray, List\[Any\]]`) – 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.**Return type** `None`**save\_params(logger)**Saves configurations as `params.json`.**Parameters** **logger** (`d3rlpy.logger.D3RLPyLogger`) – logger object.**Return type** `None`**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).

- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** *None*

### **set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** *None*

### **set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** **grad\_step** (*int*) – total gradient step counter.

**Return type** *None*

### **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** (*Any*) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** *d3rlpy.base.LearnableBase*

### **update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** *Dict[str, float]*

---

## Attributes

### `action_scaler`

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

### `action_size`

Action size.

**Returns** action size.

**Return type** Optional[int]

### `active_logger`

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

### `batch_size`

Batch size to train.

**Returns** batch size.

**Return type** int

### `gamma`

Discount factor.

**Returns** discount factor.

**Return type** float

### `grad_step`

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

### `impl`

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

### `n_frames`

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

### `n_steps`

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

**d3rlpy.algos.CQL**

```
class d3rlpy.algos.CQL(*, actor_learning_rate=0.0001, critic_learning_rate=0.0003,
                       temp_learning_rate=0.0001, alpha_learning_rate=0.0001,
                       actor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                       betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                       critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                       betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                       temp_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                       betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                       alpha_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                       betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                       actor_encoder_factory='default', critic_encoder_factory='default',
                       q_func_factory='mean', batch_size=256, n_frames=1, n_steps=1, gamma=0.99,
                       tau=0.005, n_critics=2, target_reduction_type='min', initial_temperature=1.0,
                       initial_alpha=1.0, alpha_threshold=10.0, conservative_weight=5.0,
                       n_action_samples=10, soft_q_backup=False, use_gpu=False, scaler=None,
                       action_scaler=None, reward_scaler=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.

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

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

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

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

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  $\alpha$ .
- **actor\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **temp\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the temperature.
- **alpha\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for  $\alpha$ .
- **actor\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the actor.
- **critic\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the critic.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **tau** (`float`) – target network synchronization coefficient.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **target\_reduction\_type** (`str`) – ensemble reduction method at target value estimation. The available options are `['min', 'max', 'mean', 'mix', 'none']`.
- **initial\_temperature** (`float`) – initial temperature value.
- **initial\_alpha** (`float`) – initial  $\alpha$  value.
- **alpha\_threshold** (`float`) – threshold value described as  $\tau$ .
- **conservative\_weight** (`float`) – constant weight to scale conservative loss.
- **n\_action\_samples** (`int`) – the number of sampled actions to compute  $\log \sum_a \exp Q(s, a)$ .
- **soft\_q\_backup** (`bool`) – flag to use SAC-style backup.

- **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']`.
- **action\_scaler** (`d3rlpy.preprocessing.ActionScaler` or `str`) – action preprocessor. The available options are `['min_max']`.
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler` or `str`) – reward preprocessor. The available options are `['clip', 'min_max', 'standard']`.
- **impl** (`d3rlpy.algos.torch.cql_impl.CQLImpl`) – algorithm implementation.
- **kwargs** (`Any`) –

## Methods

### `build_with_dataset(dataset)`

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

### `build_with_env(env)`

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

### `collect(env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True)`

Collects data via interaction with environment.

If `buffer` is not given, `ReplayBuffer` will be internally created.

**Parameters**

- `env` (`gym.core.Env`) – gym-like environment.
- `buffer` (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- `explorer` (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- `n_steps` (`int`) – the number of total steps to train.
- `show_progress` (`bool`) – flag to show progress bar for iterations.
- `timelimit_aware` (`bool`) – flag to turn terminal flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

### `copy_policy_from(algo)`

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
```

(continues on next page)

(continued from previous page)

```
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

**fit**(*dataset*, *n\_epochs=None*, *n\_steps=None*, *n\_steps\_per\_epoch=10000*, *save\_metrics=True*, *experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*, *show\_progress=True*, *tensorboard\_dir=None*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*, *shuffle=True*, *callback=None*)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when *n\_steps* is `None`.

- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional*[`List[d3rlpy.dataset.Episode]`]) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (*Optional*[`Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]`]) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (*Optional*[`Callable[[d3rlpy.base.LearnableBase, int, int], None]`]) – callable function that takes `(algo, epoch, total_step)` , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** `List[Tuple[int, Dict[str, float]]]`

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                 n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                 save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                 logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                 timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (*Optional*[`d3rlpy.online.buffers.BatchBuffer`]) – replay buffer.
- **explorer** (*Optional*[`d3rlpy.online.explorers.Explorer`]) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (*Optional*[`gym.core.Env`]) – gym-like environment. If `None`, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

#### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn `terminal` flag `False` when `TimeLimit`.`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional*[`Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]`]) – callable function that takes (`algo`, `epoch`, `total_step`) , which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,  
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,  
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,  
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):  
    my_plot(metrics)  
    algo.save_model(my_path)
```

#### Parameters

- **dataset** (*Union*[`List[d3rlpy.dataset.Episode]`, `d3rlpy.dataset.MDPDataset`]) – list of episodes to train.
- **n\_epochs** (*Optional*[`int`]) – the number of epochs to train.
- **n\_steps** (*Optional*[`int`]) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional*[`List[d3rlpy.dataset.Episode]`]) – list of episodes to test.

- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (`Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]`) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (`Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]`) – callable function that takes (`algo`, `epoch`, `total_step`) , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

**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** (`Optional[Union[bool, int, d3rlpy.gpu.Device]]`) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** `d3rlpy.base.LearnableBase`

**generate\_new\_data(transitions)**

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** `transitions` (`List[d3rlpy.dataset.Transition]`) – list of transitions.

**Returns** list of new transitions.

**Return type** `Optional[List[d3rlpy.dataset.Transition]]`

**get\_action\_type()**

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** `d3rlpy.constants.ActionSpace`

**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[str, Any]

**load\_model**(*fname*)

Load neural network parameters.

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

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

**Return type** None

**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 (Union[numpy.ndarray, List[Any]])` – 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,)
```

(continues on next page)

(continued from previous page)

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

**Parameters**

- **x** (*Union[`numpy.ndarray`, `List[Any]`]*) – observations
- **action** (*Union[`numpy.ndarray`, `List[Any]`]*) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (*Union[`numpy.ndarray`, `List[Any]`]*) – 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.**Return type** `None`**save\_params(*logger*)**Saves configurations as `params.json`.**Parameters** **logger** (`d3rlpy.logger.D3RLPyLogger`) – logger object.**Return type** `None`**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).

- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** *None*

### **set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** *None*

### **set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** **grad\_step** (*int*) – total gradient step counter.

**Return type** *None*

### **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** (*Any*) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** *d3rlpy.base.LearnableBase*

### **update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** *Dict[str, float]*

---

## Attributes

**action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

**action\_size**

Action size.

**Returns** action size.

**Return type** Optional[int]

**active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

**batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** int

**gamma**

Discount factor.

**Returns** discount factor.

**Return type** float

**grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

**impl**

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

**n\_frames**

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

**n\_steps**

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## d3rlpy.algos.AWR

```
class d3rlpy.algos.AWR(*, actor_learning_rate=5e-05, critic_learning_rate=0.0001,
                      actor_optim_factory=d3rlpy.models.optimizers.SGDFactory(optim_cls='SGD',
                      momentum=0.9, dampening=0, weight_decay=0, nesterov=False),
                      critic_optim_factory=d3rlpy.models.optimizers.SGDFactory(optim_cls='SGD',
                      momentum=0.9, dampening=0, weight_decay=0, nesterov=False),
                      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, action_scaler=None, reward_scaler=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  $\text{TD}(\lambda)$  to mitigate high variance issue.

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

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

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.models.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **actor\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the actor.
- **critic\_encoder\_factory** (`d3rlpy.models.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`) –  $\lambda$  for TD( $\lambda$ ).
- **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'].
- **action\_scaler** (`d3rlpy.preprocessing.ActionScaler or str`) – action preprocessor. The available options are ['min\_max'].
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler or str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.algos.torch.awr_impl.AWRImpl`) – algorithm implementation.
- **kwargs** (`Any`) –

## Methods

**build\_with\_dataset**(`dataset`)

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

**build\_with\_env**(`env`)

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

**collect**(`env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True`)

Collects data via interaction with environment.

If `buffer` is not given, ReplayBuffer will be internally created.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_steps** (`int`) – the number of total steps to train.
- **show\_progress** (`bool`) – flag to show progress bar for iterations.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag `False` when `TimeLimit`.  
truncated flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

**copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.

- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

```
fit(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
     experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
     show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
     shuffle=True, callback=None)
Trains with the given dataset.
```

```
algo.fit(episodes, n_steps=1000000)
```

### Parameters

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (`Optional[List[d3rlpy.dataset.Episode]]`) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (`Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]`) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (`Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** `List[Tuple[int, Dict[str, float]]]`

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                 n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                 save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                 logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                 timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.BatchBuffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

#### Return type `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
            logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
            timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

## Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

## Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when **n\_steps** is *None*.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if *None*, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (*algo, epoch, total\_step*), which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

#### 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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

#### generate\_new\_data(*transitions*)

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** *Optional[List[d3rlpy.dataset.Transition]]*

#### get\_action\_type()

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

#### 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[str, Any]

#### load\_model(*fname*)

Load neural network parameters.

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

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

**Return type** None

#### 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` (`Union[ndarray, List[Any]]`) – observations

**Returns** greedy actions

**Return type** `numpy.ndarray`

**predict\_value**(`x`, \*`args`, \*\*`kwargs`)

Returns predicted state values.

**Parameters**

- `x` (`Union[ndarray, List[Any]]`) – observations.
- `args` (`Any`) –
- `kwargs` (`Any`) –

**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` (`Union[ndarray, List[Any]]`) – 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.

**Return type** `None`

**save\_params**(`logger`)

Saves configurations as `params.json`.

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

### `set_active_logger(logger)`

Set active D3RLPyLogger object

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

### `set_grad_step(grad_step)`

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

### `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` (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

### `update(batch)`

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

## Attributes

### `action_scaler`

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

### `action_size`

Action size.

**Returns** action size.

**Return type** Optional[int]

### `active_logger`

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

### `batch_size`

Batch size to train.

**Returns** batch size.

**Return type** int

### `gamma`

Discount factor.

**Returns** discount factor.

**Return type** float

### `grad_step`

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

### `impl`

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

### `n_frames`

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

### `n_steps`

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

**d3rlpy.algos.AWAC**

```
class d3rlpy.algos.AWAC(*, actor_learning_rate=0.0003, critic_learning_rate=0.0003,
                        actor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                        betas=(0.9, 0.999), eps=1e-08, weight_decay=0.0001, amsgrad=False),
                        critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                        betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                        actor_encoder_factory='default', critic_encoder_factory='default',
                        q_func_factory='mean', batch_size=1024, n_frames=1, n_steps=1, gamma=0.99,
                        tau=0.005, lam=1.0, n_action_samples=1, max_weight=20.0, n_critics=2,
                        target_reduction_type='min', update_actor_interval=1, use_gpu=False,
                        scaler=None, action_scaler=None, reward_scaler=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.models.optimizers.OptimizerFactory*) – optimizer factory for the actor.

- **critic\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **actor\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the actor.
- **critic\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the critic.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **tau** (`float`) – target network synchronization coefficient.
- **lam** (`float`) –  $\lambda$  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.
- **target\_reduction\_type** (`str`) – ensemble reduction method at target value estimation. The available options are ['min', 'max', 'mean', 'mix', 'none'].
- **update\_actor\_interval** (`int`) – interval to update policy function.
- **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']
- **action\_scaler** (`d3rlpy.preprocessing.ActionScaler or str`) – action preprocessor. The available options are ['min\_max'].
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler or str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.algos.torch.awac_impl.AWACImpl`) – algorithm implementation.
- **kwargs** (`Any`) –

## Methods

**build\_with\_dataset** (`dataset`)

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

**build\_with\_env** (`env`)

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

**collect**(*env*, *buffer*=None, *explorer*=None, *n\_steps*=1000000, *show\_progress*=True, *timelimit\_aware*=True)  
 Collects data via interaction with environment.

If *buffer* is not given, ReplayBuffer will be internally created.

#### Parameters

- **env** (*gym.core.Env*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.Buffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_steps** (*int*) – the number of total steps to train.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.
- **timelimit\_aware** (*bool*) – flag to turn terminal flag False when TimeLimit truncated flag is True, which is designed to incorporate with *gym.wrappers.TimeLimit*.

**Returns** replay buffer with the collected data.

**Return type** *d3rlpy.online.buffers.Buffer*

**copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** *algo* (*d3rlpy.algos.base.AlgoBase*) – algorithm object.

**Return type** *None*

**copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** *algo* (*d3rlpy.algos.base.AlgoBase*) – algorithm object.

**Return type** *None*

**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** (*Sequence[int]*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

#### Return type `None`

**fit**(*dataset*, *n\_epochs=None*, *n\_steps=None*, *n\_steps\_per\_epoch=10000*, *save\_metrics=True*,  
*experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*,  
*show\_progress=True*, *tensorboard\_dir=None*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*,  
*shuffle=True*, *callback=None*)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

#### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when *n\_steps* is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (*algo, epoch, total\_step*) , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** `List[Tuple[int, Dict[str, float]]]`

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
    n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
    save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
    logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
    timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.BatchBuffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

#### Return type `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
    logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
    timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

## Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

## Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when **n\_steps** is *None*.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if *None*, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (*algo, epoch, total\_step*), which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

**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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

**generate\_new\_data**(*transitions*)

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** *Optional[List[d3rlpy.dataset.Transition]]*

**get\_action\_type**()

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

**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[str, Any]

**load\_model**(*fname*)

Load neural network parameters.

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

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

**Return type** None

**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` (`Union[ndarray, List[Any]]`) – 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` (`Union[ndarray, List[Any]]`) – observations
- `action` (`Union[ndarray, List[Any]]`) – 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** `Union[ndarray, Tuple[ndarray, 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` (`Union[ndarray, List[Any]]`) – 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.

**Return type** `None`

**save\_params**(*logger*)

Saves configurations as `params.json`.

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

**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` (Any) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** d3rlpy.base.LearnableBase

**update**(batch)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** Dict[str, float]

## Attributes

**action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

**action\_size**

Action size.

**Returns** action size.

**Return type** Optional[int]

**active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

**batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** int

**gamma**

Discount factor.

**Returns** discount factor.

**Return type** float

**grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

**impl**

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

#### n\_frames

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

#### n\_steps

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

#### observation\_shape

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

#### reward\_scaler

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

#### scaler

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## d3rlpy.algos.PLAS

```
class d3rlpy.algos.PLAS(*, actor_learning_rate=0.0001, critic_learning_rate=0.001,
                        imitator_learning_rate=0.0001,
                        actor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                        betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                        critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                        betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                        imitator_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                        betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                        actor_encoder_factory='default', critic_encoder_factory='default',
                        imitator_encoder_factory='default', q_func_factory='mean', batch_size=100,
                        n_frames=1, n_steps=1, gamma=0.99, tau=0.005, n_critics=2,
                        target_reduction_type='mix', update_actor_interval=1, lam=0.75,
                        warmup_steps=500000, beta=0.5, use_gpu=False, scaler=None,
                        action_scaler=None, reward_scaler=None, impl=None, **kwargs)
```

Policy in Latent Action Space algorithm.

PLAS is an offline deep reinforcement learning algorithm whose policy function is trained in latent space of Conditional VAE. Unlike other algorithms, PLAS can achieve good performance by using its less constrained

policy function.

$$a \sim p_\beta(a|s, z = \pi_\phi(s))$$

where  $\beta$  is a parameter of the decoder in Conditional VAE.

## References

- Zhou et al., PLAS: latent action space for offline reinforcement learning.

### 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.models.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **imitator\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the conditional VAE.
- **actor\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the actor.
- **critic\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the critic.
- **imitator\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the conditional VAE.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **tau** (`float`) – target network synchronization coefficient.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **target\_reduction\_type** (`str`) – ensemble reduction method at target value estimation. The available options are ['min', 'max', 'mean', 'mix', 'none'].
- **update\_actor\_interval** (`int`) – interval to update policy function.
- **lam** (`float`) – weight factor for critic ensemble.
- **warmup\_steps** (`int`) – the number of steps to warmup the VAE.
- **beta** (`float`) – KL regularization term for Conditional VAE.
- **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'].
- **action\_scaler** (*d3rlpy.preprocessing.ActionScaler or str*) – action preprocessor. The available options are ['min\_max'].
- **reward\_scaler** (*d3rlpy.preprocessing.RewardScaler or str*) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (*d3rlpy.algos.torch.bcq\_impl.BCQImpl*) – algorithm implementation.
- **kwargs** (*Any*) –

## Methods

**build\_with\_dataset**(*dataset*)

Instantiate implementation object with MDPDataset object.

**Parameters** **dataset** (*d3rlpy.dataset.MDPDataset*) – dataset.

**Return type** *None*

**build\_with\_env**(*env*)

Instantiate implementation object with OpenAI Gym object.

**Parameters** **env** (*gym.core.Env*) – gym-like environment.

**Return type** *None*

**collect**(*env, buffer=None, explorer=None, n\_steps=1000000, show\_progress=True, timelimit\_aware=True*)

Collects data via interaction with environment.

If **buffer** is not given, ReplayBuffer will be internally created.

**Parameters**

- **env** (*gym.core.Env*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.Buffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_steps** (*int*) – the number of total steps to train.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.
- **timelimit\_aware** (*bool*) – flag to turn terminal flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** *d3rlpy.online.buffers.Buffer*

**copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithmn
sac = d3rlpy.algos.SAC()
```

(continues on next page)

(continued from previous page)

```
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

```
fit(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
     experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
     show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
     shuffle=True, callback=None)
```

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when *n\_steps* is `None`.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step), which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** *List[Tuple[int, Dict[str, float]]]*

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                 n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                 save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                 logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                 timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (*d3rlpy.envs.batch.BatchEnv*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.BatchBuffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_epochs** (*int*) – the number of epochs to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (*int*) – the number of updates per epoch.
- **eval\_interval** (*int*) – the number of epochs before evaluation.
- **eval\_env** (*Optional[gym.core.Env]*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -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.
- **save\_interval** (*int*) – the number of epochs before saving models.

- **experiment\_name** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (*bool*) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]*) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
            logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
            timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

#### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.Buffer]*) – replay buffer.
- **explorer** (*Optional[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** (*Optional[gym.core.Env]*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -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.
- **save\_interval** (*int*) – the number of epochs before saving models.
- **experiment\_name** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **timelimit\_aware** (*bool*) – flag to turn terminal flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]*) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True, experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None, shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with `eval_episodes`.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.

- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

**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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

**generate\_new\_data(transitions)**

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** Optional[List[d3rlpy.dataset.Transition]]

**get\_action\_type()**

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

**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[str, Any]`

**load\_model**(*fname*)

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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** (`Union[numpy.ndarray, List[Any]]`) – observations
- **action** (`Union[numpy.ndarray, List[Any]]`) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (`Union[numpy.ndarray, List[Any]]`) – 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.**Return type** `None`**save\_params(logger)**Saves configurations as `params.json`.**Parameters** **logger** (`d3rlpy.logger.D3RLPyLogger`) – logger object.**Return type** `None`**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** `logger` (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (*int*) – total gradient step counter.

**Return type** `None`

**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` (*Any*) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

## Attributes

**action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** `Optional[ActionScaler]`

**action\_size**

Action size.

**Returns** action size.

**Return type** `Optional[int]`

**active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

**batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** int

**gamma**

Discount factor.

**Returns** discount factor.

**Return type** float

**grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

**impl**

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

**n\_frames**

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

**n\_steps**

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## d3rlpy.algos.PLASWithPerturbation

```
class d3rlpy.algos.PLASWithPerturbation(*, actor_learning_rate=0.0001, critic_learning_rate=0.001,
                                         imitator_learning_rate=0.0001, ac-
                                         tor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                                         betas=(0.9, 0.999), eps=1e-08, weight_decay=0,
                                         amsgrad=False),
                                         critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                                         betas=(0.9, 0.999), eps=1e-08, weight_decay=0,
                                         amsgrad=False), imita-
                                         tor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                                         betas=(0.9, 0.999), eps=1e-08, weight_decay=0,
                                         amsgrad=False), actor_encoder_factory='default',
                                         critic_encoder_factory='default',
                                         imitator_encoder_factory='default', q_func_factory='mean',
                                         batch_size=100, n_frames=1, n_steps=1, gamma=0.99,
                                         tau=0.005, n_critics=2, target_reduction_type='mix',
                                         update_actor_interval=1, lam=0.75, action_flexibility=0.05,
                                         warmup_steps=500000, beta=0.5, use_gpu=False,
                                         scaler=None, action_scaler=None, reward_scaler=None,
                                         impl=None, **kwargs)
```

Policy in Latent Action Space algorithm with perturbation layer.

PLAS with perturbation layer enables PLAS to output out-of-distribution action.

## References

- Zhou et al., PLAS: latent action space for offline reinforcement learning.

## 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.models.optimizers.OptimizerFactory*) – optimizer factory for the actor.
- **critic\_optim\_factory** (*d3rlpy.models.optimizers.OptimizerFactory*) – optimizer factory for the critic.
- **imitator\_optim\_factory** (*d3rlpy.models.optimizers.OptimizerFactory*) – optimizer factory for the conditional VAE.
- **actor\_encoder\_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the actor.
- **critic\_encoder\_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the critic.
- **imitator\_encoder\_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the conditional VAE.

- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory` or `str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **tau** (`float`) – target network synchronization coefficient.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **target\_reduction\_type** (`str`) – ensemble reduction method at target value estimation. The available options are `['min', 'max', 'mean', 'mix', 'none']`.
- **update\_actor\_interval** (`int`) – interval to update policy function.
- **lam** (`float`) – weight factor for critic ensemble.
- **action\_flexibility** (`float`) – output scale of perturbation layer.
- **warmup\_steps** (`int`) – the number of steps to warmup the VAE.
- **beta** (`float`) – KL regularization term for Conditional VAE.
- **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']`.
- **action\_scaler** (`d3rlpy.preprocessing.ActionScaler` or `str`) – action preprocessor. The available options are `['min_max']`.
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler` or `str`) – reward preprocessor. The available options are `['clip', 'min_max', 'standard']`.
- **impl** (`d3rlpy.algos.torch.bcq_impl.BCQImpl`) – algorithm implementation.
- **kwargs** (`Any`) –

## Methods

**build\_with\_dataset**(`dataset`)

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

**build\_with\_env**(`env`)

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

**collect**(`env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True`)

Collects data via interaction with environment.

If buffer is not given, ReplayBuffer will be internally created.

**Parameters**

- `env` (`gym.core.Env`) – gym-like environment.

- **buffer** (*Optional[d3rlpy.online.buffers.Buffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_steps** (*int*) – the number of total steps to train.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.
- **timelimit\_aware** (*bool*) – flag to turn terminal flag `False` when `TimeLimit`.  
truncated flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

**copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (*d3rlpy.algos.base.AlgoBase*) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (*d3rlpy.algos.base.AlgoBase*) – algorithm object.

**Return type** `None`

**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** (*Sequence[int]*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**Return type** `None`

```
fit(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
     experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
     show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
     shuffle=True, callback=None)
Trains with the given dataset.
```

```
algo.fit(episodes, n_steps=1000000)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when n\_steps is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with eval\_episodes.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step), which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                  n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                  save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                  logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                  timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.BatchBuffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

#### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (*Optional*[`gym.core.Env`]) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional*[`Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]`]) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union*[`List[d3rlpy.dataset.Episode]`, `d3rlpy.dataset.MDPDataset`]) – list of episodes to train.
- **n\_epochs** (*Optional*[`int`]) – the number of epochs to train.
- **n\_steps** (*Optional*[`int`]) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is None.

- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional*[`List[d3rlpy.dataset.Episode]`]) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (*Optional*[`Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]`]) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (*Optional*[`Callable[[d3rlpy.base.LearnableBase, int, int], None]`]) – callable function that takes `(algo, epoch, total_step)` , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

**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** (*Optional*[`Union[bool, int, d3rlpy.gpu.Device]`]) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** `d3rlpy.base.LearnableBase`

**generate\_new\_data(transitions)**

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** `transitions` (`List[d3rlpy.dataset.Transition]`) – list of transitions.

**Returns** list of new transitions.

**Return type** `Optional[List[d3rlpy.dataset.Transition]]`

**get\_action\_type()**

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** `d3rlpy.constants.ActionSpace`

**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[str, Any]`

**load\_model(fname)**

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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** (*Union[numpy.ndarray, List[Any]]*) – observations
- **action** (*Union[numpy.ndarray, List[Any]]*) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (*Union[numpy.ndarray, List[Any]]*) – 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.**Return type** `None`**save\_params**(*logger*)Saves configurations as `params.json`.**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** **grad\_step** (*int*) – total gradient step counter.

**Return type** `None`

**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** (*Any*) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** Dict[str, float]

## Attributes

### action\_scaler

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

### action\_size

Action size.

**Returns** action size.

**Return type** Optional[int]

### active\_logger

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

### batch\_size

Batch size to train.

**Returns** batch size.

**Return type** int

### gamma

Discount factor.

**Returns** discount factor.

**Return type** float

### grad\_step

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

### impl

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

### n\_frames

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

### n\_steps

N-step TD backup.

---

**Returns** N-step TD backup.

**Return type** `int`

**observation\_shape**  
Observation shape.

**Returns** observation shape.

**Return type** `Optional[Sequence[int]]`

**reward\_scaler**  
Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** `Optional[RewardScaler]`

**scaler**  
Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** `Optional[Scaler]`

## d3rlpy.algos.TD3PlusBC

```
class d3rlpy.algos.TD3PlusBC(*, actor_learning_rate=0.0003, critic_learning_rate=0.0003, actor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam', betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False), critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam', betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False), actor_encoder_factory='default', critic_encoder_factory='default', q_func_factory='mean', batch_size=256, n_frames=1, n_steps=1, gamma=0.99, tau=0.005, n_critics=2, target_reduction_type='min', target_smoothing_sigma=0.2, target_smoothing_clip=0.5, alpha=2.5, update_actor_interval=2, use_gpu=False, scaler='standard', action_scaler=None, reward_scaler=None, impl=None, **kwargs)
```

TD3+BC algorithm.

TD3+BC is an simple offline RL algorithm built on top of TD3. TD3+BC introduces BC-reguralized policy objective function.

$$J(\phi) = \mathbb{E}_{s,a \sim D} [\lambda Q(s, \pi(s)) - (a - \pi(s))^2]$$

where

$$\lambda = \frac{\alpha}{\frac{1}{N} \sum_i (s_i, a_i) |Q(s_i, a_i)|}$$

## References

- Fujimoto et al., A Minimalist Approach to Offline Reinforcement Learning.

### Parameters

- **actor\_learning\_rate** (`float`) – learning rate for a policy function.
- **critic\_learning\_rate** (`float`) – learning rate for Q functions.
- **actor\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **actor\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the actor.
- **critic\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the critic.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **tau** (`float`) – target network synchronization coefficient.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **target\_reduction\_type** (`str`) – ensemble reduction method at target value estimation. The available options are ['min', 'max', 'mean', 'mix', 'none'].
- **target\_smoothing\_sigma** (`float`) – standard deviation for target noise.
- **target\_smoothing\_clip** (`float`) – clipping range for target noise.
- **alpha** (`float`) –  $\alpha$  value.
- **update\_actor\_interval** (`int`) – interval to update policy function described as *delayed policy update* in the paper.
- **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'].
- **action\_scaler** (`d3rlpy.preprocessing.ActionScaler or str`) – action preprocessor. The available options are ['min\_max'].
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler or str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.algos.torch.td3_impl.TD3Impl`) – algorithm implementation.
- **kwargs** (`Any`) –

## Methods

### `build_with_dataset(dataset)`

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset (d3rlpy.dataset.MDPDataset)` – dataset.

**Return type** `None`

### `build_with_env(env)`

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env (gym.core.Env)` – gym-like environment.

**Return type** `None`

### `collect(env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True)`

Collects data via interaction with environment.

If `buffer` is not given, `ReplayBuffer` will be internally created.

**Parameters**

- `env (gym.core.Env)` – gym-like environment.
- `buffer (Optional[d3rlpy.online.buffers.Buffer])` – replay buffer.
- `explorer (Optional[d3rlpy.online.explorers.Explorer])` – action explorer.
- `n_steps (int)` – the number of total steps to train.
- `show_progress (bool)` – flag to show progress bar for iterations.
- `timelimit_aware (bool)` – flag to turn `terminal` flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

### `copy_policy_from(algo)`

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo (d3rlpy.algos.base.AlgoBase)` – algorithm object.

**Return type** `None`

### `copy_q_function_from(algo)`

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

**fit**(`dataset`, `n_epochs=None`, `n_steps=None`, `n_steps_per_epoch=10000`, `save_metrics=True`, `experiment_name=None`, `with_timestamp=True`, `logdir='d3rlpy_logs'`, `verbose=True`, `show_progress=True`, `tensorboard_dir=None`, `eval_episodes=None`, `save_interval=1`, `scorers=None`, `shuffle=True`, `callback=None`)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.

- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                  n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                  save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                  logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                  timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (*d3rlpy.envs.batch.BatchEnv*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.BatchBuffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_epochs** (*int*) – the number of epochs to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (*int*) – the number of updates per epoch.
- **eval\_interval** (*int*) – the number of epochs before evaluation.
- **eval\_env** (*Optional[gym.core.Env]*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -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.
- **save\_interval** (*int*) – the number of epochs before saving models.
- **experiment\_name** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.

- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.  
truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If `None`, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). If `None`, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.  
truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each iteration algo methods and properties can be changed or queried.**

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when n\_steps is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with eval\_episodes.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

```
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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

### generate\_new\_data(*transitions*)

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** *Optional[List[d3rlpy.dataset.Transition]]*

### get\_action\_type()

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

### 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[str, Any]

### load\_model(*fname*)

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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` (`Union[numpy.ndarray, List[Any]]`) – observations
- `action` (`Union[numpy.ndarray, List[Any]]`) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (*Union[`numpy.ndarray`, `List[Any]`]*) – 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.

**Return type** `None`

**save\_params(*logger*)**

Saves configurations as params.json.

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger(*logger*)**

Set active D3RLPyLogger object

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

**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` (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

**Attributes****action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** `Optional[ActionScaler]`

**action\_size**

Action size.

**Returns** action size.

**Return type** `Optional[int]`

**active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

**batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** `int`

**gamma**

Discount factor.

**Returns** discount factor.

**Return type** float

**grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

**impl**

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

**n\_frames**

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

**n\_steps**

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## d3rlpy.algos.MOPO

```
class d3rlpy.algos.MOPO(*, actor_learning_rate=0.0003, critic_learning_rate=0.0003,
                       temp_learning_rate=0.0003,
                       actor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                       betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                       critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                       betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                       temp_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                       betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                       actor_encoder_factory='default', critic_encoder_factory='default',
                       q_func_factory='mean', batch_size=100, n_frames=1, n_steps=1, gamma=0.99,
                       tau=0.005, n_critics=2, target_reduction_type='min', update_actor_interval=1,
                       initial_temperature=1.0, dynamics=None, rollout_interval=1000, rollout_horizon=5,
                       rollout_batch_size=50000, lam=1.0, real_ratio=0.05, generated maxlen=1250000,
                       use_gpu=False, scaler=None, action_scaler=None, reward_scaler=None,
                       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  $T_{\theta}$ .

$$s_{t+1}, r_{t+1} \sim T_{\theta}(s_t, a_t)$$

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.

$$\tilde{r_{t+1}} = r_{t+1} - \lambda \max_{i=1}^N \|\Sigma_i(s_t, a_t)\|$$

where  $\Sigma(s_t, a_t)$  is the estimated variance. Finally, the generated transitions  $(s_t, a_t, \tilde{r_{t+1}}, s_{t+1})$  are appended to dataset  $D$ . This generation process starts with randomly sampled `n_initial_transitions` transitions till `horizon` steps.

---

**Note:** Currently, MOPO only supports vector observations.

---

## References

- Yu et al., MOPO: Model-based Offline Policy Optimization.

## 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.models.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- `critic_optim_factory` (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.

- **temp\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the temperature.
- **actor\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the actor.
- **critic\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the critic.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **tau** (`float`) – target network synchronization coefficient.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **target\_reduction\_type** (`str`) – ensemble reduction method at target value estimation. The available options are `['min', 'max', 'mean', 'mix', 'none']`.
- **update\_actor\_interval** (`int`) – interval to update policy function.
- **initial\_temperature** (`float`) – initial temperature value.
- **dynamics** (`d3rlpy.dynamics.DynamicsBase`) – dynamics object.
- **rollout\_interval** (`int`) – the number of steps before rollout.
- **rollout\_horizon** (`int`) – the rollout step length.
- **rollout\_batch\_size** (`int`) – the number of initial transitions for rollout.
- **lam** (`float`) –  $\lambda$  for uncertainty penalties.
- **real\_ratio** (`float`) – the real of dataset samples in a mini-batch.
- **generated maxlen** (`int`) – the maximum number of generated samples.
- **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']`.
- **action\_scaler** (`d3rlpy.preprocessing.ActionScaler or str`) – action preprocessor. The available options are `['min_max']`.
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler or str`) – reward preprocessor. The available options are `['clip', 'min_max', 'standard']`.
- **impl** (`d3rlpy.algos.torch.sac_impl.SACImpl`) – algorithm implementation.
- **kwargs** (`Any`) –

## Methods

**build\_with\_dataset**(*dataset*)

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset (d3rlpy.dataset.MDPDataset)` – dataset.

**Return type** `None`

**build\_with\_env**(*env*)

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env (gym.core.Env)` – gym-like environment.

**Return type** `None`

**collect**(*env, buffer=None, explorer=None, n\_steps=1000000, show\_progress=True, timelimit\_aware=True*)

Collects data via interaction with environment.

If `buffer` is not given, `ReplayBuffer` will be internally created.

**Parameters**

- `env (gym.core.Env)` – gym-like environment.
- `buffer (Optional[d3rlpy.online.buffers.Buffer])` – replay buffer.
- `explorer (Optional[d3rlpy.online.explorers.Explorer])` – action explorer.
- `n_steps (int)` – the number of total steps to train.
- `show_progress (bool)` – flag to show progress bar for iterations.
- `timelimit_aware (bool)` – flag to turn `terminal` flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

**copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo (d3rlpy.algos.base.AlgoBase)` – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

**fit**(`dataset`, `n_epochs=None`, `n_steps=None`, `n_steps_per_epoch=10000`, `save_metrics=True`, `experiment_name=None`, `with_timestamp=True`, `logdir='d3rlpy_logs'`, `verbose=True`, `show_progress=True`, `tensorboard_dir=None`, `eval_episodes=None`, `save_interval=1`, `scorers=None`, `shuffle=True`, `callback=None`)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.

- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                  n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                  save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                  logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                  timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (*d3rlpy.envs.batch.BatchEnv*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.BatchBuffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_epochs** (*int*) – the number of epochs to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (*int*) – the number of updates per epoch.
- **eval\_interval** (*int*) – the number of epochs before evaluation.
- **eval\_env** (*Optional[gym.core.Env]*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -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.
- **save\_interval** (*int*) – the number of epochs before saving models.
- **experiment\_name** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.

- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.  
truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If `None`, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). If `None`, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.  
truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each iteration algo methods and properties can be changed or queried.**

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when n\_steps is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with eval\_episodes.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

```
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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

### generate\_new\_data(*transitions*)

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** *Optional[List[d3rlpy.dataset.Transition]]*

### get\_action\_type()

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

### 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[str, Any]

### load\_model(*fname*)

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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` (`Union[numpy.ndarray, List[Any]]`) – observations
- `action` (`Union[numpy.ndarray, List[Any]]`) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (*Union[`numpy.ndarray`, `List[Any]`]*) – 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.

**Return type** `None`

**save\_params(*logger*)**

Saves configurations as params.json.

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger(*logger*)**

Set active D3RLPyLogger object

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

**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` (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

**Attributes****action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** `Optional[ActionScaler]`

**action\_size**

Action size.

**Returns** action size.

**Return type** `Optional[int]`

**active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

**batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** `int`

**gamma**

Discount factor.

**Returns** discount factor.

**Return type** float

**grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

**impl**

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

**n\_frames**

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

**n\_steps**

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## d3rlpy.algos.COMBO

```
class d3rlpy.algos.COMBO(*, actor_learning_rate=0.0001, critic_learning_rate=0.0003,
                           temp_learning_rate=0.0001,
                           actor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                           betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                           critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                           betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                           temp_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                           betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                           actor_encoder_factory='default', critic_encoder_factory='default',
                           q_func_factory='mean', batch_size=256, n_frames=1, n_steps=1, gamma=0.99,
                           tau=0.005, n_critics=2, target_reduction_type='min', update_actor_interval=1,
                           initial_temperature=1.0, conservative_weight=1.0, n_action_samples=10,
                           soft_q_backup=False, dynamics=None, rollout_interval=1000, rollout_horizon=5,
                           rollout_batch_size=50000, real_ratio=0.5, generated maxlen=1250000,
                           use_gpu=False, scaler=None, action_scaler=None, reward_scaler=None,
                           impl=None, **kwargs)
```

Conservative Offline Model-Based Optimization.

COMBO is a model-based RL approach for offline policy optimization. COMBO is similar to MOPO, but it also leverages conservative loss proposed in CQL.

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

---

**Note:** Currently, COMBO only supports vector observations.

---

## References

- Yu et al., COMBO: Conservative Offline Model-Based Policy Optimization.

### 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.models.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **temp\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the temperature.
- **actor\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the actor.
- **critic\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the critic.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.

- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **tau** (`float`) – target network synchronization coefficient.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **target\_reduction\_type** (`str`) – ensemble reduction method at target value estimation. The available options are ['min', 'max', 'mean', 'mix', 'none'].
- **update\_actor\_interval** (`int`) – interval to update policy function.
- **initial\_temperature** (`float`) – initial temperature value.
- **conservative\_weight** (`float`) – constant weight to scale conservative loss.
- **n\_action\_samples** (`int`) – the number of sampled actions to compute  $\log \sum_a \exp Q(s, a)$ .
- **soft\_q\_backup** (`bool`) – flag to use SAC-style backup.
- **dynamics** (`d3rlpy.dynamics.DynamicsBase`) – dynamics object.
- **rollout\_interval** (`int`) – the number of steps before rollout.
- **rollout\_horizon** (`int`) – the rollout step length.
- **rollout\_batch\_size** (`int`) – the number of initial transitions for rollout.
- **real\_ratio** (`float`) – the real of dataset samples in a mini-batch.
- **generated maxlen** (`int`) – the maximum number of generated samples.
- **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'].
- **action\_scaler** (`d3rlpy.preprocessing.ActionScaler` or `str`) – action preprocessor. The available options are ['min\_max'].
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler` or `str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.algos.torch.combo_impl.COMBOImpl`) – algorithm implementation.
- **kwargs** (`Any`) –

## Methods

**build\_with\_dataset**(`dataset`)

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

**build\_with\_env**(`env`)

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

**collect**(*env*, *buffer*=None, *explorer*=None, *n\_steps*=1000000, *show\_progress*=True, *timelimit\_aware*=True)  
 Collects data via interaction with environment.

If *buffer* is not given, ReplayBuffer will be internally created.

#### Parameters

- **env** (*gym.core.Env*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.Buffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_steps** (*int*) – the number of total steps to train.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.
- **timelimit\_aware** (*bool*) – flag to turn terminal flag False when TimeLimit truncated flag is True, which is designed to incorporate with *gym.wrappers.TimeLimit*.

**Returns** replay buffer with the collected data.

**Return type** *d3rlpy.online.buffers.Buffer*

**copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** *algo* (*d3rlpy.algos.base.AlgoBase*) – algorithm object.

**Return type** *None*

**copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** *algo* (*d3rlpy.algos.base.AlgoBase*) – algorithm object.

**Return type** *None*

**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** (*Sequence[int]*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

#### Return type `None`

**fit**(*dataset*, *n\_epochs=None*, *n\_steps=None*, *n\_steps\_per\_epoch=10000*, *save\_metrics=True*,  
*experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*,  
*show\_progress=True*, *tensorboard\_dir=None*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*,  
*shuffle=True*, *callback=None*)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

#### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when *n\_steps* is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (*algo, epoch, total\_step*) , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** `List[Tuple[int, Dict[str, float]]]`

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,  

    n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,  

    save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,  

    logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,  

    timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (*d3rlpy.envs.batch.BatchEnv*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.BatchBuffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_epochs** (*int*) – the number of epochs to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (*int*) – the number of updates per epoch.
- **eval\_interval** (*int*) – the number of epochs before evaluation.
- **eval\_env** (*Optional[gym.core.Env]*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -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.
- **save\_interval** (*int*) – the number of epochs before saving models.
- **experiment\_name** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (*bool*) – flag to turn terminal flag False when TimeLimit.truncated flag is True, which is designed to incorporate with *gym.wrappers.TimeLimit*.
- **callback** (*Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]*) – callable function that takes (*algo*, *epoch*, *total\_step*), which is called at the end of epochs.

#### Return type

`None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,  

    logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,  

    timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

## Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

## Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when **n\_steps** is *None*.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if *None*, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (*algo, epoch, total\_step*), which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

**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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

**generate\_new\_data**(*transitions*)

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** *Optional[List[d3rlpy.dataset.Transition]]*

**get\_action\_type**()

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

**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[str, Any]

**load\_model**(*fname*)

Load neural network parameters.

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

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

**Return type** None

**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` (`Union[ndarray, List[Any]]`) – 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` (`Union[ndarray, List[Any]]`) – observations
- `action` (`Union[ndarray, List[Any]]`) – 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** `Union[ndarray, Tuple[ndarray, 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` (`Union[ndarray, List[Any]]`) – 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.

**Return type** `None`

**save\_params**(*logger*)

Saves configurations as `params.json`.

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

**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` (Any) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** d3rlpy.base.LearnableBase

**update**(batch)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** Dict[str, float]

## Attributes

**action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

**action\_size**

Action size.

**Returns** action size.

**Return type** Optional[int]

**active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

**batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** int

**gamma**

Discount factor.

**Returns** discount factor.

**Return type** float

**grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

**impl**

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

#### n\_frames

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

#### n\_steps

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

#### observation\_shape

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

#### reward\_scaler

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

#### scaler

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## d3rlpy.algos.RandomPolicy

```
class d3rlpy.algos.RandomPolicy(*, distribution='uniform', normal_std=1.0, action_scaler=None,  
                                **kwargs)
```

Random Policy for continuous control algorithm.

This is designed for data collection and lightweight interaction tests. `fit` and `fit_online` methods will raise exceptions.

#### Parameters

- **distribution** (str) – random distribution. The available options are ['uniform', 'normal'].
- **normal\_std** (float) – standard deviation of the normal distribution. This is only used when `distribution='normal'`.
- **action\_scaler** (d3rlpy.preprocessing.ActionScaler or str) – action preprocessor. The available options are ['min\_max'].
- **kwargs** (Any) –

## Methods

**build\_with\_dataset**(*dataset*)

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset (d3rlpy.dataset.MDPDataset)` – dataset.

**Return type** `None`

**build\_with\_env**(*env*)

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env (gym.core.Env)` – gym-like environment.

**Return type** `None`

**collect**(*env, buffer=None, explorer=None, n\_steps=1000000, show\_progress=True, timelimit\_aware=True*)

Collects data via interaction with environment.

If `buffer` is not given, `ReplayBuffer` will be internally created.

**Parameters**

- `env (gym.core.Env)` – gym-like environment.
- `buffer (Optional[d3rlpy.online.buffers.Buffer])` – replay buffer.
- `explorer (Optional[d3rlpy.online.explorers.Explorer])` – action explorer.
- `n_steps (int)` – the number of total steps to train.
- `show_progress (bool)` – flag to show progress bar for iterations.
- `timelimit_aware (bool)` – flag to turn `terminal` flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

**copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo (d3rlpy.algos.base.AlgoBase)` – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

**fit**(`dataset`, `n_epochs=None`, `n_steps=None`, `n_steps_per_epoch=10000`, `save_metrics=True`, `experiment_name=None`, `with_timestamp=True`, `logdir='d3rlpy_logs'`, `verbose=True`, `show_progress=True`, `tensorboard_dir=None`, `eval_episodes=None`, `save_interval=1`, `scorers=None`, `shuffle=True`, `callback=None`)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.

- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                  n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                  save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                  logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                  timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (*d3rlpy.envs.batch.BatchEnv*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.BatchBuffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_epochs** (*int*) – the number of epochs to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (*int*) – the number of updates per epoch.
- **eval\_interval** (*int*) – the number of epochs before evaluation.
- **eval\_env** (*Optional[gym.core.Env]*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -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.
- **save\_interval** (*int*) – the number of epochs before saving models.
- **experiment\_name** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.

- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.  
truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If `None`, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). If `None`, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.  
truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each iteration algo methods and properties can be changed or queried.**

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when **n\_steps** is *None*.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if *None*, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (*algo, epoch, total\_step*), which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

```
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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

### generate\_new\_data(*transitions*)

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** *Optional[List[d3rlpy.dataset.Transition]]*

### get\_action\_type()

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

### 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[str, Any]

### load\_model(*fname*)

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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` (`Union[numpy.ndarray, List[Any]]`) – observations
- `action` (`Union[numpy.ndarray, List[Any]]`) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (*Union[`numpy.ndarray`, `List[Any]`]*) – 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.

**Return type** `None`

**save\_params(*logger*)**

Saves configurations as params.json.

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger(*logger*)**

Set active D3RLPyLogger object

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

**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` (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

**Attributes****action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** `Optional[ActionScaler]`

**action\_size**

Action size.

**Returns** action size.

**Return type** `Optional[int]`

**active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

**batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** `int`

**gamma**

Discount factor.

**Returns** discount factor.

**Return type** float

**grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

**impl**

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

**n\_frames**

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

**n\_steps**

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

### 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 veriosn of Advantage-Weighted Regression algorithm.
<code>d3rlpy.algos.DiscreteRandomPolicy</code>	Random Policy for discrete control algorithm.

#### `d3rlpy.algos.DiscreteBC`

```
class d3rlpy.algos.DiscreteBC(*, learning_rate=0.001,
                             optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                             betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                             encoder_factory='default', batch_size=100, n_frames=1, beta=0.5,
                             use_gpu=False, scaler=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`) – learing rate.
- **optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory.
- **encoder\_factory** (`d3rlpy.models.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`) – reguralization factor.
- **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']
- **impl** (`d3rlpy.algos.torch.bc_impl.DiscreteBCImpl`) – implemenation of the algorithm.
- **kwargs** (`Any`) –

## Methods

### **build\_with\_dataset**(*dataset*)

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset (d3rlpy.dataset.MDPDataset)` – dataset.

**Return type** `None`

### **build\_with\_env**(*env*)

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env (gym.core.Env)` – gym-like environment.

**Return type** `None`

### **collect**(*env, buffer=None, explorer=None, n\_steps=1000000, show\_progress=True, timelimit\_aware=True*)

Collects data via interaction with environment.

If `buffer` is not given, `ReplayBuffer` will be internally created.

**Parameters**

- `env (gym.core.Env)` – gym-like environment.
- `buffer (Optional[d3rlpy.online.buffers.Buffer])` – replay buffer.
- `explorer (Optional[d3rlpy.online.explorers.Explorer])` – action explorer.
- `n_steps (int)` – the number of total steps to train.
- `show_progress (bool)` – flag to show progress bar for iterations.
- `timelimit_aware (bool)` – flag to turn `terminal` flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

### **copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo (d3rlpy.algos.base.AlgoBase)` – algorithm object.

**Return type** `None`

### **copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

**fit**(`dataset`, `n_epochs=None`, `n_steps=None`, `n_steps_per_epoch=10000`, `save_metrics=True`, `experiment_name=None`, `with_timestamp=True`, `logdir='d3rlpy_logs'`, `verbose=True`, `show_progress=True`, `tensorboard_dir=None`, `eval_episodes=None`, `save_interval=1`, `scorers=None`, `shuffle=True`, `callback=None`)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.

- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                 n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                 save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                 logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                 timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (*d3rlpy.envs.batch.BatchEnv*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.BatchBuffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_epochs** (*int*) – the number of epochs to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (*int*) – the number of updates per epoch.
- **eval\_interval** (*int*) – the number of epochs before evaluation.
- **eval\_env** (*Optional[gym.core.Env]*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -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.
- **save\_interval** (*int*) – the number of epochs before saving models.
- **experiment\_name** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.

- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.  
truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes (`algo, epoch, total_step`) , which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If `None`, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.  
truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes (`algo, epoch, total_step`) , which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when n\_steps is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with eval\_episodes.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

```
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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.**Return type** d3rlpy.base.LearnableBase**generate\_new\_data**(*transitions*)

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.**Returns** list of new transitions.**Return type** *Optional[List[d3rlpy.dataset.Transition]]***get\_action\_type**()

Returns action type (continuous or discrete).

**Returns** action type.**Return type** d3rlpy.constants.ActionSpace**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[*str*, Any]**load\_model**(*fname*)

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – observations

**Returns** greedy actions

**Return type** `numpy.ndarray`

**predict\_value**(*x*, *action*, *with\_std=False*)

value prediction is not supported by BC algorithms.

**Parameters**

- `x` (`Union[numpy.ndarray, List[Any]]`) –
- `action` (`Union[numpy.ndarray, List[Any]]`) –
- `with_std` (`bool`) –

**Return type** `numpy.ndarray`

**sample\_action**(*x*)

sampling action is not supported by BC algorithm.

**Parameters** `x` (`Union[numpy.ndarray, List[Any]]`) –

**Return type** `None`

**save\_model**(*fname*)

Saves neural network parameters.

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

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

**Return type** `None`

**save\_params**(*logger*)

Saves configurations as params.json.

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** **logger** (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** **grad\_step** (`int`) – total gradient step counter.

**Return type** `None`

**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** (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** Dict[str, float]

## Attributes

### **action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

### **action\_size**

Action size.

**Returns** action size.

**Return type** Optional[int]

### **active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

### **batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** int

### **gamma**

Discount factor.

**Returns** discount factor.

**Return type** float

### **grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

### **impl**

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

### **n\_frames**

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

### **n\_steps**

N-step TD backup.

**Returns** N-step TD backup.

---

**Return type** `int`

**observation\_shape**  
Observation shape.

**Returns** observation shape.

**Return type** `Optional[Sequence[int]]`

**reward\_scaler**  
Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** `Optional[RewardScaler]`

**scaler**  
Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** `Optional[Scaler]`

## d3rlpy.algos.DQN

```
class d3rlpy.algos.DQN(*, learning_rate=6.25e-05,
                      optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam', betas=(0.9,
                      0.999), eps=1e-08, weight_decay=0, amsgrad=False), encoder_factory='default',
                      q_func_factory='mean', batch_size=32, n_frames=1, n_steps=1, gamma=0.99,
                      n_critics=1, target_reduction_type='min', target_update_interval=8000,
                      use_gpu=False, scaler=None, reward_scaler=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  $\theta'$  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.models.optimizers.OptimizerFactory or str`) – optimizer factory.
- **encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.

- **gamma** (`float`) – discount factor.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **target\_reduction\_type** (`str`) – ensemble reduction method at target value estimation. The available options are ['min', 'max', 'mean', 'mix', 'none'].
- **target\_update\_interval** (`int`) – interval to update the target network.
- **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']
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler` or `str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.algos.torch.dqn_impl.DQNImpl`) – algorithm implementation.
- **kwargs** (`Any`) –

## Methods

### `build_with_dataset(dataset)`

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

### `build_with_env(env)`

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

### `collect(env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True)`

Collects data via interaction with environment.

If `buffer` is not given, `ReplayBuffer` will be internally created.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_steps** (`int`) – the number of total steps to train.
- **show\_progress** (`bool`) – flag to show progress bar for iterations.
- **timelimit\_aware** (`bool`) – flag to turn `terminal` flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

### `copy_policy_from(algo)`

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(`algo`)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

**fit**(`dataset`, `n_epochs=None`, `n_steps=None`, `n_steps_per_epoch=10000`, `save_metrics=True`, `experiment_name=None`, `with_timestamp=True`, `logdir='d3rlpy_logs'`, `verbose=True`, `show_progress=True`, `tensorboard_dir=None`, `eval_episodes=None`, `save_interval=1`, `scorers=None`, `shuffle=True`, `callback=None`)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.

- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (`Optional[List[d3rlpy.dataset.Episode]]`) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (`Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]`) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (`Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** `List[Tuple[int, Dict[str, float]]]`

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                 n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                 save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                 logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                 timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.BatchBuffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (`Optional[gym.core.Env]`) – gym-like environment. If `None`, evaluation is skipped.

- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when TimeLimit.truncated flag is True, which is designed to incorporate with gym.wrappers.TimeLimit.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes (algo, epoch, total\_step) , which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
            logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
            timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

#### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional*[`Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]`]) – callable function that takes (`algo, epoch, total_step`) , which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,  
experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,  
show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,  
shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):  
    my_plot(metrics)  
    algo.save_model(my_path)
```

#### Parameters

- **dataset** (*Union*[`List[d3rlpy.dataset.Episode]`, `d3rlpy.dataset.MDPDataset`]) – list of episodes to train.
- **n\_epochs** (*Optional*[`int`]) – the number of epochs to train.
- **n\_steps** (*Optional*[`int`]) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.

- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step), which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

#### **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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

#### **generate\_new\_data(transitions)**

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** Optional[List[d3rlpy.dataset.Transition]]

#### **get\_action\_type()**

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

**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[str, Any]`

**load\_model(fname)**

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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)
```

(continues on next page)

(continued from previous page)

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

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

**Parameters**

- **x** (*Union[`numpy.ndarray`, `List[Any]`]*) – observations
- **action** (*Union[`numpy.ndarray`, `List[Any]`]*) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (*Union[`numpy.ndarray`, `List[Any]`]*) – 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.**Return type** `None`**save\_params(*logger*)**Saves configurations as `params.json`.**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.**Return type** `None`**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

#### `set_active_logger(logger)`

Set active D3RLPyLogger object

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

#### `set_grad_step(grad_step)`

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

#### `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` (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

#### `update(batch)`

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

---

## Attributes

### `action_scaler`

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

### `action_size`

Action size.

**Returns** action size.

**Return type** Optional[int]

### `active_logger`

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

### `batch_size`

Batch size to train.

**Returns** batch size.

**Return type** int

### `gamma`

Discount factor.

**Returns** discount factor.

**Return type** float

### `grad_step`

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

### `impl`

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

### `n_frames`

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

### `n_steps`

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## d3rlpy.algos.DoubleDQN

```
class d3rlpy.algos.DoubleDQN(*, learning_rate=6.25e-05,
                             optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                             betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                             encoder_factory='default', q_func_factory='mean', batch_size=32,
                             n_frames=1, n_steps=1, gamma=0.99, n_critics=1,
                             target_reduction_type='min', target_update_interval=8000, use_gpu=False,
                             scaler=None, reward_scaler=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.

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

where  $\theta'$  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.models.optimizers.OptimizerFactory`) – optimizer factory.
- **encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory` or `str`) – encoder factory.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory` or `str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.

- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **n\_critics** (`int`) – the number of Q functions.
- **target\_reduction\_type** (`str`) – ensemble reduction method at target value estimation. The available options are ['min', 'max', 'mean', 'mix', 'none'].
- **target\_update\_interval** (`int`) – interval to synchronize the target network.
- **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']
- **impl** (`d3rlpy.algos.torch.dqn_impl.DoubleDQNImpl`) – algorithm implementation.
- **reward\_scaler** (`Optional[Union[d3rlpy.preprocessing.reward_scalers.RewardScaler, str]]`) –
- **kwargs** (`Any`) –

## Methods

**build\_with\_dataset**(`dataset`)

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

**build\_with\_env**(`env`)

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

**collect**(`env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True`)

Collects data via interaction with environment.

If buffer is not given, ReplayBuffer will be internally created.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_steps** (`int`) – the number of total steps to train.
- **show\_progress** (`bool`) – flag to show progress bar for iterations.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

**copy\_policy\_from**(`algo`)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(`algo`)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

**fit**(`dataset`, `n_epochs=None`, `n_steps=None`, `n_steps_per_epoch=10000`, `save_metrics=True`, `experiment_name=None`, `with_timestamp=True`, `logdir='d3rlpy_logs'`, `verbose=True`, `show_progress=True`, `tensorboard_dir=None`, `eval_episodes=None`, `save_interval=1`, `scorers=None`, `shuffle=True`, `callback=None`)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.

- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (`Optional[List[d3rlpy.dataset.Episode]]`) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (`Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]`) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (`Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** `List[Tuple[int, Dict[str, float]]]`

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                  n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                  save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                  logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                  timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.BatchBuffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (`Optional[gym.core.Env]`) – gym-like environment. If `None`, evaluation is skipped.

- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when TimeLimit.truncated flag is True, which is designed to incorporate with gym.wrappers.TimeLimit.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes (algo, epoch, total\_step) , which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

#### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag `False` when `TimeLimit`.  
truncated flag is `True`, which is designed to incorporate with `gym.wrappers`.  
`TimeLimit`.
- **callback** (*Optional*[`Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]`]) – callable function that takes (`algo, epoch, total_step`) , which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,  
experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,  
show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,  
shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):  
    my_plot(metrics)  
    algo.save_model(my_path)
```

#### Parameters

- **dataset** (*Union*[`List[d3rlpy.dataset.Episode]`, `d3rlpy.dataset.MDPDataset`]) – list of episodes to train.
- **n\_epochs** (*Optional*[`int`]) – the number of epochs to train.
- **n\_steps** (*Optional*[`int`]) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.

- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

#### **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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

#### **generate\_new\_data(transitions)**

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** Optional[List[d3rlpy.dataset.Transition]]

#### **get\_action\_type()**

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

**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[str, Any]`

**load\_model(fname)**

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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)
```

(continues on next page)

(continued from previous page)

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

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

**Parameters**

- **x** (*Union[`numpy.ndarray`, `List[Any]`]*) – observations
- **action** (*Union[`numpy.ndarray`, `List[Any]`]*) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (*Union[`numpy.ndarray`, `List[Any]`]*) – 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.**Return type** `None`**save\_params(*logger*)**Saves configurations as `params.json`.**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.**Return type** `None`**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

#### `set_active_logger(logger)`

Set active D3RLPyLogger object

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

#### `set_grad_step(grad_step)`

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

#### `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` (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

#### `update(batch)`

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

## Attributes

### `action_scaler`

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

### `action_size`

Action size.

**Returns** action size.

**Return type** Optional[int]

### `active_logger`

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

### `batch_size`

Batch size to train.

**Returns** batch size.

**Return type** int

### `gamma`

Discount factor.

**Returns** discount factor.

**Return type** float

### `grad_step`

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

### `impl`

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

### `n_frames`

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

### `n_steps`

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

**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.models.optimizers.AdamFactory(optim_cls='Adam',
                               betas=(0.9, 0.999), eps=0.0001, weight_decay=0, amsgrad=False),
                               critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                               betas=(0.9, 0.999), eps=0.0001, weight_decay=0, amsgrad=False),
                               temp_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                               betas=(0.9, 0.999), eps=0.0001, weight_decay=0, amsgrad=False),
                               actor_encoder_factory='default', critic_encoder_factory='default',
                               q_func_factory='mean', batch_size=64, n_frames=1, n_steps=1,
                               gamma=0.99, n_critics=2, initial_temperature=1.0,
                               target_update_interval=8000, use_gpu=False, scaler=None,
                               reward_scaler=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.models.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **temp\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the temperature.
- **actor\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the actor.
- **critic\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the critic.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **initial\_temperature** (`float`) – initial temperature value.
- **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']
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler or str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.algos.torch.sac_impl.DiscreteSACImpl`) – algorithm implementation.
- **target\_update\_interval** (`int`) –
- **kwargs** (`Any`) –

## Methods

### `build_with_dataset(dataset)`

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset (d3rlpy.dataset.MDPDataset)` – dataset.

**Return type** `None`

### `build_with_env(env)`

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env (gym.core.Env)` – gym-like environment.

**Return type** `None`

### `collect(env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True)`

Collects data via interaction with environment.

If `buffer` is not given, `ReplayBuffer` will be internally created.

**Parameters**

- `env (gym.core.Env)` – gym-like environment.
- `buffer (Optional[d3rlpy.online.buffers.Buffer])` – replay buffer.
- `explorer (Optional[d3rlpy.online.explorers.Explorer])` – action explorer.
- `n_steps (int)` – the number of total steps to train.
- `show_progress (bool)` – flag to show progress bar for iterations.
- `timelimit_aware (bool)` – flag to turn `terminal` flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

### `copy_policy_from(algo)`

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo (d3rlpy.algos.base.AlgoBase)` – algorithm object.

**Return type** `None`

### `copy_q_function_from(algo)`

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

**fit**(`dataset`, `n_epochs=None`, `n_steps=None`, `n_steps_per_epoch=10000`, `save_metrics=True`, `experiment_name=None`, `with_timestamp=True`, `logdir='d3rlpy_logs'`, `verbose=True`, `show_progress=True`, `tensorboard_dir=None`, `eval_episodes=None`, `save_interval=1`, `scorers=None`, `shuffle=True`, `callback=None`)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.

- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                  n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                  save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                  logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                  timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (*d3rlpy.envs.batch.BatchEnv*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.BatchBuffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_epochs** (*int*) – the number of epochs to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (*int*) – the number of updates per epoch.
- **eval\_interval** (*int*) – the number of epochs before evaluation.
- **eval\_env** (*Optional[gym.core.Env]*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -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.
- **save\_interval** (*int*) – the number of epochs before saving models.
- **experiment\_name** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.

- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.  
truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If `None`, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). If `None`, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.  
truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each iteration algo methods and properties can be changed or queried.**

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when n\_steps is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with eval\_episodes.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

```
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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

### generate\_new\_data(*transitions*)

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** *Optional[List[d3rlpy.dataset.Transition]]*

### get\_action\_type()

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

### 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[str, Any]

### load\_model(*fname*)

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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` (`Union[numpy.ndarray, List[Any]]`) – observations
- `action` (`Union[numpy.ndarray, List[Any]]`) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (*Union[`numpy.ndarray`, `List[Any]`]*) – 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.

**Return type** `None`

**save\_params(*logger*)**

Saves configurations as params.json.

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger(*logger*)**

Set active D3RLPyLogger object

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

**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` (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

**Attributes****action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** `Optional[ActionScaler]`

**action\_size**

Action size.

**Returns** action size.

**Return type** `Optional[int]`

**active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

**batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** `int`

**gamma**

Discount factor.

**Returns** discount factor.

**Return type** float

**grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

**impl**

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

**n\_frames**

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

**n\_steps**

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## d3rlpy.algos.DiscreteBCQ

```
class d3rlpy.algos.DiscreteBCQ(*, learning_rate=6.25e-05,
                                optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                                betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                                encoder_factory='default', q_func_factory='mean', batch_size=32,
                                n_frames=1, n_steps=1, gamma=0.99, n_critics=1,
                                target_reduction_type='min', action_flexibility=0.3, beta=0.5,
                                target_update_interval=8000, use_gpu=False, scaler=None,
                                reward_scaler=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.

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

which eliminates actions with probabilities  $\tau$  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.models.optimizers.OptimizerFactory`) – optimizer factory.
- **encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory` or `str`) – encoder factory.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory` or `str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **target\_reduction\_type** (`str`) – ensemble reduction method at target value estimation. The available options are `['min', 'max', 'mean', 'mix', 'none']`.
- **action\_flexibility** (`float`) – probability threshold represented as  $\tau$ .

- **beta** (*float*) – regularization term for imitation function.
- **target\_update\_interval** (*int*) – interval to update the target network.
- **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']
- **reward\_scaler** (*d3rlpy.preprocessing.RewardScaler* or *str*) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (*d3rlpy.algos.torch.bcq\_impl.DiscreteBCQImpl*) – algorithm implementation.
- **kwargs** (*Any*) –

## Methods

### `build_with_dataset(dataset)`

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (*d3rlpy.dataset.MDPDataset*) – dataset.

**Return type** `None`

### `build_with_env(env)`

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (*gym.core.Env*) – gym-like environment.

**Return type** `None`

### `collect(env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True)`

Collects data via interaction with environment.

If buffer is not given, ReplayBuffer will be internally created.

**Parameters**

- **env** (*gym.core.Env*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.Buffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_steps** (*int*) – the number of total steps to train.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.
- **timelimit\_aware** (*bool*) – flag to turn terminal flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

### `copy_policy_from(algo)`

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)
```

(continues on next page)

(continued from previous page)

```
# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

`copy_q_function_from(algo)`

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

`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` (`Sequence[int]`) – observation shape.
- `action_size` (`int`) – dimension of action-space.

**Return type** `None`

`fit(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True, experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None, shuffle=True, callback=None)`

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- `dataset` (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- `n_epochs` (`Optional[int]`) – the number of epochs to train.
- `n_steps` (`Optional[int]`) – the number of steps to train.
- `n_steps_per_epoch` (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.

- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional*[`List[d3rlpy.dataset.Episode]`]) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (*Optional*[`Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]`]) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (*Optional*[`Callable[[d3rlpy.base.LearnableBase, int, int], None]`]) – callable function that takes `(algo, epoch, total_step)` , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** `List[Tuple[int, Dict[str, float]]]`

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                 n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                 save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                 logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                 timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (*Optional*[`d3rlpy.online.buffers.BatchBuffer`]) – replay buffer.
- **explorer** (*Optional*[`d3rlpy.online.explorers.Explorer`]) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (*Optional*[`gym.core.Env`]) – gym-like environment. If `None`, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

#### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn `terminal` flag `False` when `TimeLimit`.`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional*[`Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]`]) – callable function that takes (`algo`, `epoch`, `total_step`) , which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,  
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,  
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,  
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):  
    my_plot(metrics)  
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union*[`List[d3rlpy.dataset.Episode]`, `d3rlpy.dataset.MDPDataset`]) – list of episodes to train.
- **n\_epochs** (*Optional*[`int`]) – the number of epochs to train.
- **n\_steps** (*Optional*[`int`]) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional*[`List[d3rlpy.dataset.Episode]`]) – list of episodes to test.

- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (`Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]`) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (`Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]`) – callable function that takes (`algo`, `epoch`, `total_step`) , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

**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** (`Optional[Union[bool, int, d3rlpy.gpu.Device]]`) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** `d3rlpy.base.LearnableBase`

**generate\_new\_data(transitions)**

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** `transitions` (`List[d3rlpy.dataset.Transition]`) – list of transitions.

**Returns** list of new transitions.

**Return type** `Optional[List[d3rlpy.dataset.Transition]]`

**get\_action\_type()**

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** `d3rlpy.constants.ActionSpace`

**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[str, Any]

**load\_model**(*fname*)

Load neural network parameters.

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

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

**Return type** None

**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 (Union[numpy.ndarray, List[Any]])` – 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,)
```

(continues on next page)

(continued from previous page)

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

**Parameters**

- **x** (`Union[numpy.ndarray, List\[Any\]]`) – observations
- **action** (`Union[numpy.ndarray, List\[Any\]]`) – 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** `Union[numpy.ndarray, Tuple\[numpy.ndarray, numpy.ndarray\]\]`**sample\_action(x)**

Returns sampled actions.

The sampled actions are identical to the output of `predict` method if the policy is deterministic.**Parameters** **x** (`Union[numpy.ndarray, List\[Any\]]`) – 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.**Return type** `None`**save\_params(logger)**Saves configurations as `params.json`.**Parameters** **logger** (`d3rlpy.logger.D3RLPyLogger`) – logger object.**Return type** `None`**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).

- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** *None*

### **set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** *None*

### **set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** **grad\_step** (*int*) – total gradient step counter.

**Return type** *None*

### **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** (*Any*) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** *d3rlpy.base.LearnableBase*

### **update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** *Dict[str, float]*

---

## Attributes

### `action_scaler`

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

### `action_size`

Action size.

**Returns** action size.

**Return type** Optional[int]

### `active_logger`

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

### `batch_size`

Batch size to train.

**Returns** batch size.

**Return type** int

### `gamma`

Discount factor.

**Returns** discount factor.

**Return type** float

### `grad_step`

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

### `impl`

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

### `n_frames`

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

### `n_steps`

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

**d3rlpy.algos.DiscreteCQL**

```
class d3rlpy.algos.DiscreteCQL(*, learning_rate=6.25e-05,
                                optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                                betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                                encoder_factory='default', q_func_factory='mean', batch_size=32,
                                n_frames=1, n_steps=1, gamma=0.99, n_critics=1,
                                target_reduction_type='min', target_update_interval=8000, alpha=1.0,
                                use_gpu=False, scaler=None, reward_scaler=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.

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

**References**

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

**Parameters**

- **learning\_rate** (`float`) – learning rate.
- **optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory.
- **encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory` or `str`) – encoder factory.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory` or `str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.

- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **target\_reduction\_type** (`str`) – ensemble reduction method at target value estimation. The available options are ['min', 'max', 'mean', 'mix', 'none'].
- **target\_update\_interval** (`int`) – interval to synchronize the target network.
- **alpha** (`float`) – the  $\alpha$  value above.
- **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']
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler` or `str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.algos.torch.cql_impl.DiscreteCQLImpl`) – algorithm implementation.
- **kwargs** (`Any`) –

## Methods

### `build_with_dataset(dataset)`

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

### `build_with_env(env)`

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

### `collect(env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True)`

Collects data via interaction with environment.

If `buffer` is not given, `ReplayBuffer` will be internally created.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_steps** (`int`) – the number of total steps to train.
- **show\_progress** (`bool`) – flag to show progress bar for iterations.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag `False` when `TimeLimit`.  
truncated flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

**copy\_policy\_from(algo)**

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from(algo)**

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

```
fit(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
     experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
     show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
     shuffle=True, callback=None)
```

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.

- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when n\_steps is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with eval\_episodes.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step), which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                 n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                 save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                 logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                 timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (*d3rlpy.envs.batch.BatchEnv*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.BatchBuffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_epochs** (*int*) – the number of epochs to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (*int*) – the number of updates per epoch.
- **eval\_interval** (*int*) – the number of epochs before evaluation.

- **eval\_env** (*Optional[gym.core.Env]*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -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.
- **save\_interval** (*int*) – the number of epochs before saving models.
- **experiment\_name** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (*bool*) – flag to turn terminal flag False when TimeLimit. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

#### Parameters

- **env** (*gym.core.Env*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.Buffer]*) – replay buffer.
- **explorer** (*Optional[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** (*Optional[gym.core.Env]*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -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.
- **save\_interval** (*int*) – the number of epochs before saving models.

- **experiment\_name** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **timelimit\_aware** (*bool*) – flag to turn `terminal` flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]*) – callable function that takes `(algo, epoch, total_step)` , which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,  
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,  
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,  
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each iteration algo methods and properties can be changed or queried.**

```
for epoch, metrics in algo.fitter(episodes):  
    my_plot(metrics)  
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with `eval_episodes`.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (`algo`, `epoch`, `total_step`) , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

**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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** `d3rlpy.base.LearnableBase`

**generate\_new\_data(transitions)**

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** `Optional[List[d3rlpy.dataset.Transition]]`

**get\_action\_type()**

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

**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[str, Any]

**load\_model(fname)**

Load neural network parameters.

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

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

**Return type** None

**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` (`Union[ndarray, List[Any]]`) – 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
```

(continues on next page)

(continued from previous page)

```
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** (`Union[numpy.ndarray, List[Any]]`) – observations
- **action** (`Union[numpy.ndarray, List[Any]]`) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (`Union[numpy.ndarray, List[Any]]`) – 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.**Return type** `None`**save\_params**(*logger*)Saves configurations as `params.json`.**Parameters** **logger** (`d3rlpy.logger.D3RLPyLogger`) – logger object.**Return type** `None`**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

#### `set_active_logger(logger)`

Set active D3RLPyLogger object

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

#### `set_grad_step(grad_step)`

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

#### `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` (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

#### `update(batch)`

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

## Attributes

### `action_scaler`

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

### `action_size`

Action size.

**Returns** action size.

**Return type** Optional[int]

### `active_logger`

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

### `batch_size`

Batch size to train.

**Returns** batch size.

**Return type** int

### `gamma`

Discount factor.

**Returns** discount factor.

**Return type** float

### `grad_step`

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

### `impl`

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

### `n_frames`

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

### `n_steps`

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

**d3rlpy.algos.DiscreteAWR**

```
class d3rlpy.algos.DiscreteAWR(*, actor_learning_rate=5e-05, critic_learning_rate=0.0001, ac-
                                tor_optim_factory=d3rlpy.models.optimizers.SGDFactory(optim_cls='SGD',
                                momentum=0.9, dampening=0, weight_decay=0, nesterov=False),
                                critic_optim_factory=d3rlpy.models.optimizers.SGDFactory(optim_cls='SGD',
                                momentum=0.9, dampening=0, weight_decay=0, nesterov=False),
                                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, action_scaler=None,
                                reward_scaler=None, impl=None, **kwargs)
```

Discrete veriosn 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  $\text{TD}(\lambda)$  to mitigate high variance issue.

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

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

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.models.optimizers.OptimizerFactory`) – optimizer factory for the actor.
- **critic\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the critic.
- **actor\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory for the actor.
- **critic\_encoder\_factory** (`d3rlpy.models.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`) –  $\lambda$  for TD( $\lambda$ ).
- **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'].
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler or str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.algos.torch.awr_impl.DiscreteAWRImpl`) – algorithm implementation.
- **action\_scaler** (*Optional[Union[d3rlpy.preprocessing.action\_scalers.ActionScaler, str]]*) –
- **kwargs** (`Any`) –

## Methods

### `build_with_dataset(dataset)`

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset (d3rlpy.dataset.MDPDataset)` – dataset.

**Return type** `None`

### `build_with_env(env)`

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env (gym.core.Env)` – gym-like environment.

**Return type** `None`

### `collect(env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True)`

Collects data via interaction with environment.

If `buffer` is not given, `ReplayBuffer` will be internally created.

**Parameters**

- `env (gym.core.Env)` – gym-like environment.
- `buffer (Optional[d3rlpy.online.buffers.Buffer])` – replay buffer.
- `explorer (Optional[d3rlpy.online.explorers.Explorer])` – action explorer.
- `n_steps (int)` – the number of total steps to train.
- `show_progress (bool)` – flag to show progress bar for iterations.
- `timelimit_aware (bool)` – flag to turn `terminal` flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

### `copy_policy_from(algo)`

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo (d3rlpy.algos.base.AlgoBase)` – algorithm object.

**Return type** `None`

### `copy_q_function_from(algo)`

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (`Sequence[int]`) – observation shape.
- **action\_size** (`int`) – dimension of action-space.

**Return type** `None`

**fit**(`dataset`, `n_epochs=None`, `n_steps=None`, `n_steps_per_epoch=10000`, `save_metrics=True`, `experiment_name=None`, `with_timestamp=True`, `logdir='d3rlpy_logs'`, `verbose=True`, `show_progress=True`, `tensorboard_dir=None`, `eval_episodes=None`, `save_interval=1`, `scorers=None`, `shuffle=True`, `callback=None`)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.

- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                  n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                  save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                  logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                  timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (*d3rlpy.envs.batch.BatchEnv*) – gym-like environment.
- **buffer** (*Optional[d3rlpy.online.buffers.BatchBuffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_epochs** (*int*) – the number of epochs to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (*int*) – the number of updates per epoch.
- **eval\_interval** (*int*) – the number of epochs before evaluation.
- **eval\_env** (*Optional[gym.core.Env]*) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (*float*) –  $\epsilon$ -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.
- **save\_interval** (*int*) – the number of epochs before saving models.
- **experiment\_name** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.

- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.  
truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If `None`, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). If `None`, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.  
truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each iteration algo methods and properties can be changed or queried.**

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when **n\_steps** is *None*.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if *None*, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes *(algo, epoch, total\_step)* , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

```
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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

### generate\_new\_data(*transitions*)

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** *Optional[List[d3rlpy.dataset.Transition]]*

### get\_action\_type()

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

### 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[str, Any]

### load\_model(*fname*)

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – observations

**Returns** greedy actions

**Return type** `numpy.ndarray`

**predict\_value**(*x*, \**args*, \*\**kwargs*)

Returns predicted state values.

**Parameters**

- `x` (`Union[numpy.ndarray, List[Any]]`) – observations.
- `args` (`Any`) –
- `kwargs` (`Any`) –

**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` (`Union[numpy.ndarray, List[Any]]`) – 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.

**Return type** `None`

**save\_params**(*logger*)

Saves configurations as `params.json`.

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** **grad\_step** (`int`) – total gradient step counter.

**Return type** `None`

**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** (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

---

**Parameters** `batch` (`d3rlpy.dataset.TransitionMiniBatch`) – mini-batch data.  
**Returns** dictionary of metrics.  
**Return type** `Dict[str, float]`

## Attributes

### `action_scaler`

Preprocessing action scaler.  
**Returns** preprocessing action scaler.  
**Return type** `Optional[ActionScaler]`

### `action_size`

Action size.  
**Returns** action size.  
**Return type** `Optional[int]`

### `active_logger`

Active D3RLPyLogger object.  
This will be only available during training.  
**Returns** logger object.

### `batch_size`

Batch size to train.  
**Returns** batch size.  
**Return type** `int`

### `gamma`

Discount factor.  
**Returns** discount factor.  
**Return type** `float`

### `grad_step`

Total gradient step counter.  
This value will keep counting after `fit` and `fit_online` methods finish.  
**Returns** total gradient step counter.

### `impl`

Implementation object.  
**Returns** implementation object.  
**Return type** `Optional[ImplBase]`

### `n_frames`

Number of frames to stack.  
This is only for image observation.  
**Returns** number of frames to stack.  
**Return type** `int`

**n\_steps**

N-step TD backup.

**Returns** N-step TD backup.

**Return type** `int`

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** `Optional[Sequence[int]]`

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** `Optional[RewardScaler]`

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** `Optional[Scaler]`

## d3rlpy.algos.DiscreteRandomPolicy

**class** `d3rlpy.algos.DiscreteRandomPolicy(**kwargs)`

Random Policy for discrete control algorithm.

This is designed for data collection and lightweight interaction tests. `fit` and `fit_online` methods will raise exceptions.

### Methods

**Parameters** `kwargs` (`Any`) –

**build\_with\_dataset**(`dataset`)

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

**build\_with\_env**(`env`)

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

**collect**(`env, buffer=None, explorer=None, n_steps=1000000, show_progress=True, timelimit_aware=True`)

Collects data via interaction with environment.

If `buffer` is not given, `ReplayBuffer` will be internally created.

**Parameters**

- `env` (`gym.core.Env`) – gym-like environment.

- `buffer` (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.

- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.
- **n\_steps** (*int*) – the number of total steps to train.
- **show\_progress** (*bool*) – flag to show progress bar for iterations.
- **timelimit\_aware** (*bool*) – flag to turn terminal flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

**copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (*d3rlpy.algos.base.AlgoBase*) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (*d3rlpy.algos.base.AlgoBase*) – algorithm object.

**Return type** `None`

**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** (*Sequence[int]*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**Return type** `None`

```
fit(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,  
     experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,  
     show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,  
     shuffle=True, callback=None)  
Trains with the given dataset.
```

```
algo.fit(episodes, n_steps=1000000)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when n\_steps is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with eval\_episodes.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step), which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** List[Tuple[int, Dict[str, float]]]

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,  
                  n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,  
                  save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,  
                  logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,  
                  timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.BatchBuffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
           logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
           timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

#### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (*Optional* [`gym.core.Env`]) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (*Optional* [`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\_dir** (*Optional* [`str`]) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn `terminal` flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (*Optional* [`Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]`]) – callable function that takes `(algo, epoch, total_step)` , which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,  
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,  
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,  
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):  
    my_plot(metrics)  
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union* [`List[d3rlpy.dataset.Episode]`, `d3rlpy.dataset.MDPDataset`]) – list of episodes to train.
- **n\_epochs** (*Optional* [`int`]) – the number of epochs to train.
- **n\_steps** (*Optional* [`int`]) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is None.

- **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** (*Optional*[`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\_dir** (*Optional*[`str`]) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (*Optional*[`List[d3rlpy.dataset.Episode]`]) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (*Optional*[`Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]`]) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (*Optional*[`Callable[[d3rlpy.base.LearnableBase, int, int], None]`]) – callable function that takes `(algo, epoch, total_step)` , which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

**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** (*Optional*[`Union[bool, int, d3rlpy.gpu.Device]`]) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** `d3rlpy.base.LearnableBase`

**generate\_new\_data(transitions)**

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** `transitions` (`List[d3rlpy.dataset.Transition]`) – list of transitions.

**Returns** list of new transitions.

**Return type** `Optional[List[d3rlpy.dataset.Transition]]`

**get\_action\_type()**

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** `d3rlpy.constants.ActionSpace`

**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[str, Any]`

**load\_model(fname)**

Load neural network parameters.

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

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

**Return type** `None`

**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` (`Union[numpy.ndarray, List[Any]]`) – 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** (`Union[numpy.ndarray, List[Any]]`) – observations
- **action** (`Union[numpy.ndarray, List[Any]]`) – 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** `Union[numpy.ndarray, Tuple[numpy.ndarray, 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** (`Union[numpy.ndarray, List[Any]]`) – 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.

**Return type** `None`

**save\_params**(*logger*)

Saves configurations as `params.json`.

**Parameters** **logger** (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** **logger** (*d3rlpy.logger.D3RLPyLogger*) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** **grad\_step** (*int*) – total gradient step counter.

**Return type** `None`

**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** (*Any*) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** Dict[str, float]

## Attributes

### action\_scaler

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

### action\_size

Action size.

**Returns** action size.

**Return type** Optional[int]

### active\_logger

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

### batch\_size

Batch size to train.

**Returns** batch size.

**Return type** int

### gamma

Discount factor.

**Returns** discount factor.

**Return type** float

### grad\_step

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

### impl

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

### n\_frames

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

### n\_steps

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**  
Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**  
Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**  
Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## 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\_factory argument at algorithm initialization.

```
from d3rlpy.algos import CQL

cql = CQL(q_func_factory='qr') # use Quantile Regression Q function
```

Also you can change hyper parameters.

```
from d3rlpy.models.q_functions import QRQFunctionFactory

q_func = QRQFunctionFactory(n_quantiles=32)

cql = CQL(q_func_factory=q_func)
```

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.

---

`d3rlpy.models.q_functions.  
MeanQFunctionFactory`

Standard Q function factory class.

---

`d3rlpy.models.q_functions.  
QRQFunctionFactory`

Quantile Regression Q function factory class.

continues on next page

Table 3 – continued from previous page

<code>d3rlpy.models.q_functions. IQNQFunctionFactory</code>	Implicit Quantile Network Q function factory class.
<code>d3rlpy.models.q_functions. FQFQFunctionFactory</code>	Fully parameterized Quantile Function Q function factory.

### 3.2.1 d3rlpy.models.q\_functions.MeanQFunctionFactory

`class d3rlpy.models.q_functions.MeanQFunctionFactory(bootstrap=False, share_encoder=False)`  
Standard Q function factory class.

This is the standard Q function factory class.

#### References

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

#### Parameters

- `bootstrap` (`bool`) – flag to bootstrap Q functions.
- `share_encoder` (`bool`) – flag to share encoder over multiple Q functions.

#### Methods

`create_continuous(encoder)`

Returns PyTorch's Q function module.

**Parameters** `encoder` (`d3rlpy.models.torch.encoders.EncoderWithAction`) – an encoder module that processes the observation and action to obtain feature representations.

**Returns** continuous Q function object.

**Return type** `d3rlpy.models.torch.q_functions.mean_q_function.ContinuousMeanQFunction`

`create_discrete(encoder, action_size)`

Returns PyTorch's Q function module.

#### Parameters

- `encoder` (`d3rlpy.models.torch.encoders.Encoder`) – an encoder module that processes the observation to obtain feature representations.
- `action_size` (`int`) – dimension of discrete action-space.

**Returns** discrete Q function object.

**Return type** `d3rlpy.models.torch.q_functions.mean_q_function.DiscreteMeanQFunction`

`get_params(deep=False)`

Returns Q function parameters.

**Returns** Q function parameters.

**Parameters** `deep` (`bool`) –

**Return type** `Dict[str, Any]`

**get\_type()**

Returns Q function type.

**Returns** Q function type.

**Return type** str

**Attributes**

**TYPE:** ClassVar[str] = 'mean'

**bootstrap**

**share\_encoder**

## 3.2.2 d3rlpy.models.q\_functions.QRQFunctionFactory

```
class d3rlpy.models.q_functions.QRQFunctionFactory(bootstrap=False, share_encoder=False,
                                                 n_quantiles=32)
```

Quantile Regression Q function factory class.

**References**

- Dabney et al., Distributional reinforcement learning with quantile regression.

**Parameters**

- **bootstrap** (bool) – flag to bootstrap Q functions.
- **share\_encoder** (bool) – flag to share encoder over multiple Q functions.
- **n\_quantiles** (int) – the number of quantiles.

**Methods****create\_continuous(encoder)**

Returns PyTorch's Q function module.

**Parameters** **encoder** (*d3rlpy.models.torch.encoders.EncoderWithAction*) – an encoder module that processes the observation and action to obtain feature representations.

**Returns** continuous Q function object.

**Return type** *d3rlpy.models.torch.q\_functions.qr\_q\_function.ContinuousQRQFunction*

**create\_discrete(encoder, action\_size)**

Returns PyTorch's Q function module.

**Parameters**

- **encoder** (*d3rlpy.models.torch.encoders.Encoder*) – an encoder module that processes the observation to obtain feature representations.
- **action\_size** (int) – dimension of discrete action-space.

**Returns** discrete Q function object.

**Return type** *d3rlpy.models.torch.q\_functions.qr\_q\_function.DiscreteQRQFunction*

---

```
get_params(deep=False)
    Returns Q function parameters.

        Returns Q function parameters.

    Parameters deep (bool) –
    Return type Dict[str, Any]

get_type()
    Returns Q function type.

        Returns Q function type.

    Return type str
```

### Attributes

```
TYPE: ClassVar[str] = 'qr'

bootstrap
n_quantiles
share_encoder
```

## 3.2.3 d3rlpy.models.q\_functions.IQNQFunctionFactory

```
class d3rlpy.models.q_functions.IQNQFunctionFactory(bootstrap=False, share_encoder=False,
                                                    n_quantiles=64, n_greedy_quantiles=32,
                                                    embed_size=64)
```

Implicit Quantile Network Q function factory class.

### References

- Dabney et al., Implicit quantile networks for distributional reinforcement learning.

### Parameters

- **bootstrap** (`bool`) – flag to bootstrap Q functions.
- **share\_encoder** (`bool`) – flag to share encoder over multiple Q functions.
- **n\_quantiles** (`int`) – the number of quantiles.
- **n\_greedy\_quantiles** (`int`) – the number of quantiles for inference.
- **embed\_size** (`int`) – the embedding size.

## Methods

### `create_continuous(encoder)`

Returns PyTorch's Q function module.

**Parameters** `encoder` (`d3rlpy.models.torch.encoders.EncoderWithAction`) – an encoder module that processes the observation and action to obtain feature representations.

**Returns** continuous Q function object.

**Return type** `d3rlpy.models.torch.q_functions.iqn_q_function.ContinuousIQNQFunction`

### `create_discrete(encoder, action_size)`

Returns PyTorch's Q function module.

#### Parameters

- `encoder` (`d3rlpy.models.torch.encoders.Encoder`) – an encoder module that processes the observation to obtain feature representations.
- `action_size` (`int`) – dimension of discrete action-space.

**Returns** discrete Q function object.

**Return type** `d3rlpy.models.torch.q_functions.iqn_q_function.DiscreteIQNQFunction`

### `get_params(deep=False)`

Returns Q function parameters.

**Returns** Q function parameters.

**Parameters** `deep` (`bool`) –

**Return type** `Dict[str, Any]`

### `get_type()`

Returns Q function type.

**Returns** Q function type.

**Return type** `str`

## Attributes

`TYPE: ClassVar[str] = 'iqn'`

`bootstrap`

`embed_size`

`n_greedy_quantiles`

`n_quantiles`

`share_encoder`

### 3.2.4 d3rlpy.models.q\_functions.FQFQFunctionFactory

```
class d3rlpy.models.q_functions.FQFQFunctionFactory(bootstrap=False, share_encoder=False,
                                                    n_quantiles=32, embed_size=64,
                                                    entropy_coeff=0.0)
```

Fully parameterized Quantile Function Q function factory.

#### References

- Yang et al., Fully parameterized quantile function for distributional reinforcement learning.

#### Parameters

- **bootstrap** (`bool`) – flag to bootstrap Q functions.
- **share\_encoder** (`bool`) – flag to share encoder over multiple Q functions.
- **n\_quantiles** (`int`) – the number of quantiles.
- **embed\_size** (`int`) – the embedding size.
- **entropy\_coeff** (`float`) – the coefficient of entropy penalty term.

#### Methods

**create\_continuous(encoder)**

Returns PyTorch's Q function module.

**Parameters** `encoder` (`d3rlpy.models.torch.encoders.EncoderWithAction`) – an encoder module that processes the observation and action to obtain feature representations.

**Returns** continuous Q function object.

**Return type** `d3rlpy.models.torch.q_functions.fqf_q_function.ContinuousFQFQFunction`

**create\_discrete(encoder, action\_size)**

Returns PyTorch's Q function module.

#### Parameters

- **encoder** (`d3rlpy.models.torch.encoders.Encoder`) – an encoder module that processes the observation to obtain feature representations.
- **action\_size** (`int`) – dimension of discrete action-space.

**Returns** discrete Q function object.

**Return type** `d3rlpy.models.torch.q_functions.fqf_q_function.DiscreteFQFQFunction`

**get\_params(deep=False)**

Returns Q function parameters.

**Returns** Q function parameters.

**Parameters** `deep` (`bool`) –

**Return type** `Dict[str, Any]`

**get\_type()**

Returns Q function type.

**Returns** Q function type.

**Return type** str

#### Attributes

**TYPE:** ClassVar[str] = 'fqf'  
**bootstrap**  
**embed\_size**  
**entropy\_coeff**  
**n\_quantiles**  
**share\_encoder**

### 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:
```

(continues on next page)

(continued from previous page)

```

transition = transition.next_transition

# save as HDF5
dataset.dump('dataset.h5')

# load from HDF5
new_dataset = MDPDataset.load('dataset.h5')

```

Please note that the `observations`, `actions`, `rewards` and `terminals` must be aligned with the same timestep.

```

observations = [s1, s2, s3, ...]
actions      = [a1, a2, a3, ...]
rewards       = [r1, r2, r3, ...]
terminals     = [t1, t2, t3, ...]

```

This alignment might be different from other libraries where the tuple of  $(s_t, a_t, r_{t+1})$ . The advantage of d3rlpy's formulation is that we can explicitly store the last observation which might be useful for the future goal-oriented methods and less confusing.

<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, episode_terminals=None,
                                discrete_action=None, create_mask=False, mask_size=1)
```

Markov-Decision Process Dataset class.

MDPDataset is designed 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))
# 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)
```

(continues on next page)

(continued from previous page)

```
# 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.
- **episode\_terminals** (`numpy.ndarray`) – array of binary episode terminal flags. The given data will be splitted based on this flag. This is useful if you want to specify the non-environment terminations (e.g. timeout). If `None`, the episode terminations match the environment terminations.
- **discrete\_action** (`bool`) – flag to use the given actions as discrete action-space actions. If `None`, the action type is automatically determined.
- **create\_mask** (`bool`) – flag to create binary masks for bootstrapping.
- **mask\_size** (`int`) – ensemble size for mask. If `create_mask` is `False`, this will be ignored.

## Methods

`__getitem__(index)`  
`__len__()`  
`__iter__()`  
`append(observations, actions, rewards, terminals, episode_terminals=None)`  
 Appends new data.

### Parameters

- **observations** (`numpy.ndarray`) – N-D array.
- **actions** (`numpy.ndarray`) – actions.
- **rewards** (`numpy.ndarray`) – rewards.
- **terminals** (`numpy.ndarray`) – terminals.
- **episode\_terminals** (`numpy.ndarray`) – episode terminals.

`build_episodes()`  
 Builds episode objects.

This method will be internally called when accessing the `episodes` property at the first time.

**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, create\_mask=False, mask\_size=1)**

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.
- **create\_mask** (`bool`) – flag to create bootstrapping masks.
- **mask\_size** (`int`) – size of bootstrapping masks.

**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`

**episode\_terminals**

Returns the episode terminal flags.

**Returns** array of episode terminal flags.

**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,
                             terminal=True, create_mask=False, mask_size=1)
```

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.
- **terminal** (`bool`) – binary terminal flag. If False, the episode is not terminated by the environment (e.g. timeout).
- **create\_mask** (`bool`) – flag to create binary masks for bootstrapping.
- **mask\_size** (`int`) – ensemble size for mask. If `create_mask` is False, this will be ignored.

## 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`

**terminal**  
Returns the terminal flag.

**Returns** the terminal flag.

**Return type** `bool`

**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  $t+1$ .
- **mask** (`numpy.ndarray`) – binary mask for bootstrapping.
- **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)

**mask**

Returns binary mask for bootstrapping.

**Returns** array of binary mask.

**Return type** np.ndarray

**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.
- **n\_steps** (`int`) – length of N-step sampling.
- **gamma** (`float`) – discount factor for N-step calculation.

#### Methods

##### `__getitem__(key, /)`

Return `self[key]`.

##### `__len__()`

Return `len(self)`.

##### `__iter__()`

Implement `iter(self)`.

**add\_additional\_data(key, value)**

Add arbitrary additional data.

**Parameters**

- **key** (`str`) – key of data.
- **value** (`any`) – value.

**get\_additional\_data(key)**

Returns specified additional data.

**Parameters** `key` (`str`) – key of data.**Returns** value.**Return type** `any`**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`**masks**

Returns mini-batch of binary masks for bootstrapping.

If any of transitions have an invalid mask, this will return `None`.

**Returns** binary mask.**Return type** `numpy.ndarray`**n\_steps**

Returns mini-batch of the number of steps before next observations.

This will always include only ones if `n_steps=1`. If `n_steps` is bigger than 1. the values will depend on its episode length.

**Returns** the number of steps before next observations.**Return type** `numpy.ndarray`**next\_actions**

Returns mini-batch of actions at  $t+n$ .

**Returns** actions at  $t+n$ .**Return type** `numpy.ndarray`**next\_observations**

Returns mini-batch of observations at  $t+n$ .

**Returns** observations at  $t+n$ .**Return type** `numpy.ndarray` or `torch.Tensor`

---

<b>next_rewards</b>	Returns mini-batch of rewards at $t+n$ .
<b>Returns</b>	rewards at $t+n$ .
<b>Return type</b>	numpy.ndarray
<b>observations</b>	Returns mini-batch of observations at $t$ .
<b>Returns</b>	observations at $t$ .
<b>Return type</b>	numpy.ndarray or torch.Tensor
<b>rewards</b>	Returns mini-batch of rewards at $t$ .
<b>Returns</b>	rewards at $t$ .
<b>Return type</b>	numpy.ndarray
<b>terminals</b>	Returns mini-batch of terminal flags at $t+n$ .
<b>Returns</b>	terminal flags at $t+n$ .
<b>Return type</b>	numpy.ndarray
<b>transitions</b>	Returns transitions.
<b>Returns</b>	list of transitions.
<b>Return type</b>	<i>d3rlpy.dataset.Transition</i>

## 3.4 Datasets

d3rlpy provides datasets for experimenting data-driven deep reinforcement learning algorithms.

---

<i>d3rlpy.datasets.get_cartpole</i>	Returns cartpole dataset and environment.
<i>d3rlpy.datasets.get_pendulum</i>	Returns pendulum dataset and environment.
<i>d3rlpy.datasets.get_pybullet</i>	Returns pybullet dataset and environment.
<i>d3rlpy.datasets.get_atari</i>	Returns atari dataset and environment.
<i>d3rlpy.datasets.get_d4rl</i>	Returns d4rl dataset and environment.
<i>d3rlpy.datasets.get_dataset</i>	Returns dataset and environment by guessing from name.

---

### 3.4.1 *d3rlpy.datasets.get\_cartpole*

*d3rlpy.datasets.get\_cartpole*(*create\_mask=False*, *mask\_size=1*, *dataset\_type='replay'*)

Returns cartpole dataset and environment.

The dataset is automatically downloaded to *d3rlpy\_data/cartpole.h5* if it does not exist.

#### Parameters

- **create\_mask** (*bool*) – flag to create binary mask for bootstrapping.
- **mask\_size** (*int*) – ensemble size for binary mask.

- **dataset\_type** (*str*) – dataset type. Available options are ['replay', 'random'].

**Returns** tuple of *d3rlpy.dataset.MDPDataset* and gym environment.

**Return type** Tuple[*d3rlpy.dataset.MDPDataset*, gym.core.Env]

### 3.4.2 d3rlpy.datasets.get\_pendulum

*d3rlpy.datasets.get\_pendulum*(*create\_mask=False*, *mask\_size=1*, *dataset\_type='replay'*)

Returns pendulum dataset and environment.

The dataset is automatically downloaded to `d3rlpy_data/pendulum.h5` if it does not exist.

#### Parameters

- **create\_mask** (*bool*) – flag to create binary mask for bootstrapping.
- **mask\_size** (*int*) – ensemble size for binary mask.
- **dataset\_type** (*str*) – dataset type. Available options are ['replay', 'random'].

**Returns** tuple of *d3rlpy.dataset.MDPDataset* and gym environment.

**Return type** Tuple[*d3rlpy.dataset.MDPDataset*, gym.core.Env]

### 3.4.3 d3rlpy.datasets.get\_pybullet

*d3rlpy.datasets.get\_pybullet*(*env\_name*, *create\_mask=False*, *mask\_size=1*)

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.
- **create\_mask** (*bool*) – flag to create binary mask for bootstrapping.
- **mask\_size** (*int*) – ensemble size for binary mask.

**Returns** tuple of *d3rlpy.dataset.MDPDataset* and gym environment.

**Return type** Tuple[*d3rlpy.dataset.MDPDataset*, gym.core.Env]

### 3.4.4 d3rlpy.datasets.get\_atari

`d3rlpy.datasets.get_atari(env_name, create_mask=False, mask_size=1)`  
 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.
- `create_mask (bool)` – flag to create binary mask for bootstrapping.
- `mask_size (int)` – ensemble size for binary mask.

**Returns** tuple of `d3rlpy.dataset.MDPDataset` and gym environment.

**Return type** Tuple[`d3rlpy.dataset.MDPDataset`, gym.core.Env]

### 3.4.5 d3rlpy.datasets.get\_d4rl

`d3rlpy.datasets.get_d4rl(env_name, create_mask=False, mask_size=1)`  
 Returns d4rl dataset and environment.

The dataset is provided through d4rl.

```
from d3rlpy.datasets import get_d4rl
dataset, env = get_d4rl('hopper-medium-v0')
```

#### References

- Fu et al., D4RL: Datasets for Deep Data-Driven Reinforcement Learning.
- <https://github.com/rail-berkeley/d4rl>

#### Parameters

- `env_name (str)` – environment id of d4rl dataset.
- `create_mask (bool)` – flag to create binary mask for bootstrapping.
- `mask_size (int)` – ensemble size for binary mask.

**Returns** tuple of `d3rlpy.dataset.MDPDataset` and gym environment.

**Return type** Tuple[`d3rlpy.dataset.MDPDataset`, gym.core.Env]

### 3.4.6 d3rlpy.datasets.get\_dataset

`d3rlpy.datasets.get_dataset(env_name, create_mask=False, mask_size=1)`

Returns dataset and environment by guessing from name.

This function returns dataset by matching name with the following datasets.

- cartpole-replay
- cartpole-random
- pendulum-replay
- pendulum-random
- d4rl-pybullet
- d4rl-atari
- d4rl

```
import d3rlpy

# cartpole dataset
dataset, env = d3rlpy.datasets.get_dataset('cartpole')

# pendulum dataset
dataset, env = d3rlpy.datasets.get_dataset('pendulum')

# d4rl-pybullet dataset
dataset, env = d3rlpy.datasets.get_dataset('hopper-bullet-mixed-v0')

# d4rl-atari dataset
dataset, env = d3rlpy.datasets.get_dataset('breakout-mixed-v0')

# d4rl dataset
dataset, env = d3rlpy.datasets.get_dataset('hopper-medium-v0')
```

#### Parameters

- `env_name` (`str`) – environment id of the dataset.
- `create_mask` (`bool`) – flag to create binary mask for bootstrapping.
- `mask_size` (`int`) – ensemble size for binary mask.

**Returns** tuple of `d3rlpy.dataset.MDPDataset` and gym environment.

**Return type** Tuple[`d3rlpy.dataset.MDPDataset`, gym.core.Env]

## 3.5 Preprocessing

### 3.5.1 Observation

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(...)

# 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.

#### 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
```

(continues on next page)

(continued from previous page)

```
cql = CQL(scaler='pixel')
cql.fit(dataset.episodes)
```

## Methods

### `fit(episodes)`

Estimates scaling parameters from dataset.

**Parameters** `episodes` (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**Return type** `None`

### `fit_with_env(env)`

Gets scaling parameters from environment.

**Parameters** `env` (`gym.core.Env`) – gym environment.

**Return type** `None`

### `get_params(deep=False)`

Returns scaling parameters.

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

**Returns** scaler parameters.

**Return type** `Dict[str, Any]`

### `get_type()`

Returns a scaler type.

**Returns** scaler type.

**Return type** `str`

### `reverse_transform(x)`

Returns reversely transformed observations.

**Parameters** `x` (`torch.Tensor`) – observation.

**Returns** reversely transformed observation.

**Return type** `torch.Tensor`

### `transform(x)`

Returns processed observations.

**Parameters** `x` (`torch.Tensor`) – observation.

**Returns** processed observation.

**Return type** `torch.Tensor`

## Attributes

TYPE: ClassVar[str] = 'pixel'

### 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)

# 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.
- **maximum** (`Optional[numpy.ndarray]`) –
- **minimum** (`Optional[numpy.ndarray]`) –

## Methods

### `fit(episodes)`

Estimates scaling parameters from dataset.

**Parameters** `episodes` (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**Return type** `None`

### `fit_with_env(env)`

Gets scaling parameters from environment.

**Parameters** `env` (`gym.core.Env`) – gym environment.

**Return type** `None`

### `get_params(deep=False)`

Returns scaling parameters.

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

**Returns** scaler parameters.

**Return type** `Dict[str, Any]`

### `get_type()`

Returns a scaler type.

**Returns** scaler type.

**Return type** `str`

### `reverse_transform(x)`

Returns reversely transformed observations.

**Parameters** `x` (`torch.Tensor`) – observation.

**Returns** reversely transformed observation.

**Return type** `torch.Tensor`

### `transform(x)`

Returns processed observations.

**Parameters** `x` (`torch.Tensor`) – observation.

**Returns** processed observation.

**Return type** `torch.Tensor`

## Attributes

**TYPE:** `ClassVar[str] = 'min_max'`

## d3rlpy.preprocessing.StandardScaler

```
class d3rlpy.preprocessing.StandardScaler(dataset=None, mean=None, std=None, eps=0.001)
    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.
- **eps** (`float`) – small constant value to avoid zero-division.

### Methods

#### `fit(episodes)`

Estimates scaling parameters from dataset.

**Parameters** `episodes` (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**Return type** `None`

#### `fit_with_env(env)`

Gets scaling parameters from environment.

**Parameters** `env` (`gym.core.Env`) – gym environment.

**Return type** `None`

```
get_params(deep=False)
    Returns scaling parameters.

    Parameters deep (bool) – flag to deeply copy objects.

    Returns scalar parameters.

    Return type Dict[str, Any]

get_type()
    Returns a scalar type.

    Returns scalar type.

    Return type str

reverse_transform(x)
    Returns reversely transformed observations.

    Parameters x (torch.Tensor) – observation.

    Returns reversely transformed observation.

    Return type torch.Tensor

transform(x)
    Returns processed observations.

    Parameters x (torch.Tensor) – observation.

    Returns processed observation.

    Return type torch.Tensor
```

## Attributes

**TYPE:** ClassVar[str] = 'standard'

### 3.5.2 Action

d3rlpy also provides the feature that preprocesses continuous action. With this preprocessing, you don't need to normalize actions in advance or implement normalization in the environment side.

```
from d3rlpy.algos import CQL
from d3rlpy.dataset import MDPDataset

dataset = MDPDataset(...)

# 'min_max' or None
cql = CQL(action_scaler='min_max')

# action scaler is fitted from the given episodes
cql.fit(dataset.episodes)

# postprocessing is included in TorchScript
cql.save_policy('policy.pt')

# you don't need to take care of postprocessing at production
```

(continues on next page)

(continued from previous page)

```
policy = torch.jit.load('policy.pt')
action = policy(x)
```

You can also initialize scalers by yourself.

```
from d3rlpy.preprocessing import MinMaxActionScaler

action_scaler = MinMaxActionScaler(minimum=..., maximum=...)

cql = CQL(action_scaler=action_scaler)
```

---

<i>d3rlpy.preprocessing.MinMaxActionScaler</i>	Min-Max normalization action preprocessing.
--	---

---

### **d3rlpy.preprocessing.MinMaxActionScaler**

```
class d3rlpy.preprocessing.MinMaxActionScaler(dataset=None, maximum=None, minimum=None)
Min-Max normalization action preprocessing.
```

Actions will be normalized in range [-1.0, 1.0].

$$a' = (a - \min a) / (\max a - \min a) * 2 - 1$$

```
from d3rlpy.dataset import MDPDataset
from d3rlpy.algos import CQL

dataset = MDPDataset(observations, actions, rewards, terminals)

# initialize algorithm with MinMaxActionScaler
cql = CQL(action_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 MinMaxActionScaler

# initialize with dataset
scaler = MinMaxActionScaler(dataset)

# initialize manually
minimum = actions.min(axis=0)
maximum = actions.max(axis=0)
action_scaler = MinMaxActionScaler(minimum=minimum, maximum=maximum)

cql = CQL(action_scaler=action_scaler)
```

#### Parameters

- **dataset** (*d3rlpy.dataset.MDPDataset*) – dataset object.
- **min** (*numpy.ndarray*) – minimum values at each entry.

- **max** (`numpy.ndarray`) – maximum values at each entry.
- **maximum** (*Optional*[`numpy.ndarray`]) –
- **minimum** (*Optional*[`numpy.ndarray`]) –

## Methods

### `fit(episodes)`

Estimates scaling parameters from dataset.

**Parameters** `episodes` (`List[d3rlpy.dataset.Episode]`) – a list of episode objects.

**Return type** `None`

### `fit_with_env(env)`

Gets scaling parameters from environment.

**Parameters** `env` (`gym.core.Env`) – gym environment.

**Return type** `None`

### `get_params(deep=False)`

Returns action scaler params.

**Parameters** `deep` (`bool`) – flag to deepcopy parameters.

**Returns** action scaler parameters.

**Return type** `Dict[str, Any]`

### `get_type()`

Returns action scaler type.

**Returns** action scaler type.

**Return type** `str`

### `reverse_transform(action)`

Returns reversely transformed action.

**Parameters** `action` (`torch.Tensor`) – action vector.

**Returns** reversely transformed action.

**Return type** `torch.Tensor`

### `reverse_transform_numpy(action)`

Returns reversely transformed action in numpy array.

**Parameters** `action` (`numpy.ndarray`) – action vector.

**Returns** reversely transformed action.

**Return type** `numpy.ndarray`

### `transform(action)`

Returns processed action.

**Parameters** `action` (`torch.Tensor`) – action vector.

**Returns** processed action.

**Return type** `torch.Tensor`

## Attributes

TYPE: ClassVar[str] = 'min\_max'

### 3.5.3 Reward

d3rlpy also provides the feature that preprocesses rewards. With this preprocessing, you don't need to normalize rewards in advance. Note that this preprocessor should be fitted with the dataset. Afterwards you can use it with online training.

```
from d3rlpy.algos import CQL
from d3rlpy.dataset import MDPDataset

dataset = MDPDataset(...)

# 'min_max', 'standard' or None
cql = CQL(reward_scaler='standard')

# reward scaler is fitted from the given episodes
cql.fit(dataset.episodes)

# reward scaler is also available at finetuning.
cql.fit_online(env)
```

You can also initialize scalers by yourself.

```
from d3rlpy.preprocessing import MinMaxRewardScaler

reward_scaler = MinMaxRewardScaler(minimum=..., maximum=...)

cql = CQL(reward_scaler=reward_scaler)

# ClipRewardScaler is the only option you need to initialize manually
reward_scaler = ClipRewardScaler(-1.0, 1.0)
cql = CQL(reward_scaler=reward_scaler)
```

<code>d3rlpy.preprocessing.MinMaxRewardScaler</code>	Min-Max reward normalization preprocessing.
<code>d3rlpy.preprocessing.StandardRewardScaler</code>	Reward standardization preprocessing.
<code>d3rlpy.preprocessing.ClipRewardScaler</code>	Reward clipping preprocessing.

#### `d3rlpy.preprocessing.MinMaxRewardScaler`

`class d3rlpy.preprocessing.MinMaxRewardScaler(dataset=None, minimum=None, maximum=None)`  
Min-Max reward normalization preprocessing.

$$r' = (r - \min(r)) / (\max(r) - \min(r))$$

```
from d3rlpy.algos import CQL
cql = CQL(reward_scaler="min_max")
```

You can also initialize with `d3rlpy.dataset.MDPDataset` object or manually.

```
from d3rlpy.preprocessing import MinMaxRewardScaler

# initialize with dataset
scaler = MinMaxRewardScaler(dataset)

# initialize manually
scaler = MinMaxRewardScaler(minimum=0.0, maximum=10.0)

cql = CQL(scaler=scaler)
```

## Methods

### Parameters

- **dataset** (*Optional[d3rlpy.dataset.MDPDataset]*) –
- **minimum** (*Optional[float]*) –
- **maximum** (*Optional[float]*) –

#### `fit(episodes)`

Estimates scaling parameters from dataset.

**Parameters** `episodes` (*List[d3rlpy.dataset.Episode]*) – list of episodes.

**Return type** `None`

#### `fit_with_env(env)`

Gets scaling parameters from environment.

---

**Note:** RewardScaler does not support fitting with environment.

---

**Parameters** `env` (*gym.core.Env*) – gym environment.

**Return type** `None`

#### `get_params(deep=False)`

Returns scaling parameters.

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

**Returns** scaler parameters.

**Return type** `Dict[str, Any]`

#### `get_type()`

Returns a scaler type.

**Returns** scaler type.

**Return type** `str`

#### `reverse_transform(reward)`

Returns reversely processed rewards.

**Parameters** `reward` (*torch.Tensor*) – reward.

**Returns** reversely processed reward.

**Return type** torch.Tensor

**transform**(reward)  
Returns processed rewards.

**Parameters** reward (torch.Tensor) – reward.

**Returns** processed reward.

**Return type** torch.Tensor

**transform\_numpy**(reward)  
Returns transformed rewards in numpy array.

**Parameters** reward (numpy.ndarray) – reward.

**Returns** transformed reward.

**Return type** numpy.ndarray

## Attributes

TYPE: ClassVar[str] = 'min\_max'

### d3rlpy.preprocessing.StandardRewardScaler

class d3rlpy.preprocessing.StandardRewardScaler(dataset=None, mean=None, std=None, eps=0.001)  
Reward standardization preprocessing.

$$r' = (r - \mu)/\sigma$$

```
from d3rlpy.algos import CQL
cql = CQL(reward_scaler="standard")
```

You can also initialize with `d3rlpy.dataset.MDPDataset` object or manually.

```
from d3rlpy.preprocessing import StandardRewardScaler
# initialize with dataset
scaler = StandardRewardScaler(dataset)

# initialize manually
scaler = StandardRewardScaler(mean=0.0, std=1.0)

cql = CQL(scaler=scaler)
```

## Methods

### Parameters

- **dataset** (*Optional*[`d3rlpy.dataset.MDPDataset`]) –
- **mean** (*Optional*[`float`]) –
- **std** (*Optional*[`float`]) –
- **eps** (`float`) –

### `fit(episodes)`

Estimates scaling parameters from dataset.

**Parameters** `episodes` (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**Return type** `None`

### `fit_with_env(env)`

Gets scaling parameters from environment.

---

**Note:** RewardScaler does not support fitting with environment.

---

**Parameters** `env` (`gym.core.Env`) – gym environment.

**Return type** `None`

### `get_params(deep=False)`

Returns scaling parameters.

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

**Returns** scaler parameters.

**Return type** `Dict[str, Any]`

### `get_type()`

Returns a scaler type.

**Returns** scaler type.

**Return type** `str`

### `reverse_transform(reward)`

Returns reversely processed rewards.

**Parameters** `reward` (`torch.Tensor`) – reward.

**Returns** reversely processed reward.

**Return type** `torch.Tensor`

### `transform(reward)`

Returns processed rewards.

**Parameters** `reward` (`torch.Tensor`) – reward.

**Returns** processed reward.

**Return type** `torch.Tensor`

### `transform_numpy(reward)`

Returns transformed rewards in numpy array.

**Parameters** `reward` (`numpy.ndarray`) – reward.  
**Returns** transformed reward.  
**Return type** `numpy.ndarray`

## Attributes

**TYPE:** `ClassVar[str] = 'standard'`

## d3rlpy.preprocessing.ClipRewardScaler

**class** `d3rlpy.preprocessing.ClipRewardScaler(low=None, high=None)`  
Reward clipping preprocessing.

```
from d3rlpy.preprocessing import ClipRewardScaler

# clip rewards within [-1.0, 1.0]
reward_scaler = ClipRewardScaler(low=-1.0, high=1.0)

cql = CQL(reward_scaler=reward_scaler)
```

### Parameters

- `low` (`float`) – minimum value to clip.
- `high` (`float`) – maximum value to clip.

### Methods

**fit(episodes)**

Estimates scaling parameters from dataset.

**Parameters** `episodes` (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**Return type** `None`

**fit\_with\_env(env)**

Gets scaling parameters from environment.

---

**Note:** RewardScaler does not support fitting with environment.

---

**Parameters** `env` (`gym.core.Env`) – gym environment.

**Return type** `None`

**get\_params(deep=False)**

Returns scaling parameters.

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

**Returns** scaler parameters.

**Return type** `Dict[str, Any]`

```
get_type()
    Returns a scalar type.

    Returns scalar type.

    Return type str

reverse_transform(reward)
    Returns reversely processed rewards.

    Parameters reward (torch.Tensor) – reward.

    Returns reversely processed reward.

    Return type torch.Tensor

transform(reward)
    Returns processed rewards.

    Parameters reward (torch.Tensor) – reward.

    Returns processed reward.

    Return type torch.Tensor

transform_numpy(reward)
    Returns transformed rewards in numpy array.

    Parameters reward (numpy.ndarray) – reward.

    Returns transformed reward.

    Return type numpy.ndarray
```

## Attributes

```
TYPE: ClassVar[str] = 'clip'
```

## 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.models.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.models.optimizers import AdamFactory

# alias for Adam optimizer
```

(continues on next page)

(continued from previous page)

```
optim_factory = AdamFactory(weight_decay=1e-4)

dqn = DQN(optim_factory=optim_factory)
```

<code>d3rlpy.models.optimizers.OptimizerFactory</code>	A factory class that creates an optimizer object in a lazy way.
<code>d3rlpy.models.optimizers.SGDFactory</code>	An alias for SGD optimizer.
<code>d3rlpy.models.optimizers.AdamFactory</code>	An alias for Adam optimizer.
<code>d3rlpy.models.optimizers.RMSpropFactory</code>	An alias for RMSprop optimizer.

### 3.6.1 d3rlpy.models.optimizers.OptimizerFactory

`class d3rlpy.models.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 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` (`Union[Type[torch.optim.optimizer.Optimizer], str]`) – An optimizer class.
- `kwargs` (`Any`) – arbitrary keyword-arguments.

#### 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.

`Parameters` `deep` (`bool`) – flag to deeply copy the parameters.

`Returns` optimizer parameters.

`Return type` `Dict[str, Any]`

### 3.6.2 d3rlpy.models.optimizers.SGDFactory

```
class d3rlpy.models.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.
- **kwargs** (*Any*) –

#### 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.

**Parameters** **deep** (*bool*) – flag to deeply copy the parameters.

**Returns** optimizer parameters.

**Return type** Dict[*str*, *Any*]

### 3.6.3 d3rlpy.models.optimizers.AdamFactory

```
class d3rlpy.models.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[float, float]*) – 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.
- **kwargs** (*Any*) –

## 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.

**Parameters** **deep** (*bool*) – flag to deeply copy the parameters.

**Returns** optimizer parameters.

**Return type** Dict[str, Any]

## 3.6.4 d3rlpy.models.optimizers.RMSpropFactory

```
class d3rlpy.models.optimizers.RMSpropFactory(alpha=0.95, eps=0.01, 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.
- **kwargs** (*Any*) –

## 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.

**Parameters** **deep** (*bool*) – flag to deeply copy the parameters.

**Returns** optimizer parameters.

**Return type** Dict[str, Any]

## 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 netowrk architectures.

```
from d3rlpy.algos import DQN
from d3rlpy.models.encoders import VectorEncoderFactory

# encoder factory
encoder_factory = VectorEncoderFactory(hidden_units=[300, 400], activation='tanh')

# set EncoderFactory
dqn = DQN(encoder_factory=encoder_factory)
```

You can also build your own encoder factory.

```
import torch
import torch.nn as nn

from d3rlpy.models.encoders import EncoderFactory

# your own neural network
class CustomEncoder(nn.Module):
    def __init__(self, obsevation_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))
```

(continues on next page)

(continued from previous page)

```

    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):
        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 define action-conditioned networks such as Q-functions for continuous controls. `create` or `create_with_action` will be called depending on the function.

```

class CustomEncoderWithAction(nn.Module):
    def __init__(self, obsevation_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):
    TYPE = 'custom' # this is necessary

    def __init__(self, feature_size):
        self.feature_size = feature_size

    def create(self, observation_shape):
        return CustomEncoder(observation_shape, self.feature_size)

    def create_with_action(observation_shape, action_size, discrete_action):
        return CustomEncoderWithAction(observation_shape, action_size, self.feature_size)

```

(continues on next page)

(continued from previous page)

```
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.models.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.models.encoders.DefaultEncoderFactory</code>	Default encoder factory class.
<code>d3rlpy.models.encoders.PixelEncoderFactory</code>	Pixel encoder factory class.
<code>d3rlpy.models.encoders.VectorEncoderFactory</code>	Vector encoder factory class.
<code>d3rlpy.models.encoders.DenseEncoderFactory</code>	DenseNet encoder factory class.

### 3.7.1 d3rlpy.models.encoders.DefaultEncoderFactory

```
class d3rlpy.models.encoders.DefaultEncoderFactory(activation='relu', use_batch_norm=False,
                                                dropout_rate=None)
```

Default encoder factory class.

This encoder factory returns an encoder based on observation shape.

#### Parameters

- `activation (str)` – activation function name.
- `use_batch_norm (bool)` – flag to insert batch normalization layers.
- `dropout_rate (float)` – dropout probability.

## Methods

**create**(*observation\_shape*)

Returns PyTorch's state encoder module.

**Parameters** **observation\_shape** (*Sequence[int]*) – observation shape.

**Returns** an encoder object.

**Return type** d3rlpy.models.torch.encoders.Encoder

**create\_with\_action**(*observation\_shape*, *action\_size*, *discrete\_action=False*)

Returns PyTorch's state-action encoder module.

**Parameters**

- **observation\_shape** (*Sequence[int]*) – 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** d3rlpy.models.torch.encoders.EncoderWithAction

**get\_params**(*deep=False*)

Returns encoder parameters.

**Parameters** **deep** (*bool*) – flag to deeply copy the parameters.

**Returns** encoder parameters.

**Return type** Dict[str, Any]

**get\_type**()

Returns encoder type.

**Returns** encoder type.

**Return type** str

## Attributes

**TYPE:** ClassVar[str] = 'default'

### 3.7.2 d3rlpy.models.encoders.PixelEncoderFactory

```
class d3rlpy.models.encoders.PixelEncoderFactory(filters=None, feature_size=512, activation='relu',
                                                 use_batch_norm=False, dropout_rate=None)
```

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.

- **dropout\_rate** (`float`) – dropout probability.

## Methods

**create**(*observation\_shape*)

Returns PyTorch's state encoder module.

**Parameters** **observation\_shape** (`Sequence[int]`) – observation shape.

**Returns** an encoder object.

**Return type** `d3rlpy.models.torch.encoders.PixelEncoder`

**create\_with\_action**(*observation\_shape*, *action\_size*, *discrete\_action=False*)

Returns PyTorch's state-action encoder module.

**Parameters**

- **observation\_shape** (`Sequence[int]`) – 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** `d3rlpy.models.torch.encoders.PixelEncoderWithAction`

**get\_params**(*deep=False*)

Returns encoder parameters.

**Parameters** **deep** (`bool`) – flag to deeply copy the parameters.

**Returns** encoder parameters.

**Return type** `Dict[str, Any]`

**get\_type**()

Returns encoder type.

**Returns** encoder type.

**Return type** `str`

## Attributes

**TYPE:** `ClassVar[str] = 'pixel'`

### 3.7.3 `d3rlpy.models.encoders.VectorEncoderFactory`

```
class d3rlpy.models.encoders.VectorEncoderFactory(hidden_units=None, activation='relu',
                                                 use_batch_norm=False, dropout_rate=None,
                                                 use_dense=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.
- **use\_dense** (`bool`) – flag to use DenseNet architecture.
- **dropout\_rate** (`float`) – dropout probability.

## Methods

**create**(*observation\_shape*)

Returns PyTorch's state encoder module.

**Parameters** `observation_shape` (`Sequence[int]`) – observation shape.

**Returns** an encoder object.

**Return type** `d3rlpy.models.torch.encoders.VectorEncoder`

**create\_with\_action**(*observation\_shape*, *action\_size*, *discrete\_action=False*)

Returns PyTorch's state-action encoder module.

**Parameters**

- `observation_shape` (`Sequence[int]`) – 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** `d3rlpy.models.torch.encoders.VectorEncoderWithAction`

**get\_params**(*deep=False*)

Returns encoder parameters.

**Parameters** `deep` (`bool`) – flag to deeply copy the parameters.

**Returns** encoder parameters.

**Return type** `Dict[str, Any]`

**get\_type**()

Returns encoder type.

**Returns** encoder type.

**Return type** `str`

## Attributes

**TYPE:** `ClassVar[str] = 'vector'`

### 3.7.4 d3rlpy.models.encoders.DenseEncoderFactory

```
class d3rlpy.models.encoders.DenseEncoderFactory(activation='relu', use_batch_norm=False,
                                                dropout_rate=None)
```

DenseNet encoder factory class.

This is an alias for DenseNet architecture proposed in D2RL. This class does exactly same as follows.

```
from d3rlpy.encoders import VectorEncoderFactory

factory = VectorEncoderFactory(hidden_units=[256, 256, 256, 256],
                               use_dense=True)
```

For now, this only supports vector observations.

## References

- Sinha et al., D2RL: Deep Dense Architectures in Reinforcement Learning.

### Parameters

- **activation** (*str*) – activation function name.
- **use\_batch\_norm** (*bool*) – flag to insert batch normalization layers.
- **dropout\_rate** (*float*) – dropout probability.

### Methods

**create**(*observation\_shape*)

Returns PyTorch's state encoder module.

**Parameters** **observation\_shape** (*Sequence[int]*) – observation shape.

**Returns** an encoder object.

**Return type** d3rlpy.models.torch.encoders.VectorEncoder

**create\_with\_action**(*observation\_shape*, *action\_size*, *discrete\_action=False*)

Returns PyTorch's state-action encoder module.

**Parameters**

- **observation\_shape** (*Sequence[int]*) – 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** d3rlpy.models.torch.encoders.VectorEncoderWithAction

**get\_params**(*deep=False*)

Returns encoder parameters.

**Parameters** **deep** (*bool*) – flag to deeply copy the parameters.

**Returns** encoder parameters.

**Return type** Dict[str, Any]

---

**get\_type()**  
Returns encoder type.

**Returns** encoder type.

**Return type** str

### Attributes

TYPE: ClassVar[str] = 'dense'

## 3.8 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)
                        })
```

### 3.8.1 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_opp_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)`

Returns average TD error (in negative scale).

This metics 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.metrics.scorer.AlgoProtocol`) – algorithm.
- `episodes` (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**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)`

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.

$$\mathbb{E}_{s_t, a_t \sim D} [\sum_{t'=t} \gamma^{t'-t} A(s_{t'}, a_{t'})]$$

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.metrics.scorer.AlgoProtocol`) – algorithm.
- **episodes** (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**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)`

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.metrics.scorer.AlgoProtocol`) – algorithm.
- **episodes** (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**Returns** negative average value estimation.

**Return type** `float`

## `d3rlpy.metrics.scorer.value_estimation_std_scorer`

`d3rlpy.metrics.scorer.value_estimation_std_scorer(algo, episodes)`

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.

$$\mathbb{E}_{s_t \sim D, a \sim \text{argmax}_a Q_\theta(s_t, a)} [Q_{\text{std}}(s_t, a)]$$

where  $Q_{\text{std}}(s, a)$  is a standard deviation of action-value estimation over ensemble functions.

### Parameters

- **algo** (`d3rlpy.metrics.scorer.AlgoProtocol`) – algorithm.
- **episodes** (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**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)**

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.metrics.scorer.AlgoProtocol*) – algorithm.
- **episodes** (*List[d3rlpy.dataset.Episode]*) – list of episodes.

**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 funciton 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.

$$\mathbb{E}_{s,a \sim D_{success}}[Q(s,a)] - \mathbb{E}_{s,a \sim D}[Q(s,a)]$$

```
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.AlgoProtocol`, List[`d3rlpy.dataset.Episode`]], `float`]

### `d3rlpy.metrics.scorer.continuous_action_diff_scorer`

`d3rlpy.metrics.scorer.continuous_action_diff_scorer(algo, episodes)`

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.metrics.scorer.AlgoProtocol`) – algorithm.
- `episodes` (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**Returns** negative squared action difference.

**Return type** `float`

### `d3rlpy.metrics.scorer.discrete_action_match_scorer`

`d3rlpy.metrics.scorer.discrete_action_match_scorer(algo, episodes)`

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 episdoes are near-optimal, the large percentage would be better.

$$\frac{1}{N} \sum^N \parallel \{a_t = \text{argmax}_a Q_\theta(s_t, a)\}$$

**Parameters**

- `algo` (`d3rlpy.metrics.scorer.AlgoProtocol`) – algorithm.
- `episodes` (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**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.core.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[[...], float]

**d3rlpy.metrics.comparer.compare\_continuous\_action\_diff**

`d3rlpy.metrics.comparer.compare_continuous_action_diff(base_algo)`

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.metrics.scorer.AlgoProtocol`) – algorithm to compare with.

**Returns** scorer function.

**Return type** Callable[[`d3rlpy.metrics.scorer.AlgoProtocol`, List[`d3rlpy.dataset.Episode`]], float]

### `d3rlpy.metrics.comparer.compare_discrete_action_match`

`d3rlpy.metrics.comparer.compare_discrete_action_match(base_algo)`

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.metrics.scorer.AlgoProtocol`) – algorithm to compare with.

**Returns** scorer function.

**Return type** Callable[[`d3rlpy.metrics.scorer.AlgoProtocol`, List[`d3rlpy.dataset.Episode`]], float]

## 3.8.2 Dynamics

<code>d3rlpy.metrics.scorer.</code>	Returns MSE of observation prediction (in negative scale).
<code>d3rlpy.metrics.scorer.</code>	Returns MSE of reward prediction (in negative scale).
<code>d3rlpy.metrics.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)`

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.metrics.scorer.DynamicsProtocol`) – dynamics model.
- **episodes** (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**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)`

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.metrics.scorer.DynamicsProtocol`) – dynamics model.
- **episodes** (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**Returns** negative mean squared error.

**Return type** `float`

**d3rlpy.metrics.scorer.dynamics\_prediction\_variance\_scorer**

`d3rlpy.metrics.scorer.dynamics_prediction_variance_scorer(dynamics, episodes)`

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.metrics.scorer.DynamicsProtocol`) – dynamics model.
- **episodes** (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**Returns** negative variance.

**Return type** `float`

## 3.9 Off-Policy Evaluation

The off-policy evaluation is a method to estimate the trained policy performance only with offline datasets.

```
from d3rlpy.algos import CQL
from d3rlpy.datasets import get_pybullet

# prepare the trained algorithm
cql = CQL.from_json('<path-to-json>/params.json')
cql.load_model('<path-to-model>/model.pt')

# dataset to evaluate with
dataset, env = get_pybullet('hopper-bullet-mixed-v0')

from d3rlpy.ope import FQE

# off-policy evaluation algorithm
fqe = FQE(algo=cql)

# metrics to evaluate with
from d3rlpy.metrics.scorer import initial_state_value_estimation_scorer
from d3rlpy.metrics.scorer import soft_opc_scorer

# train estimators to evaluate the trained policy
fqe.fit(dataset.episodes,
        eval_episodes=dataset.episodes,
        scorers={
            'init_value': initial_state_value_estimation_scorer,
            'soft_opc': soft_opc_scorer(return_threshold=600)
        })
```

The evaluation during fitting is evaluating the trained policy.

### 3.9.1 For continuous control algorithms

---

[d3rlpy.ope.FQE](#)

---

Fitted Q Evaluation.

---

#### d3rlpy.ope.FQE

```
class d3rlpy.ope.FQE(*, algo=None, learning_rate=0.0001,
                      optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam', betas=(0.9,
                      0.999), eps=1e-08, weight_decay=0, amsgrad=False), encoder_factory='default',
                      q_func_factory='mean', batch_size=100, n_frames=1, n_steps=1, gamma=0.99,
                      n_critics=1, target_update_interval=100, use_gpu=False, scaler=None,
                      action_scaler=None, reward_scaler=None, impl=None, **kwargs)
```

Fitted Q Evaluation.

FQE is an off-policy evaluation method that approximates a Q function  $Q_\theta(s, a)$  with the trained policy  $\pi_\phi(s)$ .

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

The trained Q function in FQE will estimate evaluation metrics more accurately than learned Q function during training.

## References

- Le et al., Batch Policy Learning under Constraints.

### Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm to evaluate.
- **learning\_rate** (`float`) – learning rate.
- **optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory or str`) – optimizer factory.
- **encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **target\_update\_interval** (`int`) – interval to update the target network.
- **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'].
- **action\_scaler** (`d3rlpy.preprocessing.ActionScaler or str`) – action preprocessor. The available options are ['min\_max'].
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler or str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.metrics.ope.torch.FQEImpl`) – algorithm implementation.
- **kwargs** (`Any`) –

### Methods

#### `build_with_dataset(dataset)`

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

#### `build_with_env(env)`

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

**collect**(*env*, *buffer*=None, *explorer*=None, *n\_steps*=1000000, *show\_progress*=True, *timelimit\_aware*=True)  
 Collects data via interaction with environment.

If *buffer* is not given, `ReplayBuffer` will be internally created.

#### Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_steps** (`int`) – the number of total steps to train.
- **show\_progress** (`bool`) – flag to show progress bar for iterations.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag `False` when `TimeLimit`.  
`truncated` flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

**copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (*Sequence[int]*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

#### Return type `None`

**fit**(*dataset*, *n\_epochs=None*, *n\_steps=None*, *n\_steps\_per\_epoch=10000*, *save\_metrics=True*,  
*experiment\_name=None*, *with\_timestamp=True*, *logdir='d3rlpy\_logs'*, *verbose=True*,  
*show\_progress=True*, *tensorboard\_dir=None*, *eval\_episodes=None*, *save\_interval=1*, *scorers=None*,  
*shuffle=True*, *callback=None*)

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

#### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when *n\_steps* is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (*algo, epoch, total\_step*) , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** `List[Tuple[int, Dict[str, float]]]`

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
    n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
    save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
    logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
    timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.BatchBuffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

#### Return type `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
    logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
    timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

## Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

## Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when **n\_steps** is *None*.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if *None*, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (*algo, epoch, total\_step*), which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

#### 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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

#### generate\_new\_data(*transitions*)

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** *Optional[List[d3rlpy.dataset.Transition]]*

#### get\_action\_type()

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

#### 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[str, Any]

#### load\_model(*fname*)

Load neural network parameters.

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

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

**Return type** None

#### 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` (`Union[ndarray, List[Any]]`) – 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` (`Union[ndarray, List[Any]]`) – observations
- `action` (`Union[ndarray, List[Any]]`) – 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** `Union[ndarray, Tuple[ndarray, 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` (`Union[ndarray, List[Any]]`) – 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.

**Return type** `None`

**save\_params**(*logger*)

Saves configurations as `params.json`.

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

**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` (Any) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** d3rlpy.base.LearnableBase

**update**(batch)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** Dict[str, float]

## Attributes

**action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

**action\_size**

Action size.

**Returns** action size.

**Return type** Optional[int]

**active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

**batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** int

**gamma**

Discount factor.

**Returns** discount factor.

**Return type** float

**grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

**impl**

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

#### n\_frames

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

#### n\_steps

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

#### observation\_shape

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

#### reward\_scaler

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

#### scaler

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## 3.9.2 For discrete control algorithms

---

d3rlpy.ope.DiscreteFQE

Fitted Q Evaluation for discrete action-space.

---

### d3rlpy.ope.DiscreteFQE

```
class d3rlpy.ope.DiscreteFQE(*, algo=None, learning_rate=0.0001,
                             optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam',
                             betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False),
                             encoder_factory='default', q_func_factory='mean', batch_size=100,
                             n_frames=1, n_steps=1, gamma=0.99, n_critics=1,
                             target_update_interval=100, use_gpu=False, scaler=None,
                             action_scaler=None, reward_scaler=None, impl=None, **kwargs)
```

Fitted Q Evaluation for discrete action-space.

FQE is an off-policy evaluation method that approximates a Q function  $Q_\theta(s, a)$  with the trained policy  $\pi_\phi(s)$ .

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

The trained Q function in FQE will estimate evaluation metrics more accurately than learned Q function during training.

## References

- Le et al., Batch Policy Learning under Constraints.

### Parameters

- **algo** (`d3rlpy.algos.base.AlgoBase`) – algorithm to evaluate.
- **learning\_rate** (`float`) – learning rate.
- **optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory or str`) – optimizer factory.
- **encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory or str`) – encoder factory.
- **q\_func\_factory** (`d3rlpy.models.q_functions.QFunctionFactory or str`) – Q function factory.
- **batch\_size** (`int`) – mini-batch size.
- **n\_frames** (`int`) – the number of frames to stack for image observation.
- **n\_steps** (`int`) – N-step TD calculation.
- **gamma** (`float`) – discount factor.
- **n\_critics** (`int`) – the number of Q functions for ensemble.
- **target\_update\_interval** (`int`) – interval to update the target network.
- **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']
- **reward\_scaler** (`d3rlpy.preprocessing.RewardScaler or str`) – reward preprocessor. The available options are ['clip', 'min\_max', 'standard'].
- **impl** (`d3rlpy.metrics.ope.torch.FQEImpl`) – algorithm implementation.
- **action\_scaler** (`Optional[Union[d3rlpy.preprocessing.action_scalers.ActionScaler, str]]`) –
- **kwargs** (`Any`) –

### Methods

#### `build_with_dataset(dataset)`

Instantiate implementation object with MDPDataset object.

**Parameters** `dataset` (`d3rlpy.dataset.MDPDataset`) – dataset.

**Return type** `None`

#### `build_with_env(env)`

Instantiate implementation object with OpenAI Gym object.

**Parameters** `env` (`gym.core.Env`) – gym-like environment.

**Return type** `None`

**collect**(*env*, *buffer*=None, *explorer*=None, *n\_steps*=1000000, *show\_progress*=True, *timelimit\_aware*=True)

Collects data via interaction with environment.

If *buffer* is not given, `ReplayBuffer` will be internally created.

**Parameters**

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_steps** (`int`) – the number of total steps to train.
- **show\_progress** (`bool`) – flag to show progress bar for iterations.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag `False` when `TimeLimit`.  
truncated flag is `True`, which is designed to incorporate with `gym.wrappers.TimeLimit`.

**Returns** replay buffer with the collected data.

**Return type** `d3rlpy.online.buffers.Buffer`

**copy\_policy\_from**(*algo*)

Copies policy parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**copy\_q\_function\_from**(*algo*)

Copies Q-function parameters from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=100000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

**Parameters** `algo` (`d3rlpy.algos.base.AlgoBase`) – algorithm object.

**Return type** `None`

**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** (*Sequence[int]*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

#### Return type

**None**

```
fit(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
     experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
     show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
     shuffle=True, callback=None)
```

Trains with the given dataset.

```
algo.fit(episodes, n_steps=1000000)
```

#### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when *n\_steps* is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step) , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** *List[Tuple[int, Dict[str, float]]]*

```
fit_batch_online(env, buffer=None, explorer=None, n_epochs=1000, n_steps_per_epoch=1000,
                 n_updates_per_epoch=1000, eval_interval=10, eval_env=None, eval_epsilon=0.0,
                 save_metrics=True, save_interval=1, experiment_name=None, with_timestamp=True,
                 logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
                 timelimit_aware=True, callback=None)
```

Start training loop of batch online deep reinforcement learning.

#### Parameters

- **env** (`d3rlpy.envs.batch.BatchEnv`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.BatchBuffer]`) – replay buffer.
- **explorer** (`Optional[d3rlpy.online.explorers.Explorer]`) – action explorer.
- **n\_epochs** (`int`) – the number of epochs to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch.
- **update\_interval** – the number of steps per update.
- **n\_updates\_per\_epoch** (`int`) – the number of updates per epoch.
- **eval\_interval** (`int`) – the number of epochs before evaluation.
- **eval\_env** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`.truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

#### Return type `None`

```
fit_online(env, buffer=None, 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, save_interval=1, experiment_name=None, with_timestamp=True,
            logdir='d3rlpy_logs', verbose=True, show_progress=True, tensorboard_dir=None,
            timelimit_aware=True, callback=None)
```

Start training loop of online deep reinforcement learning.

## Parameters

- **env** (`gym.core.Env`) – gym-like environment.
- **buffer** (`Optional[d3rlpy.online.buffers.Buffer]`) – replay buffer.
- **explorer** (`Optional[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** (`Optional[gym.core.Env]`) – gym-like environment. If None, evaluation is skipped.
- **eval\_epsilon** (`float`) –  $\epsilon$ -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.
- **save\_interval** (`int`) – the number of epochs before saving models.
- **experiment\_name** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **timelimit\_aware** (`bool`) – flag to turn terminal flag False when `TimeLimit`. truncated flag is True, which is designed to incorporate with `gym.wrappers.TimeLimit`.
- **callback** (`Optional[Callable[[d3rlpy.online.iterators.AlgoProtocol, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)`, which is called at the end of epochs.

**Return type** `None`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

## Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when **n\_steps** is None.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]*) – list of scorer functions used with **eval\_episodes**.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step), which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

#### 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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

**generate\_new\_data**(*transitions*)

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** **transitions** (*List[d3rlpy.dataset.Transition]*) – list of transitions.

**Returns** list of new transitions.

**Return type** *Optional[List[d3rlpy.dataset.Transition]]*

**get\_action\_type**()

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** d3rlpy.constants.ActionSpace

**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[str, Any]

**load\_model**(*fname*)

Load neural network parameters.

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

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

**Return type** None

**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` (`Union[ndarray, List[Any]]`) – 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` (`Union[ndarray, List[Any]]`) – observations
- `action` (`Union[ndarray, List[Any]]`) – 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** `Union[ndarray, Tuple[ndarray, 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` (`Union[ndarray, List[Any]]`) – 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.

**Return type** `None`

**save\_params**(*logger*)

Saves configurations as `params.json`.

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**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](https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html) (for Python).
- [https://pytorch.org/tutorials/advanced/cpp\\_export.html](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.

**Return type** `None`

**set\_active\_logger**(*logger*)

Set active D3RLPyLogger object

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

**set\_grad\_step**(*grad\_step*)

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

**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` (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

**update**(*batch*)

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

## Attributes

**action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** `Optional[ActionScaler]`

**action\_size**

Action size.

**Returns** action size.

**Return type** `Optional[int]`

**active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

**batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** `int`

**gamma**

Discount factor.

**Returns** discount factor.

**Return type** `float`

**grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

**impl**

Implementation object.

**Returns** implementation object.  
**Return type** Optional[ImplBase]

**n\_frames**

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

**n\_steps**

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## 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')
```

save\_model method saves all parameters including optimizer states, which is useful when checking all the outputs or re-training from snapshots.

Once you save your model, you can load it via load\_model method. Before loading the model, the algorithm object must be initialized as follows.

```
dqn = DQN()

# initialize with dataset
dqn.build_with_dataset(dataset)

# initialize with environment
# dqn.build_with_env(env)

# load entire model parameters.
dqn.load_model('model.pt')
```

### 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 can be also automatically saved for tensorboard under the specified directory so that you can interactively visualize training metrics easily.

```
$ pip install tensorboard
$ tensorboard --logdir runs
```

This tensorboard logs can be enabled by passing *tensorboard\_dir=/path/to/log\_dir*.

```
# saving tensorboard data is disabled by default
dqn.fit(dataset.episodes, tensorboard_dir='runs')
```

## 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
```

(continues on next page)

(continued from previous page)

```
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)

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

dqn = DQN(use_gpu=Device(0)) # assign id=0
dqn = DQN(use_gpu=Device(1)) # assign id=1
```

## 3.13 Online Training

### 3.13.1 Standard 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 probabilistic policy
               # algorithms
               eval_env=eval_env,
               n_epochs=30,
               n_steps_per_epoch=1000,
               n_updates_per_epoch=100)
```

#### Replay Buffer

---

<i>d3rlpy.online.buffers.ReplayBuffer</i>	Standard Replay Buffer.
---	-------------------------

---

**d3rlpy.online.buffers.ReplayBuffer**

```
class d3rlpy.online.buffers.ReplayBuffer(maxlen, env=None, episodes=None, create_mask=False,  
                                         mask_size=1)
```

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.
- **create\_mask** (`bool`) – flag to create bootstrapping mask.
- **mask\_size** (`int`) – ensemble size for binary mask.

**Methods****`__len__()`**

**Return type** `int`

**`append(observation, action, reward, terminal, clip_episode=None)`**

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`) – action.
- **reward** (`float`) – reward.
- **terminal** (`float`) – terminal flag.
- **clip\_episode** (`Optional[bool]`) – flag to clip the current episode. If None, the episode is clipped based on `terminal`.

**Return type** `None`

**`append_episode(episode)`**

Append Episode object to buffer.

**Parameters** `episode` (`d3rlpy.dataset.Episode`) – episode.

**Return type** `None`

**`sample(batch_size, n_frames=1, n_steps=1, gamma=0.99)`**

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)
```

(continues on next page)

(continued from previous page)

```
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.
- **n\_steps** (`int`) – the number of steps before the next observation.
- **gamma** (`float`) – discount factor used in N-step return calculation.

**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`

### `to_mdp_dataset()`

Convert replay data into static dataset.

The length of the dataset can be longer than the length of the replay buffer because this conversion is done by tracing `Transition` objects.

**Returns** MDPDataset object.

**Return type** `d3rlpy.dataset.MDPDataset`

## Attributes

### `transitions`

Returns a FIFO queue of transitions.

**Returns** FIFO queue of transitions.

**Return type** `d3rlpy.online.buffers.FIFOQueue`

## Explorers

<code>d3rlpy.online.explorers. ConstantEpsilonGreedy</code>	$\epsilon$ -greedy explorer with constant $\epsilon$ .
<code>d3rlpy.online.explorers. LinearDecayEpsilonGreedy</code>	$\epsilon$ -greedy explorer with linear decay schedule.
<code>d3rlpy.online.explorers.NormalNoise</code>	Normal noise explorer.

**d3rlpy.online.explorers.ConstantEpsilonGreedy**

```
class d3rlpy.online.explorers.ConstantEpsilonGreedy(epsilon)
     $\epsilon$ -greedy explorer with constant  $\epsilon$ .
```

**Parameters** `epsilon` (`float`) – the constant  $\epsilon$ .

**Methods**

`sample(algo, x, step)`

**Parameters**

- `algo` (`d3rlpy.online.explorers._ActionProtocol`) –
- `x` (`numpy.ndarray`) –
- `step` (`int`) –

**Return type** `numpy.ndarray`

**d3rlpy.online.explorers.LinearDecayEpsilonGreedy**

```
class d3rlpy.online.explorers.LinearDecayEpsilonGreedy(start_epsilon=1.0, end_epsilon=0.1,
                                                       duration=1000000)
```

$\epsilon$ -greedy explorer with linear decay schedule.

**Parameters**

- `start_epsilon` (`float`) – the beginning  $\epsilon$ .
- `end_epsilon` (`float`) – the end  $\epsilon$ .
- `duration` (`int`) – the scheduling duration.

**Methods**

`compute_epsilon(step)`

Returns decayed  $\epsilon$ .

**Returns**  $\epsilon$ .

**Parameters** `step` (`int`) –

**Return type** `float`

`sample(algo, x, step)`

Returns  $\epsilon$ -greedy action.

**Parameters**

- `algo` (`d3rlpy.online.explorers._ActionProtocol`) – algorithm.
- `x` (`numpy.ndarray`) – observation.
- `step` (`int`) – current environment step.

**Returns**  $\epsilon$ -greedy action.

**Return type** `numpy.ndarray`

**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.

**Methods****sample**(*algo*, *x*, *step*)

Returns action with noise injection.

**Parameters**

- **algo** (*d3rlpy.online.explorers.\_ActionProtocol*) – algorithm.
- **x** (*numpy.ndarray*) – observation.
- **step** (*int*) –

**Returns** action with noise injection.

**Return type** *numpy.ndarray*

### 3.13.2 Batch Concurrent Training

d3rlpy supports computationally efficient batch concurrent training.

```
import gym

from d3rlpy.algos import DQN
from d3rlpy.envs import AsyncBatchEnv
from d3rlpy.online.buffers import BatchReplayBuffer
from d3rlpy.online.explorers import LinearDecayEpsilonGreedy

# this condition is necessary due to spawning processes
if __name__ == '__main__':
    env = AsyncBatchEnv([lambda: gym.make('CartPole-v0') for _ in range(10)])

    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 = BatchReplayBuffer(maxlen=1000000, env=env)

    # setup explorers
    explorer = LinearDecayEpsilonGreedy(start_epsilon=1.0,
```

(continues on next page)

(continued from previous page)

```

        end_epsilon=0.1,
        duration=10000)

# start training
dqn.fit_batch_online(env,
                     buffer,
                     explorer=explorer, # you don't need this with probabilistic_
                     ←policy algorithms
                     eval_env=eval_env,
                     n_epochs=30,
                     n_steps_per_epoch=1000,
                     n_updates_per_epoch=100)

```

For the environment wrapper, please see `d3rlpy.envs.AsyncBatchEnv` and `d3rlpy.envs.SyncBatchEnv`.

## Replay Buffer

---

`d3rlpy.online.buffers.BatchReplayBuffer`

Standard Replay Buffer for batch training.

---

### `d3rlpy.online.buffers.BatchReplayBuffer`

```
class d3rlpy.online.buffers.BatchReplayBuffer(maxlen, env, episodes=None, create_mask=False,
                                               mask_size=1)
```

Standard Replay Buffer for batch training.

#### Parameters

- `maxlen` (`int`) – the maximum number of data length.
- `n_envs` (`int`) – the number of environments.
- `env` (`gym.Env`) – gym-like environment to extract shape information.
- `episodes` (`list(d3rlpy.dataset.Episode)`) – list of episodes to initialize buffer
- `create_mask` (`bool`) – flag to create bootstrapping mask.
- `mask_size` (`int`) – ensemble size for binary mask.

#### Methods

`__len__()`

**Return type** `int`

`append(observations, actions, rewards, terminals, clip_episodes=None)`

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

- `observations` (`numpy.ndarray`) – observation.

- **actions** (`numpy.ndarray`) – action.
- **rewards** (`numpy.ndarray`) – reward.
- **terminals** (`numpy.ndarray`) – terminal flag.
- **clip\_episodes** (*Optional*[`numpy.ndarray`]) – flag to clip the current episode. If `None`, the episode is clipped based on `terminal`.

**Return type** `None`

**append\_episode**(*episode*)

Append Episode object to buffer.

**Parameters** `episode` (`d3rlpy.dataset.Episode`) – episode.

**Return type** `None`

**sample**(*batch\_size*, *n\_frames*=1, *n\_steps*=1, *gamma*=0.99)

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.
- **n\_steps** (`int`) – the number of steps before the next observation.
- **gamma** (`float`) – discount factor used in N-step return calculation.

**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`

**to\_mdp\_dataset()**

Convert replay data into static dataset.

The length of the dataset can be longer than the length of the replay buffer because this conversion is done by tracing `Transition` objects.

**Returns** MDPDataset object.

**Return type** `d3rlpy.dataset.MDPDataset`

## Attributes

### `transitions`

Returns a FIFO queue of transitions.

**Returns** FIFO queue of transitions.

**Return type** d3rlpy.online.buffers.FIFOQueue

## 3.14 (experimental) Model-based Algorithms

d3rlpy provides model-based reinforcement learning algorithms.

```
from d3rlpy.datasets import get_pendulum
from d3rlpy.dynamics import ProbabilisticEnsembleDynamics
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)

dynamics = d3rlpy.dynamics.ProbabilisticEnsembleDynamics(learning_rate=1e-4, use_
˓→gpu=True)

# same as algorithms
dynamics.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 and pass it to the model-based RL algorithm.

```
from d3rlpy.algos import MOPO

# load trained dynamics model
dynamics = ProbabilisticEnsembleDynamics.from_json('<path-to-params.json>/params.json')
dynamics.load_model('<path-to-model>/model_xx.pt')

# give mopo as generator argument.
mopo = MOPO(dynamics=dynamics)
```

### 3.14.1 Dynamics Model

---

`d3rlpy.dynamics.ProbabilisticEnsembleDynamics`Probabilistic ensemble dynamics.

---

#### d3rlpy.dynamics.ProbabilisticEnsembleDynamics

```
class d3rlpy.dynamics.ProbabilisticEnsembleDynamics(*, learning_rate=0.001, optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam', betas=(0.9, 0.999), eps=1e-08, weight_decay=0.0001, amsgrad=False), encoder_factory='default', batch_size=100, n_frames=1, n_ensembles=5, variance_type='max', discrete_action=False, scaler=None, action_scaler=None, reward_scaler=None, use_gpu=False, impl=None, **kwargs)
```

Probabilistic ensemble dynamics.

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  $T_{\theta}$ .

$$s_{t+1}, r_{t+1} \sim T_{\theta}(s_t, a_t)$$

where  $s_t \sim D$  for the first step, otherwise  $s_t$  is the previous generated observation, and  $a_t \sim \pi(\cdot | s_t)$ .

---

**Note:** Currently, `ProbabilisticEnsembleDynamics` 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.models.optimizers.OptimizerFactory`) – optimizer factory.
- **encoder\_factory** (`d3rlpy.models.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.
- **variance\_type** (`str`) – variance calculation type. The available options are `['max', 'data']`.
- **discrete\_action** (`bool`) – flag to take discrete actions.
- **scaler** (`d3rlpy.preprocessing.scalers.Scaler` or `str`) – preprocessor. The available options are `['pixel', 'min_max', 'standard']`.

- **action\_scaler** (*d3rlpy.preprocessing.ActionsCalbers or str*) – action preprocessor. The available options are [ 'min\_max' ].
- **reward\_scaler** (*d3rlpy.preprocessing.RewardScaler or str*) – reward preprocessor. The available options are [ 'clip', 'min\_max', 'standard' ].
- **use\_gpu** (*bool or d3rlpy.gpu.Device*) – flag to use GPU or device.
- **impl** (*d3rlpy.dynamics.torch.ProbabilisticEnsembleDynamicsImpl*) – dynamics implementation.
- **kwargs** (*Any*) –

## Methods

**build\_with\_dataset**(*dataset*)

Instantiate implementation object with MDPDataset object.

**Parameters** **dataset** (*d3rlpy.dataset.MDPDataset*) – dataset.

**Return type** *None*

**build\_with\_env**(*env*)

Instantiate implementation object with OpenAI Gym object.

**Parameters** **env** (*gym.core.Env*) – gym-like environment.

**Return type** *None*

**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** (*Sequence[int]*) – observation shape.
- **action\_size** (*int*) – dimension of action-space.

**Return type** *None*

**fit**(*dataset, n\_epochs=None, n\_steps=None, n\_steps\_per\_epoch=10000, save\_metrics=True, experiment\_name=None, with\_timestamp=True, logdir='d3rlpy\_logs', verbose=True, show\_progress=True, tensorboard\_dir=None, eval\_episodes=None, save\_interval=1, scorers=None, shuffle=True, callback=None*)

Trains with the given dataset.

algo.fit(episodes, n\_steps=1000000)

**Parameters**

- **dataset** (*Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]*) – list of episodes to train.
- **n\_epochs** (*Optional[int]*) – the number of epochs to train.
- **n\_steps** (*Optional[int]*) – the number of steps to train.
- **n\_steps\_per\_epoch** (*int*) – the number of steps per epoch. This value will be ignored when *n\_steps* is None.

- **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** (`Optional[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\_dir** (`Optional[str]`) – directory to save logged information in tensorboard (additional to the csv data). if `None`, the directory will not be created.
- **eval\_episodes** (`Optional[List[d3rlpy.dataset.Episode]]`) – list of episodes to test.
- **save\_interval** (`int`) – interval to save parameters.
- **scorers** (`Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]]`) – list of scorer functions used with `eval_episodes`.
- **shuffle** (`bool`) – flag to shuffle transitions on each epoch.
- **callback** (`Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]`) – callable function that takes `(algo, epoch, total_step)` , which is called every step.

**Returns** list of result tuples (epoch, metrics) per epoch.

**Return type** `List[Tuple[int, Dict[str, float]]]`

```
fitter(dataset, n_epochs=None, n_steps=None, n_steps_per_epoch=10000, save_metrics=True,
       experiment_name=None, with_timestamp=True, logdir='d3rlpy_logs', verbose=True,
       show_progress=True, tensorboard_dir=None, eval_episodes=None, save_interval=1, scorers=None,
       shuffle=True, callback=None)
```

**Iterate over epochs steps to train with the given dataset. At each** iteration algo methods and properties can be changed or queried.

```
for epoch, metrics in algo.fitter(episodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (`Union[List[d3rlpy.dataset.Episode], d3rlpy.dataset.MDPDataset]`) – list of episodes to train.
- **n\_epochs** (`Optional[int]`) – the number of epochs to train.
- **n\_steps** (`Optional[int]`) – the number of steps to train.
- **n\_steps\_per\_epoch** (`int`) – the number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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** (*Optional[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\_dir** (*Optional[str]*) – directory to save logged information in tensorboard (additional to the csv data). if None, the directory will not be created.
- **eval\_episodes** (*Optional[List[d3rlpy.dataset.Episode]]*) – list of episodes to test.
- **save\_interval** (*int*) – interval to save parameters.
- **scorers** (*Optional[Dict[str, Callable[[Any, List[d3rlpy.dataset.Episode]], float]]*) – list of scorer functions used with *eval\_episodes*.
- **shuffle** (*bool*) – flag to shuffle transitions on each epoch.
- **callback** (*Optional[Callable[[d3rlpy.base.LearnableBase, int, int], None]]*) – callable function that takes (algo, epoch, total\_step), which is called every step.

**Returns** iterator yielding current epoch and metrics dict.

**Return type** Generator[Tuple[int, Dict[str, float]], None, None]

#### **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** (*Optional[Union[bool, int, d3rlpy.gpu.Device]]*) – flag to use GPU, device ID or device.

**Returns** algorithm.

**Return type** d3rlpy.base.LearnableBase

#### **generate\_new\_data(transitions)**

Returns generated transitions for data augmentation.

This method is for model-based RL algorithms.

**Parameters** `transitions` (`List[d3rlpy.dataset.Transition]`) – list of transitions.

**Returns** list of new transitions.

**Return type** `Optional[List[d3rlpy.dataset.Transition]]`

**get\_action\_type()**

Returns action type (continuous or discrete).

**Returns** action type.

**Return type** `d3rlpy.constants.ActionSpace`

**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[str, Any]`

**load\_model(fname)**

Load neural network parameters.

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

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

**Return type** `None`

**predict(x, action, with\_variance=False, indices=None)**

Returns predicted observation and reward.

#### Parameters

- `x` (`Union[numpy.ndarray, List[Any]]`) – observation
- `action` (`Union[numpy.ndarray, List[Any]]`) – action
- `with_variance` (`bool`) – flag to return prediction variance.
- `indices` (`Optional[numpy.ndarray]`) – index of ensemble model to return.

**Returns** tuple of predicted observation and reward. If `with_variance` is True, the prediction variance will be added as the 3rd element.

**Return type** `Union[Tuple[numpy.ndarray, numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]]`

**save\_model(fname)**

Saves neural network parameters.

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

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

**Return type** `None`

`save_params(logger)`

Saves configurations as `params.json`.

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

`set_active_logger(logger)`

Set active D3RLPyLogger object

**Parameters** `logger` (`d3rlpy.logger.D3RLPyLogger`) – logger object.

**Return type** `None`

`set_grad_step(grad_step)`

Set total gradient step counter.

This method can be used to restart the middle of training with an arbitrary gradient step counter, which has effects on periodic functions such as the target update.

**Parameters** `grad_step` (`int`) – total gradient step counter.

**Return type** `None`

`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` (`Any`) – arbitrary inputs to set as attributes.

**Returns** itself.

**Return type** `d3rlpy.base.LearnableBase`

`update(batch)`

Update parameters with mini-batch of data.

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

**Returns** dictionary of metrics.

**Return type** `Dict[str, float]`

---

## Attributes

**action\_scaler**

Preprocessing action scaler.

**Returns** preprocessing action scaler.

**Return type** Optional[ActionScaler]

**action\_size**

Action size.

**Returns** action size.

**Return type** Optional[int]

**active\_logger**

Active D3RLPyLogger object.

This will be only available during training.

**Returns** logger object.

**batch\_size**

Batch size to train.

**Returns** batch size.

**Return type** int

**gamma**

Discount factor.

**Returns** discount factor.

**Return type** float

**grad\_step**

Total gradient step counter.

This value will keep counting after `fit` and `fit_online` methods finish.

**Returns** total gradient step counter.

**impl**

Implementation object.

**Returns** implementation object.

**Return type** Optional[ImplBase]

**n\_frames**

Number of frames to stack.

This is only for image observation.

**Returns** number of frames to stack.

**Return type** int

**n\_steps**

N-step TD backup.

**Returns** N-step TD backup.

**Return type** int

**observation\_shape**

Observation shape.

**Returns** observation shape.

**Return type** Optional[Sequence[int]]

**reward\_scaler**

Preprocessing reward scaler.

**Returns** preprocessing reward scaler.

**Return type** Optional[RewardScaler]

**scaler**

Preprocessing scaler.

**Returns** preprocessing scaler.

**Return type** Optional[Scaler]

## 3.15 Stable-Baselines3 Wrapper

d3rlpy provides a minimal wrapper to use [Stable-Baselines3 \(SB3\)](#) features, like utility helpers or SB3 algorithms to create datasets.

---

**Note:** This wrapper is far from complete, and only provide a minimal integration with SB3.

---

### 3.15.1 Convert SB3 replay buffer to d3rlpy dataset

A replay buffer from Stable-Baselines3 can be easily converted to a `d3rlpy.dataset.MDPDataset` using `to_mdp_dataset()` utility function.

```
import stable_baselines3 as sb3

from d3rlpy.algos import AWR
from d3rlpy.wrappers.sb3 import to_mdp_dataset

# Train an off-policy agent with SB3
model = sb3.SAC("MlpPolicy", "Pendulum-v0", learning_rate=1e-3, verbose=1)
model.learn(6000)

# Convert to d3rlpy MDPDataset
dataset = to_mdp_dataset(model.replay_buffer)
# The dataset can then be used to train a d3rlpy model
offline_model = AWR()
offline_model.fit(dataset.episodes, n_epochs=100)
```

### 3.15.2 Convert d3rlpy to use SB3 helpers

An agent from d3rlpy can be converted to use the SB3 interface (notably follow the interface of SB3 predict()). This allows to use SB3 helpers like evaluate\_policy.

```
import gym
from stable_baselines3.common.evaluation import evaluate_policy

from d3rlpy.algos import AWAC
from d3rlpy.wrappers.sb3 import SB3Wrapper

env = gym.make("Pendulum-v0")

# Define an offline RL model
offline_model = AWAC()
# Train it using for instance a dataset created by a SB3 agent (see above)
offline_model.fit(dataset.episodes, n_epochs=10)

# Use SB3 wrapper (convert `predict()` method to follow SB3 API)
# to have access to SB3 helpers
# d3rlpy model is accessible via `wrapped_model.algo`
wrapped_model = SB3Wrapper(offline_model)

observation = env.reset()

# We can now use SB3's predict style
# it returns the action and the hidden states (for RNN policies)
action, _ = wrapped_model.predict([observation], deterministic=True)
# The following is equivalent to offline_model.sample_action(obs)
action, _ = wrapped_model.predict([observation], deterministic=False)

# Evaluate the trained model using SB3 helper
mean_reward, std_reward = evaluate_policy(wrapped_model, env)

print(f"mean_reward={mean_reward} +/- {std_reward}")

# Call methods from the wrapped d3rlpy model
wrapped_model.sample_action([observation])
wrapped_model.fit(dataset.episodes, n_epochs=10)

# Set attributes
wrapped_model.n_epochs = 2
# wrapped_model.n_epochs points to d3rlpy wrapped_model.algo.n_epochs
assert wrapped_model.algo.n_epochs == 2
```



## COMMAND LINE INTERFACE

d3rlpy provides the convenient CLI tool.

### 4.1 plot

Plot the saved metrics by specifying paths:

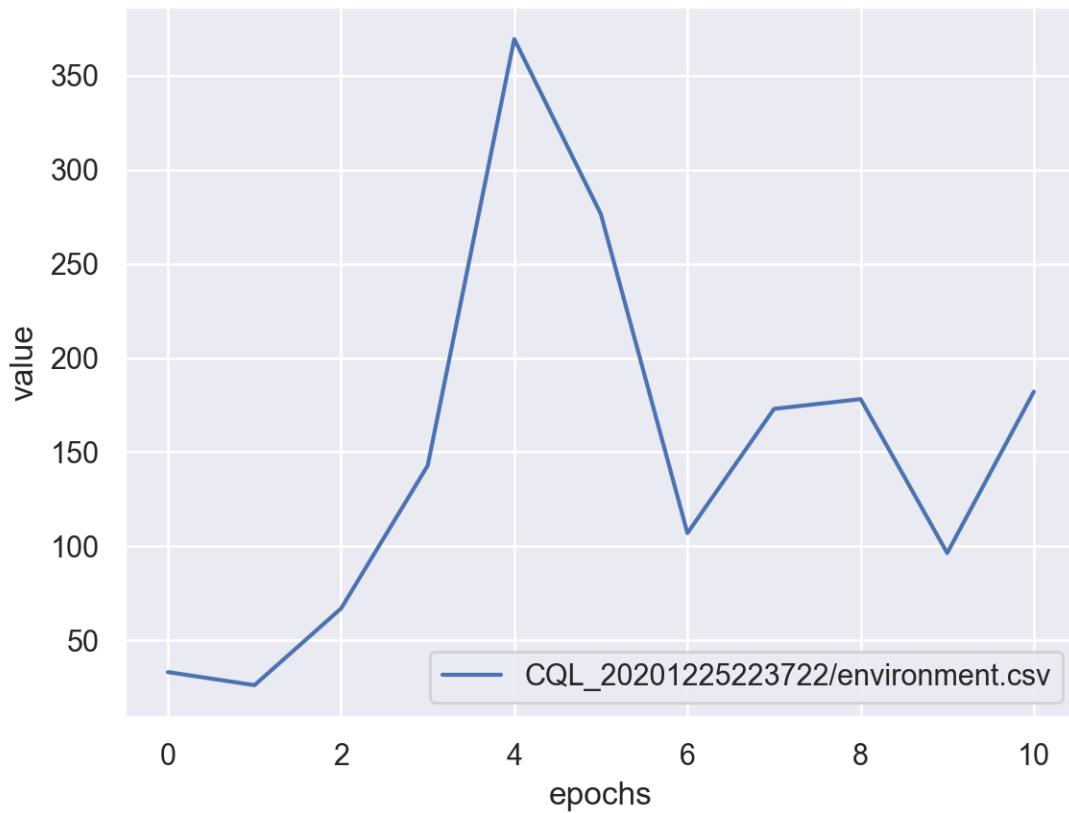
```
$ d3rlpy plot <path> [<path>...]
```

Table 1: options

option	description
--window	moving average window.
--show-steps	use iterations on x-axis.
--show-max	show maximum value.
--label	label in legend.
--xlim	limit on x-axis (tuple).
--ylim	limit on y-axis (tuple).
--title	title of the plot.

example:

```
$ d3rlpy plot d3rlpy_logs/CQL_20201224224314/environment.csv
```



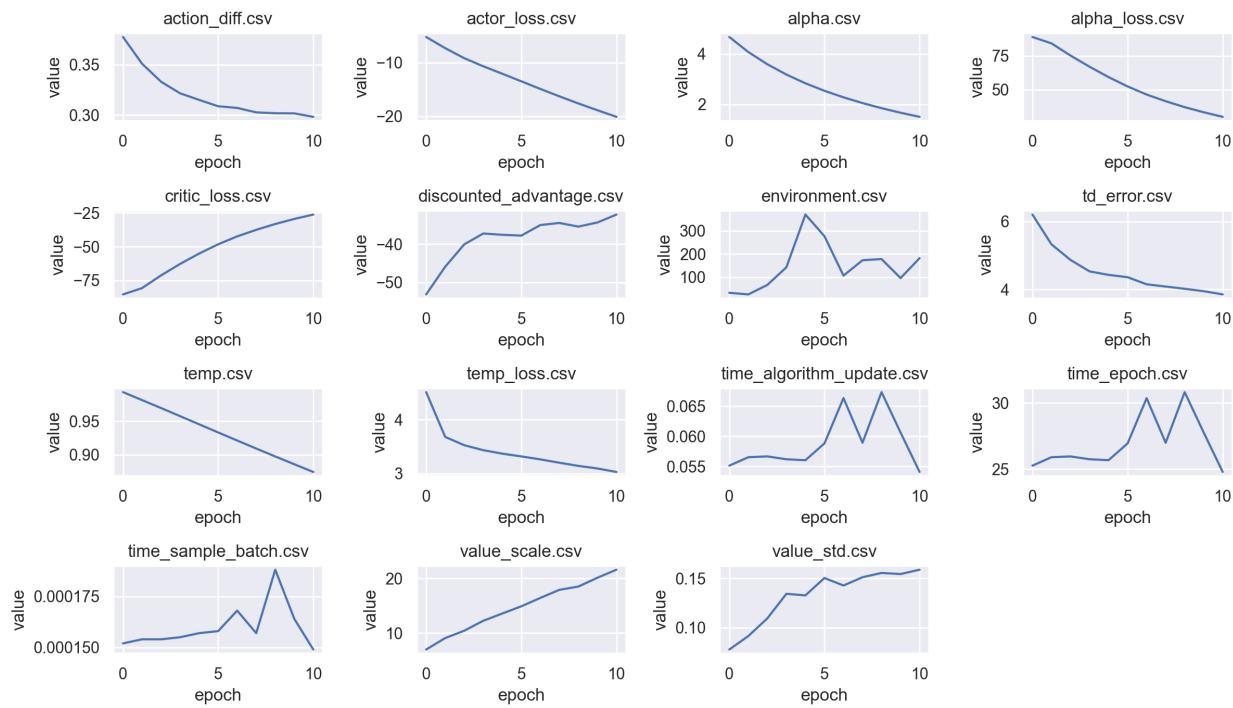
## 4.2 plot-all

Plot the all metrics saved in the directory:

```
$ d3rlpy plot-all <path>
```

example:

```
$ d3rlpy plot-all d3rlpy_logs/CQL_20201224224314
```



## 4.3 export

Export the saved model to the inference format, onnx and torchscript:

```
$ d3rlpy export <path>
```

Table 2: options

option	description
--format	model format (torchscript, onnx).
--params-json	explicitly specify params.json.
--out	output path.

example:

```
$ d3rlpy export d3rlpy_logs/CQL_20201224224314/model_100.pt
```

## 4.4 record

Record evaluation episodes as videos with the saved model:

```
$ d3rlpy record <path> --env-id <environment id>
```

Table 3: options

option	description
--env-id	Gym environment id.
--env-header	arbitrary Python code to define environment to evaluate.
--out	output directory.
--params-json	explicitly specify params.json
--n-episodes	the number of episodes to record.
--frame-rate	video frame rate.
--record-rate	images are recorded every record-rate frames.
--epsilon	$\epsilon$ -greedy evaluation.

example:

```
# record simple environment
$ d3rlpy record d3rlpy_logs/CQL_20201224224314/model_100.pt --env-id HopperBulletEnv-v0

# record wrapped environment
$ d3rlpy record d3rlpy_logs/Discrete_CQL_20201224224314/model_100.pt \
    --env-header 'import gym; from d3rlpy.envs import Atari; env = Atari(gym.make(
    "BreakoutNoFrameskip-v4"), is_eval=True)'
```

## 4.5 play

Run evaluation episodes with rendering:

```
$ d3rlpy play <path> --env-id <environment id>
```

Table 4: options

option	description
--env-id	Gym environment id.
--env-header	arbitrary Python code to define environment to evaluate.
--params-json	explicitly specify params.json
--n-episodes	the number of episodes to run.

example:

```
# record simple environment
$ d3rlpy play d3rlpy_logs/CQL_20201224224314/model_100.pt --env-id HopperBulletEnv-v0

# record wrapped environment
$ d3rlpy play d3rlpy_logs/Discrete_CQL_20201224224314/model_100.pt \
    --env-header 'import gym; from d3rlpy.envs import Atari; env = Atari(gym.make(
    "BreakoutNoFrameskip-v4"), is_eval=True)'
```

## INSTALLATION

### 5.1 Recommended Platforms

d3rlpy supports Linux, macOS and also Windows.

### 5.2 Install d3rlpy

#### 5.2.1 Install via PyPI

*pip* is a recommended way to install d3rlpy:

```
$ pip install d3rlpy
```

#### 5.2.2 Install via Anaconda

d3rlpy is also available on *conda-forge*:

```
$ conda install -c conda-forge d3rlpy
```

#### 5.2.3 Install via Docker

d3rlpy is also available on Docker Hub:

```
$ docker run -it --gpus all --name d3rlpy takuseno/d3rlpy:latest bash
```

#### 5.2.4 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 .
```



## 6.1 Reproducibility

Reproducibility is one of the most important things when doing research activity. Here is a simple example in d3rlpy.

```
import d3rlpy
import gym

# set random seeds in random module, numpy module and PyTorch module.
d3rlpy.seed(313)

# set environment seed
env = gym.make('Hopper-v2')
env.seed(313)
```

## 6.2 Create your own dataset

It's easy to create your own dataset with d3rlpy.

```
import d3rlpy

# vector observation
# 1000 steps of observations with shape of (100,)
observations = np.random.random((1000, 100))

# image observation
# 1000 steps of observations with shape of (3, 84, 84)
observations = np.random.randint(256, size=(1000, 3, 84, 84), dtype=np.uint8)

# 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 = d3rlpy.dataset.MDPDataset(observations, actions, rewards, terminals)

# train with your dataset
```

(continues on next page)

(continued from previous page)

```
cql = d3rlpy.algos.CQL()
cql.fit(dataset)
```

Please note that the `observations`, `actions`, `rewards` and `terminals` must be aligned with the same timestep.

```
observations = [s1, s2, s3, ...]
actions      = [a1, a2, a3, ...]
rewards       = [r1, r2, r3, ...]
terminals     = [t1, t2, t3, ...]
```

This alignment might be different from other libraries where the tuple of  $(s_t, a_t, r_{t+1})$  is saved. The advantage of d3rlpy's formulation is that we can explicitly store the last observation which might be useful for the future goal-oriented methods and less confusing.

If you have an access to the environment, you can automate the process.

```
import gym

env = gym.make("Hopper-v2")

# collect with random policy
random_policy = d3rlpy.algo.RandomPolicy()
random_buffer = d3rlpy.online.ReplayBuffer(100000, env=env)
random_policy.collect(env, buffer=random_buffer, n_steps=100000)
random_dataset = random_buffer.to_mdp_dataset()

# collect during training
sac = d3rlpy.algos.SAC()
replay_buffer = d3rlpy.online.ReplayBuffer(100000, env=env)
sac.fit_online(env, buffer=replay_buffer, n_steps=100000)
replay_dataset = replay_buffer.to_mdp_dataset()

# collect with the trained policy
medium_buffer = d3rlpy.online.ReplayBuffer(100000, env=env)
sac.collect(env, buffer=medium_buffer, n_steps=100000)
medium_dataset = medium_buffer.to_mdp_dataset()
```

Please check `MDPDataset` for more details.

## 6.3 Learning from image observation

d3rlpy supports both vector observations and image observations. There are several things you need to care about if you want to train RL agents from image observations.

```
from d3rlpy.dataset import MDPDataset

# observation MUST be uint8 array, and the channel-first images
observations = np.random.randint(256, size=(100000, 1, 84, 84), dtype=np.uint8)
actions = np.random.randint(4, size=100000)
rewards = np.random.random(100000)
terminals = np.random.randint(2, size=100000)
```

(continues on next page)

(continued from previous page)

```
dataset = MDPDataset(observations, actions, rewards, terminals)

from d3rlpy.algos import DQN

dqn = DQN(scaler='pixel', # you MUST set pixel scaler
           n_frames=4) # you CAN set the number of frames to stack
```

## 6.4 Improve performance beyond the original paper

d3rlpy provides many options that you can use to improve performance potentially beyond the original paper. All the options are powerful, but the best combinations and hyperparameters are always dependent on the tasks.

```
from d3rlpy.models.encoders import DefaultEncoderFactory
from d3rlpy.models.q_functions import QRQFunctionFactory
from d3rlpy.algos import DQN, SAC

# use batch normalization
# this seems to improve performance with discrete action-spaces
encoder = DefaultEncoderFactory(use_batch_norm=True)

dqn = DQN(encoder_factory=encoder,
           n_critics=5, # Q function ensemble size
           n_steps=5, # N-step TD backup
           q_func_factory='qr') # use distributional Q function

# use dropout
# this will dramatically improve performance
encoder = DefaultEncoderFactory(dropout_rate=0.2)

sac = SAC(actor_encoder_factory=encoder)
```



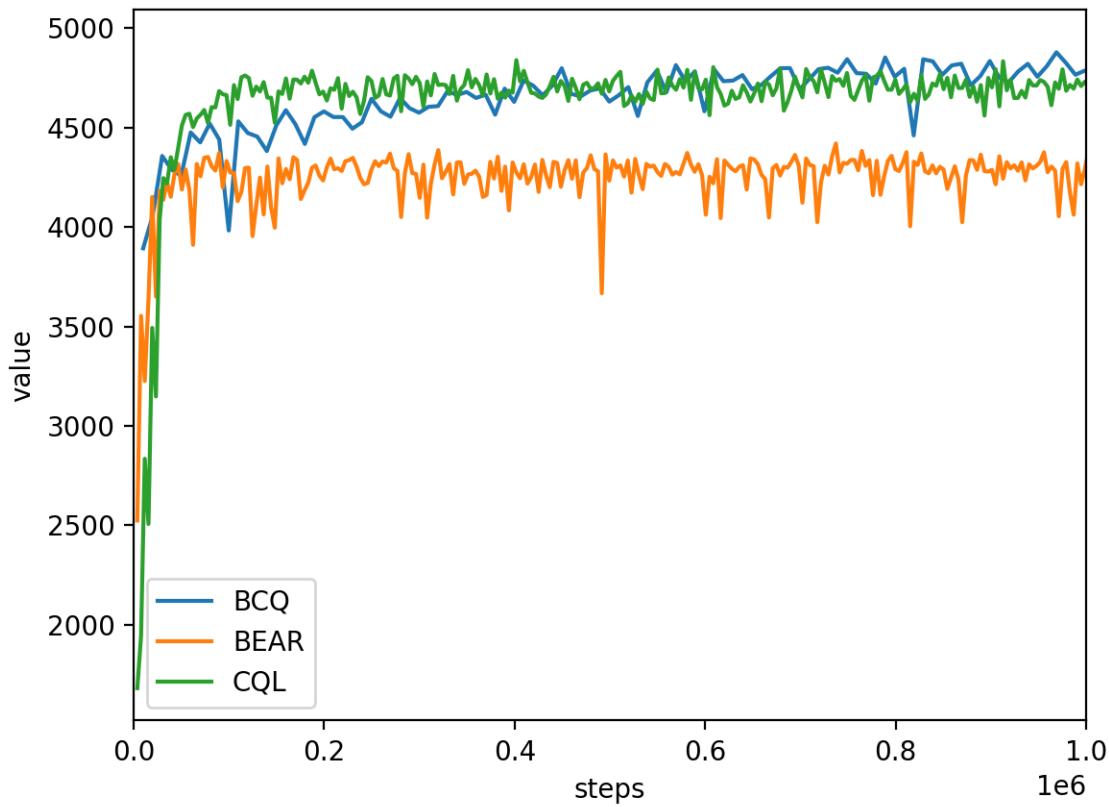
## PAPER REPRODUCTIONS

For the experiment code, please take a look at `reproductions` directory.

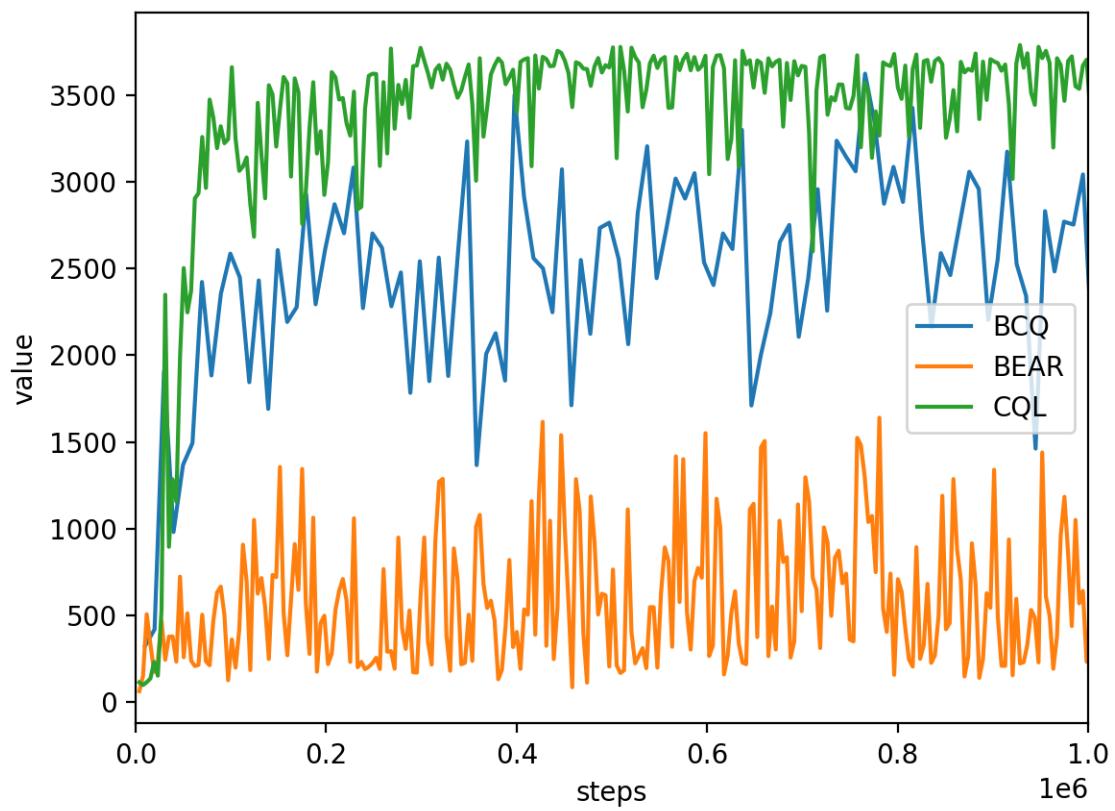
### 7.1 Offline

Fu et al., D4RL: Datasets for Deep Data-Driven Reinforcement Learning.

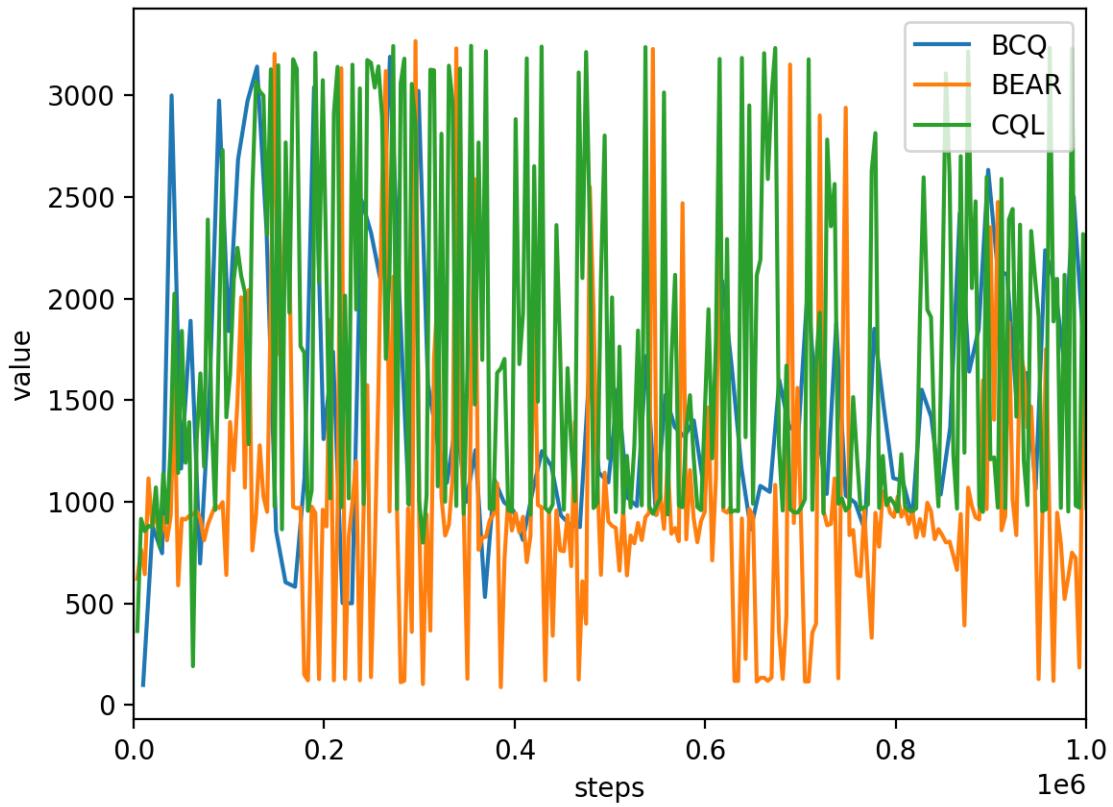
#### 7.1.1 halfcheetah-medium-v0



### 7.1.2 walker2d-medium-v0



### 7.1.3 hopper-medium-v0



## 7.2 Online

TBD.



---

**CHAPTER  
EIGHT**

---

**LICENSE**

MIT License

Copyright (c) 2021 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  
NINE**

---

**INDICES AND TABLES**

- genindex
- modindex
- search



## PYTHON MODULE INDEX

### d

`d3rlpy`, 9  
`d3rlpy.algos`, 9  
`d3rlpy.dataset`, 286  
`d3rlpy.datasets`, 297  
`d3rlpy.dynamics`, 368  
`d3rlpy.metrics`, 325  
`d3rlpy.models.encoders`, 318  
`d3rlpy.models.optimizers`, 314  
`d3rlpy.models.q_functions`, 280  
`d3rlpy.online`, 361  
`d3rlpy.ope`, 333  
`d3rlpy.preprocessing`, 301



# INDEX

## Symbols

`__getitem__()` (*d3rlpy.dataset.Episode* method), 292  
`__getitem__()` (*d3rlpy.dataset.MDPDataset* method), 288  
`__getitem__()` (*d3rlpy.dataset.TransitionMiniBatch* method), 295  
`__iter__()` (*d3rlpy.dataset.Episode* method), 292  
`__iter__()` (*d3rlpy.dataset.MDPDataset* method), 288  
`__iter__()` (*d3rlpy.dataset.TransitionMiniBatch* method), 295  
`__len__()` (*d3rlpy.dataset.Episode* method), 292  
`__len__()` (*d3rlpy.dataset.MDPDataset* method), 288  
`__len__()` (*d3rlpy.dataset.TransitionMiniBatch* method), 295  
`__len__()` (*d3rlpy.online.buffers.BatchReplayBuffer* method), 366  
`__len__()` (*d3rlpy.online.buffers.ReplayBuffer* method), 362

## A

`action` (*d3rlpy.dataset.Transition* attribute), 294  
`action_scaler` (*d3rlpy.algos.AWAC* attribute), 123  
`action_scaler` (*d3rlpy.algos.AWR* attribute), 112  
`action_scaler` (*d3rlpy.algos.BC* attribute), 18  
`action_scaler` (*d3rlpy.algos.BCQ* attribute), 65  
`action_scaler` (*d3rlpy.algos.BEAR* attribute), 77  
`action_scaler` (*d3rlpy.algos.COMBO* attribute), 181  
`action_scaler` (*d3rlpy.algos.CQL* attribute), 101  
`action_scaler` (*d3rlpy.algos.CRR* attribute), 89  
`action_scaler` (*d3rlpy.algos.DDPG* attribute), 30  
`action_scaler` (*d3rlpy.algos.DiscreteAWR* attribute), 269  
`action_scaler` (*d3rlpy.algos.DiscreteBC* attribute), 202  
`action_scaler` (*d3rlpy.algos.DiscreteBCQ* attribute), 247  
`action_scaler` (*d3rlpy.algos.DiscreteCQL* attribute), 258  
`action_scaler` (*d3rlpy.algos.DiscreteRandomPolicy* attribute), 279  
`action_scaler` (*d3rlpy.algos.DiscreteSAC* attribute), 235

`action_scaler` (*d3rlpy.algos.DoubleDQN* attribute), 224  
`action_scaler` (*d3rlpy.algos.DQN* attribute), 213  
`action_scaler` (*d3rlpy.algos.MOPO* attribute), 169  
`action_scaler` (*d3rlpy.algos.PLAS* attribute), 134  
`action_scaler` (*d3rlpy.algos.PLASWithPerturbation* attribute), 146  
`action_scaler` (*d3rlpy.algos.RandomPolicy* attribute), 191  
`action_scaler` (*d3rlpy.algos.SAC* attribute), 53  
`action_scaler` (*d3rlpy.algos.TD3* attribute), 41  
`action_scaler` (*d3rlpy.algos.TD3PlusBC* attribute), 157  
`action_scaler` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* attribute), 375  
`action_scaler` (*d3rlpy.ope.DiscreteFQE* attribute), 354  
`action_scaler` (*d3rlpy.ope.FQE* attribute), 343  
`action_size` (*d3rlpy.algos.AWAC* attribute), 123  
`action_size` (*d3rlpy.algos.AWR* attribute), 112  
`action_size` (*d3rlpy.algos.BC* attribute), 18  
`action_size` (*d3rlpy.algos.BCQ* attribute), 65  
`action_size` (*d3rlpy.algos.BEAR* attribute), 77  
`action_size` (*d3rlpy.algos.COMBO* attribute), 181  
`action_size` (*d3rlpy.algos.CQL* attribute), 101  
`action_size` (*d3rlpy.algos.CRR* attribute), 89  
`action_size` (*d3rlpy.algos.DDPG* attribute), 30  
`action_size` (*d3rlpy.algos.DiscreteAWR* attribute), 269  
`action_size` (*d3rlpy.algos.DiscreteBC* attribute), 202  
`action_size` (*d3rlpy.algos.DiscreteBCQ* attribute), 247  
`action_size` (*d3rlpy.algos.DiscreteCQL* attribute), 258  
`action_size` (*d3rlpy.algos.DiscreteRandomPolicy* attribute), 279  
`action_size` (*d3rlpy.algos.DiscreteSAC* attribute), 235  
`action_size` (*d3rlpy.algos.DoubleDQN* attribute), 224  
`action_size` (*d3rlpy.algos.DQN* attribute), 213  
`action_size` (*d3rlpy.algos.MOPO* attribute), 169  
`action_size` (*d3rlpy.algos.PLAS* attribute), 134  
`action_size` (*d3rlpy.algos.PLASWithPerturbation* attribute), 146  
`action_size` (*d3rlpy.algos.RandomPolicy* attribute), 191  
`action_size` (*d3rlpy.algos.SAC* attribute), 53

`action_size (d3rlpy.algos.TD3 attribute), 41`  
`action_size (d3rlpy.algos.TD3PlusBC attribute), 157`  
`action_size (d3rlpy.dynamics.ProbabilisticEnsembleDynamics attribute), 375`  
`action_size (d3rlpy.ope.DiscreteFQE attribute), 354`  
`action_size (d3rlpy.ope.FQE attribute), 343`  
`actions (d3rlpy.dataset.Episode attribute), 292`  
`actions (d3rlpy.dataset.MDPDataset attribute), 290`  
`actions (d3rlpy.dataset.TransitionMiniBatch attribute), 296`  
`active_logger (d3rlpy.algos.AWAC attribute), 123`  
`active_logger (d3rlpy.algos.AWR attribute), 112`  
`active_logger (d3rlpy.algos.BC attribute), 18`  
`active_logger (d3rlpy.algos.BCQ attribute), 65`  
`active_logger (d3rlpy.algos.BEAR attribute), 77`  
`active_logger (d3rlpy.algos.COMBO attribute), 181`  
`active_logger (d3rlpy.algos.CQL attribute), 101`  
`active_logger (d3rlpy.algos.CRR attribute), 89`  
`active_logger (d3rlpy.algos.DDPG attribute), 30`  
`active_logger (d3rlpy.algos.DiscreteAWR attribute), 269`  
`active_logger (d3rlpy.algos.DiscreteBC attribute), 202`  
`active_logger (d3rlpy.algos.DiscreteBCQ attribute), 247`  
`active_logger (d3rlpy.algos.DiscreteCQL attribute), 258`  
`active_logger (d3rlpy.algos.DiscreteRandomPolicy attribute), 279`  
`active_logger (d3rlpy.algos.DiscreteSAC attribute), 235`  
`active_logger (d3rlpy.algos.DoubleDQN attribute), 224`  
`active_logger (d3rlpy.algos.DQN attribute), 213`  
`active_logger (d3rlpy.algos.MOPO attribute), 169`  
`active_logger (d3rlpy.algos.PLAS attribute), 134`  
`active_logger (d3rlpy.algos.PLASWithPerturbation attribute), 146`  
`active_logger (d3rlpy.algos.RandomPolicy attribute), 191`  
`active_logger (d3rlpy.algos.SAC attribute), 53`  
`active_logger (d3rlpy.algos.TD3 attribute), 41`  
`active_logger (d3rlpy.algos.TD3PlusBC attribute), 157`  
`active_logger (d3rlpy.dynamics.ProbabilisticEnsembleDynamics attribute), 375`  
`active_logger (d3rlpy.ope.DiscreteFQE attribute), 354`  
`active_logger (d3rlpy.ope.FQE attribute), 343`  
`AdamFactory (class in d3rlpy.models.optimizers), 316`  
`add_additional_data() (d3rlpy.dataset.TransitionMiniBatch method), 295`  
`append() (d3rlpy.dataset.MDPDataset method), 288`  
`append() (d3rlpy.online.buffers.BatchReplayBuffer`  
`method), 366`  
`append() (d3rlpy.online.buffers.ReplayBuffer method), 362`  
`append_episode() (d3rlpy.online.buffers.BatchReplayBuffer method), 367`  
`append_episode() (d3rlpy.online.buffers.ReplayBuffer method), 362`  
`average_value_estimation_scorer() (in module d3rlpy.metrics.scorer), 327`  
`AWAC (class in d3rlpy.algos), 113`  
`AWR (class in d3rlpy.algos), 102`

## B

`batch_size (d3rlpy.algos.AWAC attribute), 123`  
`batch_size (d3rlpy.algos.AWR attribute), 112`  
`batch_size (d3rlpy.algos.BC attribute), 18`  
`batch_size (d3rlpy.algos.BCQ attribute), 65`  
`batch_size (d3rlpy.algos.BEAR attribute), 77`  
`batch_size (d3rlpy.algos.COMBO attribute), 181`  
`batch_size (d3rlpy.algos.CQL attribute), 101`  
`batch_size (d3rlpy.algos.CRR attribute), 89`  
`batch_size (d3rlpy.algos.DDPG attribute), 30`  
`batch_size (d3rlpy.algos.DiscreteAWR attribute), 269`  
`batch_size (d3rlpy.algos.DiscreteBC attribute), 202`  
`batch_size (d3rlpy.algos.DiscreteBCQ attribute), 247`  
`batch_size (d3rlpy.algos.DiscreteCQL attribute), 258`  
`batch_size (d3rlpy.algos.DiscreteRandomPolicy attribute), 279`  
`batch_size (d3rlpy.algos.DiscreteSAC attribute), 235`  
`batch_size (d3rlpy.algos.DoubleDQN attribute), 224`  
`batch_size (d3rlpy.algos.DQN attribute), 213`  
`batch_size (d3rlpy.algos.MOPO attribute), 169`  
`batch_size (d3rlpy.algos.PLAS attribute), 135`  
`batch_size (d3rlpy.algos.PLASWithPerturbation attribute), 146`  
`batch_size (d3rlpy.algos.RandomPolicy attribute), 191`  
`batch_size (d3rlpy.algos.SAC attribute), 53`  
`batch_size (d3rlpy.algos.TD3 attribute), 41`  
`batch_size (d3rlpy.algos.TD3PlusBC attribute), 157`  
`batch_size (d3rlpy.dynamics.ProbabilisticEnsembleDynamics attribute), 375`  
`batch_size (d3rlpy.ope.DiscreteFQE attribute), 354`  
`batch_size (d3rlpy.ope.FQE attribute), 343`  
`BatchReplayBuffer (class in d3rlpy.online.buffers), 366`  
`BC (class in d3rlpy.algos), 9`  
`BCQ (class in d3rlpy.algos), 54`  
`BEAR (class in d3rlpy.algos), 66`  
`bootstrap (d3rlpy.models.q_functions.FQFQFunctionFactory attribute), 286`  
`bootstrap (d3rlpy.models.q_functions.IQNQFunctionFactory attribute), 284`  
`bootstrap (d3rlpy.models.q_functions.MeanQFunctionFactory attribute), 282`

<code>bootstrap(d3rlpy.models.q_functions.QRQFunctionFactory attribute), 283</code>	<code>(d3rlpy.dynamics.ProbabilisticEnsembleDynamics method), 370</code>
<code>build_episodes() (d3rlpy.dataset.MDPDataset method), 288</code>	<code>build_with_dataset() (d3rlpy.ope.DiscreteFQE method), 345</code>
<code>build_transitions() (d3rlpy.dataset.Episode method), 292</code>	<code>build_with_dataset() (d3rlpy.ope.FQE method), 334</code>
<code>build_with_dataset() (d3rlpy.algos.AWAC method), 114</code>	<code>build_with_env() (d3rlpy.algos.AWAC method), 114</code>
<code>build_with_dataset() (d3rlpy.algos.AWR method), 103</code>	<code>build_with_env() (d3rlpy.algos.AWR method), 103</code>
<code>build_with_dataset() (d3rlpy.algos.BC method), 56</code>	<code>build_with_env() (d3rlpy.algos.BC method), 10</code>
<code>build_with_dataset() (d3rlpy.algos.BCQ method), 68</code>	<code>build_with_env() (d3rlpy.algos.BCQ method), 56</code>
<code>build_with_dataset() (d3rlpy.algos.COMBO method), 172</code>	<code>build_with_env() (d3rlpy.algos.BEAR method), 68</code>
<code>build_with_dataset() (d3rlpy.algos.CQL method), 92</code>	<code>build_with_env() (d3rlpy.algos.COMBO method), 172</code>
<code>build_with_dataset() (d3rlpy.algos.CRR method), 80</code>	<code>build_with_env() (d3rlpy.algos.CQL method), 92</code>
<code>build_with_dataset() (d3rlpy.algos.DDPG method), 21</code>	<code>build_with_env() (d3rlpy.algos.CRR method), 80</code>
<code>build_with_dataset() (d3rlpy.algos.DiscreteAWR method), 261</code>	<code>build_with_env() (d3rlpy.algos.DDPG method), 21</code>
<code>build_with_dataset() (d3rlpy.algos.DiscreteBC method), 194</code>	<code>build_with_env() (d3rlpy.algos.DiscreteAWR method), 261</code>
<code>build_with_dataset() (d3rlpy.algos.DiscreteBCQ method), 238</code>	<code>build_with_env() (d3rlpy.algos.DiscreteBC method), 194</code>
<code>build_with_dataset() (d3rlpy.algos.DiscreteCQL method), 249</code>	<code>build_with_env() (d3rlpy.algos.DiscreteBCQ method), 238</code>
<code>build_with_dataset() (d3rlpy.algos.DiscreteRandomPolicy method), 270</code>	<code>build_with_env() (d3rlpy.algos.DiscreteCQL method), 249</code>
<code>build_with_dataset() (d3rlpy.algos.DoubleDQN method), 227</code>	<code>build_with_env() (d3rlpy.algos.DiscreteRandomPolicy method), 270</code>
<code>build_with_dataset() (d3rlpy.algos.DoubleDQN method), 215</code>	<code>build_with_env() (d3rlpy.algos.DoubleDQN method), 227</code>
<code>build_with_dataset() (d3rlpy.algos.DQN method), 204</code>	<code>build_with_env() (d3rlpy.algos.DoubleDQN method), 215</code>
<code>build_with_dataset() (d3rlpy.algos.MOPO method), 161</code>	<code>build_with_env() (d3rlpy.algos.DQN method), 204</code>
<code>build_with_dataset() (d3rlpy.algos.PLAS method), 126</code>	<code>build_with_env() (d3rlpy.algos.MOPO method), 161</code>
<code>build_with_dataset() (d3rlpy.algos.PLASWithPerturbation method), 137</code>	<code>build_with_env() (d3rlpy.algos.PLAS method), 126</code>
<code>build_with_dataset() (d3rlpy.algos.RandomPolicy method), 183</code>	<code>build_with_env() (d3rlpy.algos.PLASWithPerturbation method), 137</code>
<code>build_with_dataset() (d3rlpy.algos.SAC method), 44</code>	<code>build_with_env() (d3rlpy.algos.RandomPolicy method), 183</code>
<code>build_with_dataset() (d3rlpy.algos.TD3 method), 33</code>	<code>build_with_env() (d3rlpy.algos.SAC method), 44</code>
<code>build_with_dataset() (d3rlpy.algos.TD3PlusBC method), 149</code>	<code>build_with_env() (d3rlpy.algos.TD3 method), 33</code>
<code>build_with_dataset()</code>	<code>build_with_env() (d3rlpy.algos.TD3PlusBC method), 149</code>
	<code>build_with_env() (d3rlpy.dynamics.ProbabilisticEnsembleDynamics method), 370</code>
	<code>build_with_env() (d3rlpy.ope.DiscreteFQE method), 345</code>
	<code>build_with_env() (d3rlpy.ope.FQE method), 334</code>
	<b>C</b>
	<code>clear_links() (d3rlpy.dataset.Transition method), 293</code>
	<code>ClipRewardScaler (class in d3rlpy.preprocessing), 313</code>
	<code>collect() (d3rlpy.algos.AWAC method), 114</code>
	<code>collect() (d3rlpy.algos.AWR method), 103</code>
	<code>collect() (d3rlpy.algos.BC method), 10</code>
	<code>collect() (d3rlpy.algos.BCQ method), 56</code>
	<code>collect() (d3rlpy.algos.BEAR method), 68</code>

<code>collect()</code> ( <i>d3rlpy.algos.COMBO method</i> ), 172	<code>copy_policy_from()</code> ( <i>d3rlpy.algos.DiscreteBCQ method</i> ), 238
<code>collect()</code> ( <i>d3rlpy.algos.CQL method</i> ), 92	<code>copy_policy_from()</code> ( <i>d3rlpy.algos.DiscreteCQL method</i> ), 249
<code>collect()</code> ( <i>d3rlpy.algos.CRR method</i> ), 80	<code>copy_policy_from()</code> ( <i>d3rlpy.algos.DiscreteRandomPolicy method</i> ), 271
<code>collect()</code> ( <i>d3rlpy.algos.DDPG method</i> ), 21	<code>copy_policy_from()</code> ( <i>d3rlpy.algos.DiscreteSAC method</i> ), 227
<code>collect()</code> ( <i>d3rlpy.algos.DiscreteAWR method</i> ), 261	<code>copy_policy_from()</code> ( <i>d3rlpy.algos.DoubleDQN method</i> ), 215
<code>collect()</code> ( <i>d3rlpy.algos.DiscreteBC method</i> ), 194	<code>copy_policy_from()</code> ( <i>d3rlpy.algos.DQN method</i> ), 204
<code>collect()</code> ( <i>d3rlpy.algos.DiscreteBCQ method</i> ), 238	<code>copy_policy_from()</code> ( <i>d3rlpy.algos.MOPO method</i> ), 161
<code>collect()</code> ( <i>d3rlpy.algos.DiscreteCQL method</i> ), 249	<code>copy_policy_from()</code> ( <i>d3rlpy.algos.PLAS method</i> ), 126
<code>collect()</code> ( <i>d3rlpy.algos.DiscreteRandomPolicy method</i> ), 270	<code>copy_policy_from()</code> ( <i>d3rlpy.algos.PLASWithPerturbation method</i> ), 138
<code>collect()</code> ( <i>d3rlpy.algos.DiscreteSAC method</i> ), 227	<code>copy_policy_from()</code> ( <i>d3rlpy.algos.RandomPolicy method</i> ), 183
<code>collect()</code> ( <i>d3rlpy.algos.DoubleDQN method</i> ), 215	<code>copy_policy_from()</code> ( <i>d3rlpy.algos.SAC method</i> ), 45
<code>collect()</code> ( <i>d3rlpy.algos.DQN method</i> ), 204	<code>copy_policy_from()</code> ( <i>d3rlpy.algos.TD3 method</i> ), 33
<code>collect()</code> ( <i>d3rlpy.algos.MOPO method</i> ), 161	<code>copy_policy_from()</code> ( <i>d3rlpy.algos.TD3PlusBC method</i> ), 149
<code>collect()</code> ( <i>d3rlpy.algos.PLAS method</i> ), 126	<code>copy_policy_from()</code> ( <i>d3rlpy.ope.DiscreteFQE method</i> ), 346
<code>collect()</code> ( <i>d3rlpy.algos.PLASWithPerturbation method</i> ), 137	<code>copy_policy_from()</code> ( <i>d3rlpy.ope.FQE method</i> ), 335
<code>collect()</code> ( <i>d3rlpy.algos.RandomPolicy method</i> ), 183	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.AWAC method</i> ), 115
<code>collect()</code> ( <i>d3rlpy.algos.SAC method</i> ), 44	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.AWR method</i> ), 101
<code>collect()</code> ( <i>d3rlpy.algos.TD3 method</i> ), 33	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.BC method</i> ), 11
<code>collect()</code> ( <i>d3rlpy.algos.TD3PlusBC method</i> ), 149	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.BCQ method</i> ), 57
<code>collect()</code> ( <i>d3rlpy.ope.DiscreteFQE method</i> ), 346	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.BEAR method</i> ), 69
<code>collect()</code> ( <i>d3rlpy.ope.FQE method</i> ), 335	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.COMBO method</i> ), 173
<code>COMBO</code> ( <i>class in d3rlpy.algos</i> ), 171	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.CQL method</i> ), 93
<code>compare_continuous_action_diff()</code> ( <i>in module d3rlpy.metrics.comparer</i> ), 330	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.CRR method</i> ), 81
<code>compare_discrete_action_match()</code> ( <i>in module d3rlpy.metrics.comparer</i> ), 331	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DDPG method</i> ), 22
<code>compute_epsilon()</code> ( <i>d3rlpy.online.explorers.LinearDecayEpsilonGreedy method</i> ), 364	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DiscreteAWR method</i> ), 261
<code>compute_return()</code> ( <i>d3rlpy.dataset.Episode method</i> ), 292	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DiscreteBC method</i> ), 194
<code>compute_stats()</code> ( <i>d3rlpy.dataset.MDPDataset method</i> ), 288	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DiscreteBCQ method</i> ), 239
<code>ConstantEpsilonGreedy</code> ( <i>class in d3rlpy.online.explorers</i> ), 364	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DiscreteCQL method</i> ), 250
<code>continuous_action_diff_scorer()</code> ( <i>in module d3rlpy.metrics.scorer</i> ), 329	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DiscreteRandomPolicy method</i> ), 271
<code>copy_policy_from()</code> ( <i>d3rlpy.algos.AWAC method</i> ), 115	
<code>copy_policy_from()</code> ( <i>d3rlpy.algos.AWR method</i> ), 104	
<code>copy_policy_from()</code> ( <i>d3rlpy.algos.BC method</i> ), 11	
<code>copy_policy_from()</code> ( <i>d3rlpy.algos.BCQ method</i> ), 57	
<code>copy_policy_from()</code> ( <i>d3rlpy.algos.BEAR method</i> ), 69	
<code>copy_policy_from()</code> ( <i>d3rlpy.algos.COMBO method</i> ), 173	
<code>copy_policy_from()</code> ( <i>d3rlpy.algos.CQL method</i> ), 92	
<code>copy_policy_from()</code> ( <i>d3rlpy.algos.CRR method</i> ), 80	
<code>copy_policy_from()</code> ( <i>d3rlpy.algos.DDPG method</i> ), 21	
<code>copy_policy_from()</code> ( <i>d3rlpy.algos.DiscreteAWR method</i> ), 261	
<code>copy_policy_from()</code> ( <i>d3rlpy.algos.DiscreteBC method</i> ), 194	

**copy\_q\_function\_from()** (*d3rlpy.algos.DiscreteSAC method*), 227  
**copy\_q\_function\_from()** (*d3rlpy.algos.DoubleDQN method*), 216  
**copy\_q\_function\_from()** (*d3rlpy.algos.DQN method*), 205  
**copy\_q\_function\_from()** (*d3rlpy.algos.MOPO method*), 161  
**copy\_q\_function\_from()** (*d3rlpy.algos.PLAS method*), 127  
**copy\_q\_function\_from()** (*d3rlpy.algos.PLASWithPerturbation method*), 138  
**copy\_q\_function\_from()** (*d3rlpy.algos.RandomPolicy method*), 183  
**copy\_q\_function\_from()** (*d3rlpy.algos.SAC method*), 45  
**copy\_q\_function\_from()** (*d3rlpy.algos.TD3 method*), 33  
**copy\_q\_function\_from()** (*d3rlpy.algos.TD3PlusBC method*), 149  
**copy\_q\_function\_from()** (*d3rlpy.ope.DiscreteFQE method*), 346  
**copy\_q\_function\_from()** (*d3rlpy.ope.FQE method*), 335  
**CQL (class in *d3rlpy.algos*)**, 90  
**create()** (*d3rlpy.models.encoders.DefaultEncoderFactory method*), 321  
**create()** (*d3rlpy.models.encoders.DenseEncoderFactory method*), 324  
**create()** (*d3rlpy.models.encoders.PixelEncoderFactory method*), 322  
**create()** (*d3rlpy.models.encoders.VectorEncoderFactory method*), 323  
**create()** (*d3rlpy.models.optimizers.AdamFactory method*), 317  
**create()** (*d3rlpy.models.optimizers.OptimizerFactory method*), 315  
**create()** (*d3rlpy.models.optimizers.RMSpropFactory method*), 318  
**create()** (*d3rlpy.models.optimizers.SGDFactory method*), 316  
**create\_continuous()**  
     (*d3rlpy.models.q\_functions.FQFQFunctionFactory method*), 285  
**create\_continuous()**  
     (*d3rlpy.models.q\_functions.IQNQFunctionFactory method*), 284  
**create\_continuous()**  
     (*d3rlpy.models.q\_functions.MeanQFunctionFactory method*), 281  
**create\_continuous()**  
     (*d3rlpy.models.q\_functions.QRQFunctionFactory method*), 282  
**create\_discrete()** (*d3rlpy.models.q\_functions.FQFQFunctionFactory method*), 285  
**create\_discrete()** (*d3rlpy.models.q\_functions.IQNQFunctionFactory method*), 284  
**create\_discrete()** (*d3rlpy.models.q\_functions.MeanQFunctionFactory method*), 281  
**create\_discrete()** (*d3rlpy.models.q\_functions.QRQFunctionFactory method*), 282  
**create\_impl()** (*d3rlpy.algos.AWAC method*), 115  
**create\_impl()** (*d3rlpy.algos.AWR method*), 104  
**create\_impl()** (*d3rlpy.algos.BC method*), 11  
**create\_impl()** (*d3rlpy.algos.BCQ method*), 57  
**create\_impl()** (*d3rlpy.algos.BEAR method*), 69  
**create\_impl()** (*d3rlpy.algos.COMBO method*), 173  
**create\_impl()** (*d3rlpy.algos.CQL method*), 93  
**create\_impl()** (*d3rlpy.algos.CRR method*), 81  
**create\_impl()** (*d3rlpy.algos.DDPG method*), 22  
**create\_impl()** (*d3rlpy.algos.DiscreteAWR method*), 262  
**create\_impl()** (*d3rlpy.algos.DiscreteBC method*), 195  
**create\_impl()** (*d3rlpy.algos.DiscreteBCQ method*), 239  
**create\_impl()** (*d3rlpy.algos.DiscreteCQL method*), 250  
**create\_impl()** (*d3rlpy.algos.DiscreteRandomPolicy method*), 271  
**create\_impl()** (*d3rlpy.algos.DiscreteSAC method*), 228  
**create\_impl()** (*d3rlpy.algos.DoubleDQN method*), 216  
**create\_impl()** (*d3rlpy.algos.DQN method*), 205  
**create\_impl()** (*d3rlpy.algos.MOPO method*), 162  
**create\_impl()** (*d3rlpy.algos.PLAS method*), 127  
**create\_impl()** (*d3rlpy.algos.PLASWithPerturbation method*), 138  
**create\_impl()** (*d3rlpy.algos.RandomPolicy method*), 184  
**create\_impl()** (*d3rlpy.algos.SAC method*), 45  
**create\_impl()** (*d3rlpy.algos.TD3 method*), 34  
**create\_impl()** (*d3rlpy.algos.TD3PlusBC method*), 150  
**create\_impl()** (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics method*), 370  
**create\_impl()** (*d3rlpy.ope.DiscreteFQE method*), 346  
**create\_impl()** (*d3rlpy.ope.FQE method*), 335  
**create\_with\_action()**  
     (*d3rlpy.models.encoders.DefaultEncoderFactory method*), 321  
**create\_with\_action()**  
     (*d3rlpy.models.encoders.DenseEncoderFactory method*), 324  
**create\_with\_action()**  
     (*d3rlpy.models.encoders.PixelEncoderFactory method*), 322  
**create\_with\_action()**

(*d3rlpy.models.encoders.VectorEncoderFactory method*), 323  
 CRR (*class in d3rlpy.algos*), 78

**D**

d3rlpy  
 module, 9  
 d3rlpy.algos  
 module, 9  
 d3rlpy.dataset  
 module, 286  
 d3rlpy.datasets  
 module, 297  
 d3rlpy.dynamics  
 module, 368  
 d3rlpy.metrics  
 module, 325  
 d3rlpy.models.encoders  
 module, 318  
 d3rlpy.models.optimizers  
 module, 314  
 d3rlpy.models.q\_functions  
 module, 280  
 d3rlpy.online  
 module, 361  
 d3rlpy.ope  
 module, 333  
 d3rlpy.preprocessing  
 module, 301  
 DDPG (*class in d3rlpy.algos*), 20  
 DefaultEncoderFactory (*class d3rlpy.models.encoders*), 320  
 DenseEncoderFactory (*class d3rlpy.models.encoders*), 324  
 discounted\_sum\_of\_advantage\_scorer() (*in module d3rlpy.metrics.scorer*), 326  
 discrete\_action\_match\_scorer() (*in module d3rlpy.metrics.scorer*), 329  
 DiscreteAWR (*class in d3rlpy.algos*), 259  
 DiscreteBC (*class in d3rlpy.algos*), 193  
 DiscreteBCQ (*class in d3rlpy.algos*), 237  
 DiscreteCQL (*class in d3rlpy.algos*), 248  
 DiscreteFQE (*class in d3rlpy.ope*), 344  
 DiscreteRandomPolicy (*class in d3rlpy.algos*), 270  
 DiscreteSAC (*class in d3rlpy.algos*), 225  
 DoubleDQN (*class in d3rlpy.algos*), 214  
 DQN (*class in d3rlpy.algos*), 203  
 dump() (*d3rlpy.dataset.MDPDataset method*), 289  
 dynamics\_observation\_prediction\_error\_scorer() (*in module d3rlpy.metrics.scorer*), 332  
 dynamics\_prediction\_variance\_scorer() (*in module d3rlpy.metrics.scorer*), 332  
 dynamics\_reward\_prediction\_error\_scorer() (*in module d3rlpy.metrics.scorer*), 332

**E**

embed\_size (*d3rlpy.models.q\_functions.FQFQFunctionFactory attribute*), 286  
 embed\_size (*d3rlpy.models.q\_functions.IQNQFunctionFactory attribute*), 284  
 entropy\_coeff (*d3rlpy.models.q\_functions.FQFQFunctionFactory attribute*), 286  
 Episode (*class in d3rlpy.dataset*), 291  
 episode\_terminals (*d3rlpy.dataset.MDPDataset attribute*), 290  
 episodes (*d3rlpy.dataset.MDPDataset attribute*), 290  
 evaluate\_on\_environment() (*in module d3rlpy.metrics.scorer*), 330  
 extend() (*d3rlpy.dataset.MDPDataset method*), 289

**F**

fit() (*d3rlpy.algos.AWAC method*), 116  
 fit() (*d3rlpy.algos.AWR method*), 105  
 fit() (*d3rlpy.algos.BC method*), 11  
 fit() (*d3rlpy.algos.BCQ method*), 57  
 fit() (*d3rlpy.algos.BEAR method*), 69  
 fit() (*d3rlpy.algos.COMBO method*), 174  
 fit() (*d3rlpy.algos.CQL method*), 93  
 fit() (*d3rlpy.algos.CRR method*), 81  
 fit() (*d3rlpy.algos.DDPG method*), 22  
 fit() (*d3rlpy.algos.DiscreteAWR method*), 262  
 fit() (*d3rlpy.algos.DiscreteBC method*), 195  
 fit() (*d3rlpy.algos.DiscreteBCQ method*), 239  
 fit() (*d3rlpy.algos.DiscreteCQL method*), 250  
 fit() (*d3rlpy.algos.DiscreteRandomPolicy method*), 271  
 fit() (*d3rlpy.algos.DiscreteSAC method*), 228  
 fit() (*d3rlpy.algos.DoubleDQN method*), 216  
 fit() (*d3rlpy.algos.DQN method*), 205  
 fit() (*d3rlpy.algos.MOPO method*), 162  
 fit() (*d3rlpy.algos.PLAS method*), 127  
 fit() (*d3rlpy.algos.PLASWithPerturbation method*), 138  
 fit() (*d3rlpy.algos.RandomPolicy method*), 184  
 fit() (*d3rlpy.algos.SAC method*), 45  
 fit() (*d3rlpy.algos.TD3 method*), 34  
 fit() (*d3rlpy.algos.TD3PlusBC method*), 150  
 fit() (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics method*), 370  
 fit() (*d3rlpy.ope.DiscreteFQE method*), 347  
 fit() (*d3rlpy.ope.FQE method*), 336  
 fit() (*d3rlpy.preprocessing.ClipRewardScaler method*), 313  
 fit() (*d3rlpy.preprocessing.MinMaxActionScaler method*), 308  
 fit() (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 310  
 fit() (*d3rlpy.preprocessing.MinMaxScaler method*), 304  
 fit() (*d3rlpy.preprocessing.PixelScaler method*), 302

`fit()` (*d3rlpy.preprocessing.StandardRewardScaler method*), 312  
`fit()` (*d3rlpy.preprocessing.StandardScaler method*), 305  
`fit_batch_online()` (*d3rlpy.algos.AWAC method*), 116  
`fit_batch_online()` (*d3rlpy.algos.AWR method*), 105  
`fit_batch_online()` (*d3rlpy.algos.BC method*), 12  
`fit_batch_online()` (*d3rlpy.algos.BCQ method*), 58  
`fit_batch_online()` (*d3rlpy.algos.BEAR method*), 70  
`fit_batch_online()` (*d3rlpy.algos.COMBO method*), 174  
`fit_batch_online()` (*d3rlpy.algos.CQL method*), 94  
`fit_batch_online()` (*d3rlpy.algos.CRR method*), 82  
`fit_batch_online()` (*d3rlpy.algos.DDPG method*), 23  
`fit_batch_online()` (*d3rlpy.algos.DiscreteAWR method*), 263  
`fit_batch_online()` (*d3rlpy.algos.DiscreteBC method*), 196  
`fit_batch_online()` (*d3rlpy.algos.DiscreteBCQ method*), 240  
`fit_batch_online()` (*d3rlpy.algos.DiscreteCQL method*), 251  
`fit_batch_online()` (*d3rlpy.algos.DiscreteRandomPolicy method*), 272  
`fit_batch_online()` (*d3rlpy.algos.DiscreteSAC method*), 229  
`fit_batch_online()` (*d3rlpy.algos.DoubleDQN method*), 217  
`fit_batch_online()` (*d3rlpy.algos.DQN method*), 206  
`fit_batch_online()` (*d3rlpy.algos.MOPO method*), 163  
`fit_batch_online()` (*d3rlpy.algos.PLAS method*), 128  
`fit_batch_online()` (*d3rlpy.algos.PLASWithPerturbation method*), 139  
`fit_batch_online()` (*d3rlpy.algos.RandomPolicy method*), 185  
`fit_batch_online()` (*d3rlpy.algos.SAC method*), 46  
`fit_batch_online()` (*d3rlpy.algos.TD3 method*), 35  
`fit_batch_online()` (*d3rlpy.algos.TD3PlusBC method*), 151  
`fit_batch_online()` (*d3rlpy.ope.DiscreteFQE method*), 347  
`fit_batch_online()` (*d3rlpy.ope.FQE method*), 336  
`fit_online()` (*d3rlpy.algos.AWAC method*), 117  
`fit_online()` (*d3rlpy.algos.AWR method*), 106  
`fit_online()` (*d3rlpy.algos.BC method*), 13  
`fit_online()` (*d3rlpy.algos.BCQ method*), 59  
`fit_online()` (*d3rlpy.algos.BEAR method*), 71  
`fit_online()` (*d3rlpy.algos.COMBO method*), 175  
`fit_online()` (*d3rlpy.algos.CQL method*), 95  
`fit_online()` (*d3rlpy.algos.CRR method*), 83  
`fit_online()` (*d3rlpy.algos.DDPG method*), 24  
`fit_online()` (*d3rlpy.algos.DiscreteAWR method*), 264  
`fit_online()` (*d3rlpy.algos.DiscreteBC method*), 197  
`fit_online()` (*d3rlpy.algos.DiscreteBCQ method*), 242  
`fit_online()` (*d3rlpy.algos.DiscreteCQL method*), 253  
`fit_online()` (*d3rlpy.algos.DiscreteRandomPolicy method*), 274  
`fit_online()` (*d3rlpy.algos.DiscreteSAC method*), 230  
`fit_online()` (*d3rlpy.algos.DoubleDQN method*), 218  
`fit_online()` (*d3rlpy.algos.DQN method*), 207  
`fit_online()` (*d3rlpy.algos.MOPO method*), 164  
`fit_online()` (*d3rlpy.algos.PLAS method*), 129  
`fit_online()` (*d3rlpy.algos.PLASWithPerturbation method*), 140  
`fit_online()` (*d3rlpy.algos.RandomPolicy method*), 186  
`fit_online()` (*d3rlpy.algos.SAC method*), 47  
`fit_online()` (*d3rlpy.algos.TD3 method*), 36  
`fit_online()` (*d3rlpy.algos.TD3PlusBC method*), 152  
`fit_online()` (*d3rlpy.ope.DiscreteFQE method*), 348  
`fit_online()` (*d3rlpy.ope.FQE method*), 337  
`fit_with_env()` (*d3rlpy.preprocessing.ClipRewardScaler method*), 313  
`fit_with_env()` (*d3rlpy.preprocessing.MinMaxActionScaler method*), 308  
`fit_with_env()` (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 310  
`fit_with_env()` (*d3rlpy.preprocessing.MinMaxScaler method*), 304  
`fit_with_env()` (*d3rlpy.preprocessing.PixelScaler method*), 302  
`fit_with_env()` (*d3rlpy.preprocessing.StandardRewardScaler method*), 312  
`fit_with_env()` (*d3rlpy.preprocessing.StandardScaler method*), 305  
`fitter()` (*d3rlpy.algos.AWAC method*), 118  
`fitter()` (*d3rlpy.algos.AWR method*), 107  
`fitter()` (*d3rlpy.algos.BC method*), 14  
`fitter()` (*d3rlpy.algos.BCQ method*), 60  
`fitter()` (*d3rlpy.algos.BEAR method*), 72  
`fitter()` (*d3rlpy.algos.COMBO method*), 176  
`fitter()` (*d3rlpy.algos.CQL method*), 96  
`fitter()` (*d3rlpy.algos.CRR method*), 84  
`fitter()` (*d3rlpy.algos.DDPG method*), 25  
`fitter()` (*d3rlpy.algos.DiscreteAWR method*), 264  
`fitter()` (*d3rlpy.algos.DiscreteBC method*), 197  
`fitter()` (*d3rlpy.algos.DiscreteBCQ method*), 242  
`fitter()` (*d3rlpy.algos.DiscreteCQL method*), 253  
`fitter()` (*d3rlpy.algos.DiscreteRandomPolicy method*), 274  
`fitter()` (*d3rlpy.algos.DiscreteSAC method*), 230  
`fitter()` (*d3rlpy.algos.DoubleDQN method*), 219  
`fitter()` (*d3rlpy.algos.DQN method*), 208  
`fitter()` (*d3rlpy.algos.MOPO method*), 164  
`fitter()` (*d3rlpy.algos.PLAS method*), 130

**fitter()** (*d3rlpy.algos.PLASWithPerturbation method*), 141  
**fitter()** (*d3rlpy.algos.RandomPolicy method*), 186  
**fitter()** (*d3rlpy.algos.SAC method*), 48  
**fitter()** (*d3rlpy.algos.TD3 method*), 36  
**fitter()** (*d3rlpy.algos.TD3PlusBC method*), 152  
**fitter()** (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics method*), 371  
**fitter()** (*d3rlpy.ope.DiscreteFQE method*), 349  
**fitter()** (*d3rlpy.ope.FQE method*), 338  
**FQE** (*class in d3rlpy.ope*), 333  
**FQFQFunctionFactory** (*class in d3rlpy.models.q\_functions*), 285  
**from\_json()** (*d3rlpy.algos.AWAC class method*), 119  
**from\_json()** (*d3rlpy.algos.AWR class method*), 108  
**from\_json()** (*d3rlpy.algos.BC class method*), 15  
**from\_json()** (*d3rlpy.algos.BCQ class method*), 61  
**from\_json()** (*d3rlpy.algos.BEAR class method*), 73  
**from\_json()** (*d3rlpy.algos.COMBO class method*), 177  
**from\_json()** (*d3rlpy.algos.CQL class method*), 97  
**from\_json()** (*d3rlpy.algos.CRR class method*), 85  
**from\_json()** (*d3rlpy.algos.DDPG class method*), 26  
**from\_json()** (*d3rlpy.algos.DiscreteAWR class method*), 265  
**from\_json()** (*d3rlpy.algos.DiscreteBC class method*), 198  
**from\_json()** (*d3rlpy.algos.DiscreteBCQ class method*), 243  
**from\_json()** (*d3rlpy.algos.DiscreteCQL class method*), 254  
**from\_json()** (*d3rlpy.algos.DiscreteRandomPolicy class method*), 275  
**from\_json()** (*d3rlpy.algos.DiscreteSAC class method*), 231  
**from\_json()** (*d3rlpy.algos.DoubleDQN class method*), 220  
**from\_json()** (*d3rlpy.algos.DQN class method*), 209  
**from\_json()** (*d3rlpy.algos.MOPO class method*), 165  
**from\_json()** (*d3rlpy.algos.PLAS class method*), 131  
**from\_json()** (*d3rlpy.algos.PLASWithPerturbation class method*), 142  
**from\_json()** (*d3rlpy.algos.RandomPolicy class method*), 187  
**from\_json()** (*d3rlpy.algos.SAC class method*), 49  
**from\_json()** (*d3rlpy.algos.TD3 class method*), 37  
**from\_json()** (*d3rlpy.algos.TD3PlusBC class method*), 153  
**from\_json()** (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics class method*), 372  
**from\_json()** (*d3rlpy.ope.DiscreteFQE class method*), 350  
**from\_json()** (*d3rlpy.ope.FQE class method*), 339

**G**

**gamma** (*d3rlpy.algos.AWAC attribute*), 123  
**gamma** (*d3rlpy.algos.AWR attribute*), 112  
**gamma** (*d3rlpy.algos.BC attribute*), 18  
**gamma** (*d3rlpy.algos.BCQ attribute*), 65  
**gamma** (*d3rlpy.algos.BEAR attribute*), 77  
**gamma** (*d3rlpy.algos.COMBO attribute*), 181  
**gamma** (*d3rlpy.algos.CQL attribute*), 101  
**gamma** (*d3rlpy.algos.CRR attribute*), 89  
**gamma** (*d3rlpy.algos.DDPG attribute*), 30  
**gamma** (*d3rlpy.algos.DiscreteAWR attribute*), 269  
**gamma** (*d3rlpy.algos.DiscreteBC attribute*), 202  
**gamma** (*d3rlpy.algos.DiscreteBCQ attribute*), 247  
**gamma** (*d3rlpy.algos.DiscreteCQL attribute*), 258  
**gamma** (*d3rlpy.algos.DiscreteRandomPolicy attribute*), 279  
**gamma** (*d3rlpy.algos.DiscreteSAC attribute*), 235  
**gamma** (*d3rlpy.algos.DoubleDQN attribute*), 224  
**gamma** (*d3rlpy.algos.DQN attribute*), 213  
**gamma** (*d3rlpy.algos.MOPO attribute*), 169  
**gamma** (*d3rlpy.algos.PLAS attribute*), 135  
**gamma** (*d3rlpy.algos.PLASWithPerturbation attribute*), 146  
**gamma** (*d3rlpy.algos.RandomPolicy attribute*), 191  
**gamma** (*d3rlpy.algos.SAC attribute*), 53  
**gamma** (*d3rlpy.algos.TD3 attribute*), 41  
**gamma** (*d3rlpy.algos.TD3PlusBC attribute*), 157  
**gamma** (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics attribute*), 375  
**gamma** (*d3rlpy.ope.DiscreteFQE attribute*), 354  
**gamma** (*d3rlpy.ope.FQE attribute*), 343  
**generate\_new\_data()** (*d3rlpy.algos.AWAC method*), 120  
**generate\_new\_data()** (*d3rlpy.algos.AWR method*), 109  
**generate\_new\_data()** (*d3rlpy.algos.BC method*), 15  
**generate\_new\_data()** (*d3rlpy.algos.BCQ method*), 61  
**generate\_new\_data()** (*d3rlpy.algos.BEAR method*), 73  
**generate\_new\_data()** (*d3rlpy.algos.COMBO method*), 178  
**generate\_new\_data()** (*d3rlpy.algos.CQL method*), 97  
**generate\_new\_data()** (*d3rlpy.algos.CRR method*), 85  
**generate\_new\_data()** (*d3rlpy.algos.DDPG method*), 26  
**generate\_new\_data()** (*d3rlpy.algos.DiscreteAWR method*), 266  
**generate\_new\_data()** (*d3rlpy.algos.DiscreteBC method*), 199  
**generate\_new\_data()** (*d3rlpy.algos.DiscreteBCQ method*), 243  
**generate\_new\_data()** (*d3rlpy.algos.DiscreteCQL method*), 254

generate_new_data()	( <i>d3rlpy.algos.DiscreteRandomPolicy</i> method), 275	get_action_type()	( <i>d3rlpy.algos.DiscreteSAC</i> method), 232
generate_new_data()	( <i>d3rlpy.algos.DiscreteSAC</i> method), 232	get_action_type()	( <i>d3rlpy.algos.DoubleDQN</i> method), 220
generate_new_data()	( <i>d3rlpy.algos.DoubleDQN</i> method), 220	get_action_type()	( <i>d3rlpy.algos.DQN</i> method), 209
generate_new_data()	( <i>d3rlpy.algos.DQN</i> method), 209	get_action_type()	( <i>d3rlpy.algos.MOPO</i> method), 166
generate_new_data()	( <i>d3rlpy.algos.MOPO</i> method), 166	get_action_type()	( <i>d3rlpy.algos.PLAS</i> method), 131
generate_new_data()	( <i>d3rlpy.algos.PLAS</i> method), 131	get_action_type()	( <i>d3rlpy.algos.PLASWithPerturbation</i> method), 143
generate_new_data()	( <i>d3rlpy.algos.PLASWithPerturbation</i> method), 142	get_action_type()	( <i>d3rlpy.algos.RandomPolicy</i> method), 188
generate_new_data()	( <i>d3rlpy.algos.RandomPolicy</i> method), 188	get_action_type()	( <i>d3rlpy.algos.SAC</i> method), 50
generate_new_data()	( <i>d3rlpy.algos.SAC</i> method), 49	get_action_type()	( <i>d3rlpy.algos.TD3</i> method), 38
generate_new_data()	( <i>d3rlpy.algos.TD3</i> method), 38	get_action_type()	( <i>d3rlpy.algos.TD3PlusBC</i> method), 154
generate_new_data()	( <i>d3rlpy.algos.TD3PlusBC</i> method), 154	get_action_type()	( <i>d3rlpy.dynamics.ProbabilisticEnsembleDynamics</i> method), 373
generate_new_data()	( <i>d3rlpy.dynamics.ProbabilisticEnsembleDynamic</i> method), 372	get_action_type()	( <i>d3rlpy.ope.DiscreteFQE</i> method), 351
generate_new_data()	( <i>d3rlpy.ope.DiscreteFQE</i> method), 351	get_action_type()	( <i>d3rlpy.ope.FQE</i> method), 340
generate_new_data()	( <i>d3rlpy.ope.FQE</i> method), 340	get_additional_data()	( <i>d3rlpy.dataset.TransitionMiniBatch</i> method), 296
get_action_size()	( <i>d3rlpy.dataset.Episode</i> method), 292	get_atari()	(in module <i>d3rlpy.datasets</i> ), 299
get_action_size()	( <i>d3rlpy.dataset.MDPDataset</i> method), 289	get_cartpole()	(in module <i>d3rlpy.datasets</i> ), 297
get_action_size()	( <i>d3rlpy.dataset.Transition</i> method), 293	get_d4rl()	(in module <i>d3rlpy.datasets</i> ), 299
get_action_type()	( <i>d3rlpy.algos.AWAC</i> method), 120	get_dataset()	(in module <i>d3rlpy.datasets</i> ), 300
get_action_type()	( <i>d3rlpy.algos.AWR</i> method), 109	get_observation_shape()	( <i>d3rlpy.dataset.Episode</i> method), 292
get_action_type()	( <i>d3rlpy.algos.BC</i> method), 16	get_observation_shape()	( <i>d3rlpy.dataset.MDPDataset</i> method), 289
get_action_type()	( <i>d3rlpy.algos.BCQ</i> method), 62	get_observation_shape()	( <i>d3rlpy.dataset.Transition</i> method), 293
get_action_type()	( <i>d3rlpy.algos.BEAR</i> method), 74	get_params()	( <i>d3rlpy.algos.AWAC</i> method), 120
get_action_type()	( <i>d3rlpy.algos.COMBO</i> method), 178	get_params()	( <i>d3rlpy.algos.AWR</i> method), 109
get_action_type()	( <i>d3rlpy.algos.CQL</i> method), 97	get_params()	( <i>d3rlpy.algos.BC</i> method), 16
get_action_type()	( <i>d3rlpy.algos.CRR</i> method), 85	get_params()	( <i>d3rlpy.algos.BCQ</i> method), 62
get_action_type()	( <i>d3rlpy.algos.DDPG</i> method), 26	get_params()	( <i>d3rlpy.algos.BEAR</i> method), 74
get_action_type()	( <i>d3rlpy.algos.DiscreteAWR</i> method), 266	get_params()	( <i>d3rlpy.algos.COMBO</i> method), 178
get_action_type()	( <i>d3rlpy.algos.DiscreteBC</i> method), 199	get_params()	( <i>d3rlpy.algos.CQL</i> method), 97
get_action_type()	( <i>d3rlpy.algos.DiscreteBCQ</i> method), 243	get_params()	( <i>d3rlpy.algos.CRR</i> method), 85
get_action_type()	( <i>d3rlpy.algos.DiscreteCQL</i> method), 254	get_params()	( <i>d3rlpy.algos.DDPG</i> method), 26
get_action_type()	( <i>d3rlpy.algos.DiscreteRandomPolicy</i> method), 276	get_params()	( <i>d3rlpy.algos.DiscreteAWR</i> method), 266
		get_params()	( <i>d3rlpy.algos.DiscreteBC</i> method), 199
		get_params()	( <i>d3rlpy.algos.DiscreteBCQ</i> method), 243
		get_params()	( <i>d3rlpy.algos.DiscreteCQL</i> method), 255
		get_params()	( <i>d3rlpy.algos.DiscreteRandomPolicy</i> method), 276
		get_params()	( <i>d3rlpy.algos.DiscreteSAC</i> method), 232
		get_params()	( <i>d3rlpy.algos.DoubleDQN</i> method), 220
		get_params()	( <i>d3rlpy.algos.DQN</i> method), 209
		get_params()	( <i>d3rlpy.algos.MOPO</i> method), 166
		get_params()	( <i>d3rlpy.algos.PLAS</i> method), 131

get\_params() (*d3rlpy.algos.PLASWithPerturbation method*), 143  
 get\_params() (*d3rlpy.algos.RandomPolicy method*), 188  
 get\_params() (*d3rlpy.algos.SAC method*), 50  
 get\_params() (*d3rlpy.algos.TD3 method*), 38  
 get\_params() (*d3rlpy.algos.TD3PlusBC method*), 154  
 get\_params() (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics*), 373  
 get\_params() (*d3rlpy.models.encoders.DefaultEncoderFactory*), 321  
 get\_params() (*d3rlpy.models.encoders.DenseEncoderFactory*), 324  
 get\_params() (*d3rlpy.models.encoders.PixelEncoderFactory*), 322  
 get\_type() (*d3rlpy.models.encoders.VectorEncoderFactory*), 323  
 get\_type() (*d3rlpy.models.q\_functions.FQFQFunctionFactory*), 285  
 get\_type() (*d3rlpy.models.q\_functions.IQNQFunctionFactory*), 284  
 get\_type() (*d3rlpy.models.q\_functions.MeanQFunctionFactory*), 281  
 get\_type() (*d3rlpy.models.q\_functions.QRQFunctionFactory*), 283  
 get\_type() (*d3rlpy.preprocessing.ClipRewardScaler*), 313  
 get\_type() (*d3rlpy.preprocessing.MinMaxActionScaler*), 308  
 get\_type() (*d3rlpy.preprocessing.MinMaxRewardScaler*), 310  
 get\_type() (*d3rlpy.preprocessing.OptimizerFactory*), 304  
 get\_type() (*d3rlpy.preprocessing.Pixelscaler*), 302  
 get\_type() (*d3rlpy.preprocessing.RMSpropFactory*), 302  
 get\_type() (*d3rlpy.preprocessing.StandardRewardScaler*), 312  
 get\_type() (*d3rlpy.preprocessing.StandardScaler*), 306  
 get\_type() (*d3rlpy.preprocessing.StandardScaler*), 306  
 grad\_step (*d3rlpy.algos.AWAC attribute*), 123  
 grad\_step (*d3rlpy.algos.AWR attribute*), 112  
 grad\_step (*d3rlpy.algos.BC attribute*), 19  
 grad\_step (*d3rlpy.algos.BCQ attribute*), 65  
 grad\_step (*d3rlpy.algos.BEAR attribute*), 77  
 grad\_step (*d3rlpy.algos.COMBO attribute*), 181  
 grad\_step (*d3rlpy.algos.CQL attribute*), 101  
 grad\_step (*d3rlpy.algos.CRR attribute*), 89  
 grad\_step (*d3rlpy.algos.DDPG attribute*), 30  
 grad\_step (*d3rlpy.algos.DiscreteAWR attribute*), 269  
 grad\_step (*d3rlpy.algos.DiscreteBC attribute*), 202  
 grad\_step (*d3rlpy.algos.DiscreteBCQ attribute*), 247  
 grad\_step (*d3rlpy.algos.DiscreteCQL attribute*), 258  
 grad\_step (*d3rlpy.algos.DiscreteRandomPolicy attribute*), 279  
 grad\_step (*d3rlpy.algos.DiscreteSAC attribute*), 236  
 grad\_step (*d3rlpy.algos.DoubleDQN attribute*), 224  
 grad\_step (*d3rlpy.algos.DQN attribute*), 213  
 grad\_step (*d3rlpy.algos.MOPO attribute*), 170  
 grad\_step (*d3rlpy.algos.PLAS attribute*), 135  
 grad\_step (*d3rlpy.algos.PLASWithPerturbation attribute*), 146  
 grad\_step (*d3rlpy.algos.RandomPolicy attribute*), 192  
 grad\_step (*d3rlpy.algos.SAC attribute*), 53  
 grad\_step (*d3rlpy.models.encoders.DefaultEncoderFactory*), 42  
 grad\_step (*d3rlpy.algos.TD3 attribute*), 42  
 grad\_step (*d3rlpy.algos.TD3PlusBC attribute*), 158  
 grad\_step (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics*), 373

*attribute), 375*

`grad_step (d3rlpy.ope.DiscreteFQE attribute)`, 354

`grad_step (d3rlpy.ope.FQE attribute)`, 343

|

`impl (d3rlpy.algos.AWAC attribute)`, 123

`impl (d3rlpy.algos.AWR attribute)`, 112

`impl (d3rlpy.algos.BC attribute)`, 19

`impl (d3rlpy.algos.BCQ attribute)`, 65

`impl (d3rlpy.algos.BEAR attribute)`, 77

`impl (d3rlpy.algos.COMBO attribute)`, 181

`impl (d3rlpy.algos.CQL attribute)`, 101

`impl (d3rlpy.algos.CRR attribute)`, 89

`impl (d3rlpy.algos.DDPG attribute)`, 30

`impl (d3rlpy.algos.DiscreteAWR attribute)`, 269

`impl (d3rlpy.algos.DiscreteBC attribute)`, 202

`impl (d3rlpy.algos.DiscreteBCQ attribute)`, 247

`impl (d3rlpy.algos.DiscreteCQL attribute)`, 258

`impl (d3rlpy.algos.DiscreteRandomPolicy attribute)`, 279

`impl (d3rlpy.algos.DiscreteSAC attribute)`, 236

`impl (d3rlpy.algos.DoubleDQN attribute)`, 224

`impl (d3rlpy.algos.DQN attribute)`, 213

`impl (d3rlpy.algos.MOPO attribute)`, 170

`impl (d3rlpy.algos.PLAS attribute)`, 135

`impl (d3rlpy.algos.PLASWithPerturbation attribute)`, 146

`impl (d3rlpy.algos.RandomPolicy attribute)`, 192

`impl (d3rlpy.algos.SAC attribute)`, 53

`impl (d3rlpy.algos.TD3 attribute)`, 42

`impl (d3rlpy.algos.TD3PlusBC attribute)`, 158

`impl (d3rlpy.dynamics.ProbabilisticEnsembleDynamics attribute)`, 375

`impl (d3rlpy.ope.DiscreteFQE attribute)`, 354

`impl (d3rlpy.ope.FQE attribute)`, 343

`initial_state_value_estimation_scorer() (in module d3rlpy.metrics.scorer)`, 328

`IQNFunctionFactory (class in d3rlpy.models.q_functions)`, 283

`is_action_discrete() (d3rlpy.dataset.MDPDataset method)`, 289

L

`LinearDecayEpsilonGreedy (class in d3rlpy.online.explorers)`, 364

`load() (d3rlpy.dataset.MDPDataset class method)`, 290

`load_model() (d3rlpy.algos.AWAC method)`, 120

`load_model() (d3rlpy.algos.AWR method)`, 109

`load_model() (d3rlpy.algos.BC method)`, 16

`load_model() (d3rlpy.algos.BCQ method)`, 62

`load_model() (d3rlpy.algos.BEAR method)`, 74

`load_model() (d3rlpy.algos.COMBO method)`, 178

`load_model() (d3rlpy.algos.CQL method)`, 98

`load_model() (d3rlpy.algos.CRR method)`, 86

`load_model() (d3rlpy.algos.DDPG method)`, 27

`load_model() (d3rlpy.algos.DiscreteAWR method)`, 266

`load_model() (d3rlpy.algos.DiscreteBC method)`, 199

`load_model() (d3rlpy.algos.DiscreteBCQ method)`, 244

`load_model() (d3rlpy.algos.DiscreteCQL method)`, 255

`load_model() (d3rlpy.algos.DiscreteRandomPolicy method)`, 276

`load_model() (d3rlpy.algos.DiscreteSAC method)`, 232

`load_model() (d3rlpy.algos.DoubleDQN method)`, 221

`load_model() (d3rlpy.algos.DQN method)`, 210

`load_model() (d3rlpy.algos.MOPO method)`, 166

`load_model() (d3rlpy.algos.PLAS method)`, 132

`load_model() (d3rlpy.algos.PLASWithPerturbation method)`, 143

`load_model() (d3rlpy.algos.RandomPolicy method)`, 188

`load_model() (d3rlpy.algos.SAC method)`, 50

`load_model() (d3rlpy.algos.TD3 method)`, 38

`load_model() (d3rlpy.algos.TD3PlusBC method)`, 154

`load_model() (d3rlpy.dynamics.ProbabilisticEnsembleDynamics method)`, 373

`load_model() (d3rlpy.ope.DiscreteFQE method)`, 351

`load_model() (d3rlpy.ope.FQE method)`, 340

M

`mask (d3rlpy.dataset.Transition attribute)`, 294

`masks (d3rlpy.dataset.TransitionMiniBatch attribute)`, 296

`MDPDataset (class in d3rlpy.dataset)`, 287

`MeanQFunctionFactory (class in d3rlpy.models.q_functions)`, 281

`MinMaxActionScaler (class in d3rlpy.preprocessing)`, 307

`MinMaxRewardScaler (class in d3rlpy.preprocessing)`, 309

`MinMaxScaler (class in d3rlpy.preprocessing)`, 303

`module`

- d3rlpy, 9
- d3rlpy.algos, 9
- d3rlpy.dataset, 286
- d3rlpy.datasets, 297
- d3rlpy.dynamics, 368
- d3rlpy.metrics, 325
- d3rlpy.models.encoders, 318
- d3rlpy.models.optimizers, 314
- d3rlpy.models.q\_functions, 280
- d3rlpy.online, 361
- d3rlpy.ope, 333
- d3rlpy.preprocessing, 301

`MOPO (class in d3rlpy.algos)`, 159

N

`n_frames (d3rlpy.algos.AWAC attribute)`, 124

`n_frames (d3rlpy.algos.AWR attribute)`, 112

`n_frames (d3rlpy.algos.BC attribute)`, 19

`n_frames (d3rlpy.algos.BCQ attribute)`, 65

n\_frames (*d3rlpy.algos.BEAR attribute*), 77  
n\_frames (*d3rlpy.algos.COMBO attribute*), 182  
n\_frames (*d3rlpy.algos.CQL attribute*), 101  
n\_frames (*d3rlpy.algos.CRR attribute*), 89  
n\_frames (*d3rlpy.algos.DDPG attribute*), 30  
n\_frames (*d3rlpy.algos.DiscreteAWR attribute*), 269  
n\_frames (*d3rlpy.algos.DiscreteBC attribute*), 202  
n\_frames (*d3rlpy.algos.DiscreteBCQ attribute*), 247  
n\_frames (*d3rlpy.algos.DiscreteCQL attribute*), 258  
n\_frames (*d3rlpy.algos.DiscreteRandomPolicy attribute*), 279  
n\_frames (*d3rlpy.algos.DiscreteSAC attribute*), 236  
n\_frames (*d3rlpy.algos.DoubleDQN attribute*), 224  
n\_frames (*d3rlpy.algos.DQN attribute*), 213  
n\_frames (*d3rlpy.algos.MOPO attribute*), 170  
n\_frames (*d3rlpy.algos.PLAS attribute*), 135  
n\_frames (*d3rlpy.algos.PLASWithPerturbation attribute*), 146  
n\_frames (*d3rlpy.algos.RandomPolicy attribute*), 192  
n\_frames (*d3rlpy.algos.SAC attribute*), 53  
n\_frames (*d3rlpy.algos.TD3 attribute*), 42  
n\_frames (*d3rlpy.algos.TD3PlusBC attribute*), 158  
n\_frames (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics attribute*), 375  
n\_frames (*d3rlpy.ope.DiscreteFQE attribute*), 355  
n\_frames (*d3rlpy.ope.FQE attribute*), 344  
n\_greedy\_quantiles (*d3rlpy.models.q\_functions.IQNFunctionFactory attribute*), 284  
n\_quantiles (*d3rlpy.models.q\_functions.FQFQFunctionFactory attribute*), 286  
n\_quantiles (*d3rlpy.models.q\_functions.IQNFunctionFactory attribute*), 284  
n\_quantiles (*d3rlpy.models.q\_functions.QRQFunctionFactory attribute*), 283  
n\_steps (*d3rlpy.algos.AWAC attribute*), 124  
n\_steps (*d3rlpy.algos.AWR attribute*), 112  
n\_steps (*d3rlpy.algos.BC attribute*), 19  
n\_steps (*d3rlpy.algos.BCQ attribute*), 65  
n\_steps (*d3rlpy.algos.BEAR attribute*), 77  
n\_steps (*d3rlpy.algos.COMBO attribute*), 182  
n\_steps (*d3rlpy.algos.CQL attribute*), 101  
n\_steps (*d3rlpy.algos.CRR attribute*), 89  
n\_steps (*d3rlpy.algos.DDPG attribute*), 30  
n\_steps (*d3rlpy.algos.DiscreteAWR attribute*), 269  
n\_steps (*d3rlpy.algos.DiscreteBC attribute*), 202  
n\_steps (*d3rlpy.algos.DiscreteBCQ attribute*), 247  
n\_steps (*d3rlpy.algos.DiscreteCQL attribute*), 258  
n\_steps (*d3rlpy.algos.DiscreteRandomPolicy attribute*), 279  
n\_steps (*d3rlpy.algos.DiscreteSAC attribute*), 236  
n\_steps (*d3rlpy.algos.DoubleDQN attribute*), 224  
n\_steps (*d3rlpy.algos.DQN attribute*), 213  
n\_steps (*d3rlpy.algos.MOPO attribute*), 170  
n\_steps (*d3rlpy.algos.PLAS attribute*), 135  
n\_steps (*d3rlpy.algos.PLASWithPerturbation attribute*), 146  
n\_steps (*d3rlpy.algos.RandomPolicy attribute*), 192  
n\_steps (*d3rlpy.algos.SAC attribute*), 53  
n\_steps (*d3rlpy.algos.TD3 attribute*), 42  
n\_steps (*d3rlpy.algos.TD3PlusBC attribute*), 158  
n\_steps (*d3rlpy.dataset.TransitionMiniBatch attribute*), 296  
n\_steps (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics attribute*), 375  
n\_steps (*d3rlpy.ope.DiscreteFQE attribute*), 355  
n\_steps (*d3rlpy.ope.FQE attribute*), 344  
next\_action (*d3rlpy.dataset.Transition attribute*), 294  
next\_actions (*d3rlpy.dataset.TransitionMiniBatch attribute*), 296  
next\_observation (*d3rlpy.dataset.Transition attribute*), 294  
next\_observations (*d3rlpy.dataset.TransitionMiniBatch attribute*), 296  
next\_reward (*d3rlpy.dataset.Transition attribute*), 294  
next\_rewards (*d3rlpy.dataset.TransitionMiniBatch attribute*), 296  
next\_transition (*d3rlpy.dataset.Transition attribute*), 294  
NormalNoise (*class in d3rlpy.online.explorers*), 365

## O

**ObservationFactory**  
observation (*d3rlpy.dataset.Transition attribute*), 294  
observation\_shape (*d3rlpy.algos.AWAC attribute*), 124  
observation\_shape (*d3rlpy.algos.AWR attribute*), 112  
observation\_shape (*d3rlpy.algos.BC attribute*), 19  
observation\_shape (*d3rlpy.algos.BCQ attribute*), 66  
observation\_shape (*d3rlpy.algos.BEAR attribute*), 78  
observation\_shape (*d3rlpy.algos.COMBO attribute*), 182  
observation\_shape (*d3rlpy.algos.CQL attribute*), 101  
observation\_shape (*d3rlpy.algos.CRR attribute*), 89  
observation\_shape (*d3rlpy.algos.DDPG attribute*), 30  
observation\_shape (*d3rlpy.algos.DiscreteAWR attribute*), 270  
observation\_shape (*d3rlpy.algos.DiscreteBC attribute*), 203  
observation\_shape (*d3rlpy.algos.DiscreteBCQ attribute*), 247  
observation\_shape (*d3rlpy.algos.DiscreteCQL attribute*), 258  
observation\_shape (*d3rlpy.algos.DiscreteRandomPolicy attribute*), 280  
observation\_shape (*d3rlpy.algos.DiscreteSAC attribute*), 236  
observation\_shape (*d3rlpy.algos.DoubleDQN attribute*), 224  
observation\_shape (*d3rlpy.algos.DQN attribute*), 213

<code>observation_shape</code> ( <i>d3rlpy.algos.MOPO</i> attribute), 170	<code>predict()</code> ( <i>d3rlpy.algos.TD3</i> method), 39
<code>observation_shape</code> ( <i>d3rlpy.algos.PLAS</i> attribute), 135	<code>predict()</code> ( <i>d3rlpy.algos.TD3PlusBC</i> method), 155
<code>observation_shape</code> ( <i>d3rlpy.algos.PLASWithPerturbation</i> attribute), 147	<code>predict()</code> ( <i>d3rlpy.dynamics.ProbabilisticEnsembleDynamics</i> method), 373
<code>observation_shape</code> ( <i>d3rlpy.algos.RandomPolicy</i> attribute), 192	<code>predict()</code> ( <i>d3rlpy.ope.DiscreteFQE</i> method), 351
<code>observation_shape</code> ( <i>d3rlpy.algos.SAC</i> attribute), 54	<code>predict()</code> ( <i>d3rlpy.ope.FQE</i> method), 340
<code>observation_shape</code> ( <i>d3rlpy.algos.TD3</i> attribute), 42	<code>predict_value()</code> ( <i>d3rlpy.algos.AWAC</i> method), 121
<code>observation_shape</code> ( <i>d3rlpy.algos.TD3PlusBC</i> attribute), 158	<code>predict_value()</code> ( <i>d3rlpy.algos.AWR</i> method), 110
<code>observation_shape</code> ( <i>d3rlpy.dynamics.ProbabilisticEnsemble</i> attribute), 375	<code>predict_value()</code> ( <i>d3rlpy.algos.BC</i> method), 16
<code>observation_shape</code> ( <i>d3rlpy.ope.DiscreteFQE</i> attribute), 355	<code>predict_value()</code> ( <i>d3rlpy.algos.BCQ</i> method), 62
<code>observation_shape</code> ( <i>d3rlpy.ope.FQE</i> attribute), 344	<code>predict_value()</code> ( <i>d3rlpy.algos.BEAR</i> method), 74
<code>observations</code> ( <i>d3rlpy.dataset.Episode</i> attribute), 292	<code>predict_value()</code> ( <i>d3rlpy.algos.COMBO</i> method), 179
<code>observations</code> ( <i>d3rlpy.dataset.MDPDataset</i> attribute), 290	<code>predict_value()</code> ( <i>d3rlpy.algos.CQL</i> method), 98
<code>observations</code> ( <i>d3rlpy.dataset.TransitionMiniBatch</i> attribute), 297	<code>predict_value()</code> ( <i>d3rlpy.algos.CRR</i> method), 86
<code>OptimizerFactory</code> (class in <i>d3rlpy.models.optimizers</i> ), 315	<code>predict_value()</code> ( <i>d3rlpy.algos.DDPG</i> method), 27
<b>P</b>	
<code>PixelEncoderFactory</code> (class in <i>d3rlpy.models.encoders</i> ), 321	<code>predict_value()</code> ( <i>d3rlpy.algos.DiscreteAWR</i> method), 267
<code>PixelScaler</code> (class in <i>d3rlpy.preprocessing</i> ), 301	<code>predict_value()</code> ( <i>d3rlpy.algos.DiscreteBC</i> method), 200
<code>PLAS</code> (class in <i>d3rlpy.algos</i> ), 124	<code>predict_value()</code> ( <i>d3rlpy.algos.DiscreteBCQ</i> method), 244
<code>PLASWithPerturbation</code> (class in <i>d3rlpy.algos</i> ), 136	<code>predict_value()</code> ( <i>d3rlpy.algos.DiscreteCQL</i> method), 255
<code>predict()</code> ( <i>d3rlpy.algos.AWAC</i> method), 120	<code>predict_value()</code> ( <i>d3rlpy.algos.DiscreteRandomPolicy</i> method), 276
<code>predict()</code> ( <i>d3rlpy.algos.AWR</i> method), 109	<code>in</code> <code>predict_value()</code> ( <i>d3rlpy.algos.DiscreteSAC</i> method), 233
<code>predict()</code> ( <i>d3rlpy.algos.BC</i> method), 16	<code>predict_value()</code> ( <i>d3rlpy.algos.DoubleDQN</i> method), 221
<code>predict()</code> ( <i>d3rlpy.algos.BCQ</i> method), 62	<code>predict_value()</code> ( <i>d3rlpy.algos.DQN</i> method), 210
<code>predict()</code> ( <i>d3rlpy.algos.BEAR</i> method), 74	<code>predict_value()</code> ( <i>d3rlpy.algos.MOPO</i> method), 167
<code>predict()</code> ( <i>d3rlpy.algos.COMBO</i> method), 178	<code>predict_value()</code> ( <i>d3rlpy.algos.PLAS</i> method), 132
<code>predict()</code> ( <i>d3rlpy.algos.CQL</i> method), 98	<code>predict_value()</code> ( <i>d3rlpy.algos.PLASWithPerturbation</i> method), 143
<code>predict()</code> ( <i>d3rlpy.algos.CRR</i> method), 86	<code>predict_value()</code> ( <i>d3rlpy.algos.RandomPolicy</i> method), 189
<code>predict()</code> ( <i>d3rlpy.algos.DDPG</i> method), 27	<code>predict_value()</code> ( <i>d3rlpy.algos.SAC</i> method), 50
<code>predict()</code> ( <i>d3rlpy.algos.DiscreteAWR</i> method), 267	<code>predict_value()</code> ( <i>d3rlpy.algos.TD3</i> method), 39
<code>predict()</code> ( <i>d3rlpy.algos.DiscreteBC</i> method), 200	<code>predict_value()</code> ( <i>d3rlpy.algos.TD3PlusBC</i> method), 155
<code>predict()</code> ( <i>d3rlpy.algos.DiscreteBCQ</i> method), 244	<code>predict_value()</code> ( <i>d3rlpy.ope.DiscreteFQE</i> method), 352
<code>predict()</code> ( <i>d3rlpy.algos.DiscreteCQL</i> method), 255	<code>predict_value()</code> ( <i>d3rlpy.ope.FQE</i> method), 341
<code>predict()</code> ( <i>d3rlpy.algos.DiscreteRandomPolicy</i> method), 276	<code>prev_transition</code> ( <i>d3rlpy.dataset.Transition</i> attribute), 294
<code>predict()</code> ( <i>d3rlpy.algos.DiscreteSAC</i> method), 233	<code>ProbabilisticEnsembleDynamics</code> (class in <i>d3rlpy.dynamics</i> ), 369
<code>predict()</code> ( <i>d3rlpy.algos.DoubleDQN</i> method), 221	
<code>predict()</code> ( <i>d3rlpy.algos.DQN</i> method), 210	
<code>predict()</code> ( <i>d3rlpy.algos.MOPO</i> method), 167	
<code>predict()</code> ( <i>d3rlpy.algos.PLAS</i> method), 132	
<code>predict()</code> ( <i>d3rlpy.algos.PLASWithPerturbation</i> method), 143	
<code>predict()</code> ( <i>d3rlpy.algos.RandomPolicy</i> method), 189	
<code>predict()</code> ( <i>d3rlpy.algos.SAC</i> method), 50	
<b>Q</b>	
<b>R</b>	
<code>RandomPolicy</code> (class in <i>d3rlpy.algos</i> ), 182	

`ReplayBuffer` (*class in d3rlpy.online.buffers*), 362  
`reverse_transform()`  
     (*d3rlpy.preprocessing.ClipRewardScaler method*), 314  
`reverse_transform()`  
     (*d3rlpy.preprocessing.MinMaxActionScaler method*), 308  
`reverse_transform()`  
     (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 310  
`reverse_transform()`  
     (*d3rlpy.preprocessing.MinMaxScaler method*), 304  
`reverse_transform()`  
     (*d3rlpy.preprocessing.PixelScaler method*), 302  
`reverse_transform()`  
     (*d3rlpy.preprocessing.StandardRewardScaler method*), 312  
`reverse_transform()`  
     (*d3rlpy.preprocessing.StandardScaler method*), 306  
`reverse_transform_numpy()`  
     (*d3rlpy.preprocessing.MinMaxActionScaler method*), 308  
`reward` (*d3rlpy.dataset.Transition attribute*), 295  
`reward_scaler` (*d3rlpy.algos.AWAC attribute*), 124  
`reward_scaler` (*d3rlpy.algos.AWR attribute*), 113  
`reward_scaler` (*d3rlpy.algos.BC attribute*), 19  
`reward_scaler` (*d3rlpy.algos.BCQ attribute*), 66  
`reward_scaler` (*d3rlpy.algos.BEAR attribute*), 78  
`reward_scaler` (*d3rlpy.algos.COMBO attribute*), 182  
`reward_scaler` (*d3rlpy.algos.CQL attribute*), 102  
`reward_scaler` (*d3rlpy.algos.CRR attribute*), 90  
`reward_scaler` (*d3rlpy.algos.DDPG attribute*), 31  
`reward_scaler` (*d3rlpy.algos.DiscreteAWR attribute*), 270  
`reward_scaler` (*d3rlpy.algos.DiscreteBC attribute*), 203  
`reward_scaler` (*d3rlpy.algos.DiscreteBCQ attribute*), 248  
`reward_scaler` (*d3rlpy.algos.DiscreteCQL attribute*), 259  
`reward_scaler` (*d3rlpy.algos.DiscreteRandomPolicy attribute*), 280  
`reward_scaler` (*d3rlpy.algos.DiscreteSAC attribute*), 236  
`reward_scaler` (*d3rlpy.algos.DoubleDQN attribute*), 225  
`reward_scaler` (*d3rlpy.algos.DQN attribute*), 214  
`reward_scaler` (*d3rlpy.algos.MOPO attribute*), 170  
`reward_scaler` (*d3rlpy.algos.PLAS attribute*), 135  
`reward_scaler` (*d3rlpy.algos.PLASWithPerturbation attribute*), 147  
`reward_scaler` (*d3rlpy.algos.RandomPolicy attribute*), 192  
`reward_scaler` (*d3rlpy.algos.SAC attribute*), 54  
`reward_scaler` (*d3rlpy.algos.TD3 attribute*), 42  
`reward_scaler` (*d3rlpy.algos.TD3PlusBC attribute*), 158  
`reward_scaler` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics attribute*), 376  
`reward_scaler` (*d3rlpy.ope.DiscreteFQE attribute*), 355  
`reward_scaler` (*d3rlpy.ope.FQE attribute*), 344  
`rewards` (*d3rlpy.dataset.Episode attribute*), 292  
`rewards` (*d3rlpy.dataset.MDPDataset attribute*), 291  
`rewards` (*d3rlpy.dataset.TransitionMiniBatch attribute*), 297  
`RMSpropFactory` (*class in d3rlpy.models.optimizers*), 317

## S

`SAC` (*class in d3rlpy.algos*), 43  
`sample()`    (*d3rlpy.online.buffers.BatchReplayBuffer method*), 367  
`sample()` (*d3rlpy.online.buffers.ReplayBuffer method*), 362  
`sample()` (*d3rlpy.online.explorers.ConstantEpsilonGreedy method*), 364  
`sample()` (*d3rlpy.online.explorers.LinearDecayEpsilonGreedy method*), 364  
`sample()`    (*d3rlpy.online.explorers.NormalNoise method*), 365  
`sample_action()` (*d3rlpy.algos.AWAC method*), 121  
`sample_action()` (*d3rlpy.algos.AWR method*), 110  
`sample_action()` (*d3rlpy.algos.BC method*), 17  
`sample_action()` (*d3rlpy.algos.BCQ method*), 63  
`sample_action()` (*d3rlpy.algos.BEAR method*), 75  
`sample_action()` (*d3rlpy.algos.COMBO method*), 179  
`sample_action()` (*d3rlpy.algos.CQL method*), 99  
`sample_action()` (*d3rlpy.algos.CRR method*), 87  
`sample_action()` (*d3rlpy.algos.DDPG method*), 28  
`sample_action()` (*d3rlpy.algos.DiscreteAWR method*), 267  
`sample_action()` (*d3rlpy.algos.DiscreteBC method*), 200  
`sample_action()` (*d3rlpy.algos.DiscreteBCQ method*), 245  
`sample_action()` (*d3rlpy.algos.DiscreteCQL method*), 256  
`sample_action()` (*d3rlpy.algos.DiscreteRandomPolicy method*), 277  
`sample_action()` (*d3rlpy.algos.DiscreteSAC method*), 233  
`sample_action()` (*d3rlpy.algos.DoubleDQN method*), 222  
`sample_action()` (*d3rlpy.algos.DQN method*), 211  
`sample_action()` (*d3rlpy.algos.MOPO method*), 167

`sample_action()` (*d3rlpy.algos.PLAS method*), 133  
`sample_action()` (*d3rlpy.algos.PLASWithPerturbation method*), 144  
`sample_action()` (*d3rlpy.algos.RandomPolicy method*), 189  
`sample_action()` (*d3rlpy.algos.SAC method*), 51  
`sample_action()` (*d3rlpy.algos.TD3 method*), 39  
`sample_action()` (*d3rlpy.algos.TD3PlusBC method*), 155  
`sample_action()` (*d3rlpy.ope.DiscreteFQE method*), 352  
`sample_action()` (*d3rlpy.ope.FQE method*), 341  
`save_model()` (*d3rlpy.algos.AWAC method*), 121  
`save_model()` (*d3rlpy.algos.AWR method*), 110  
`save_model()` (*d3rlpy.algos.BC method*), 17  
`save_model()` (*d3rlpy.algos.BCQ method*), 63  
`save_model()` (*d3rlpy.algos.BEAR method*), 75  
`save_model()` (*d3rlpy.algos.COMBO method*), 179  
`save_model()` (*d3rlpy.algos.CQL method*), 99  
`save_model()` (*d3rlpy.algos.CRR method*), 87  
`save_model()` (*d3rlpy.algos.DDPG method*), 28  
`save_model()` (*d3rlpy.algos.DiscreteAWR method*), 267  
`save_model()` (*d3rlpy.algos.DiscreteBC method*), 200  
`save_model()` (*d3rlpy.algos.DiscreteBCQ method*), 245  
`save_model()` (*d3rlpy.algos.DiscreteCQL method*), 256  
`save_model()` (*d3rlpy.algos.DiscreteRandomPolicy method*), 277  
`save_model()` (*d3rlpy.algos.DiscreteSAC method*), 234  
`save_model()` (*d3rlpy.algos.DoubleDQN method*), 222  
`save_model()` (*d3rlpy.algos.DQN method*), 211  
`save_model()` (*d3rlpy.algos.MOPO method*), 168  
`save_model()` (*d3rlpy.algos.PLAS method*), 133  
`save_model()` (*d3rlpy.algos.PLASWithPerturbation method*), 144  
`save_model()` (*d3rlpy.algos.RandomPolicy method*), 190  
`save_model()` (*d3rlpy.algos.SAC method*), 51  
`save_model()` (*d3rlpy.algos.TD3 method*), 40  
`save_model()` (*d3rlpy.algos.TD3PlusBC method*), 156  
`save_model()` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics method*), 373  
`save_model()` (*d3rlpy.ope.DiscreteFQE method*), 352  
`save_model()` (*d3rlpy.ope.FQE method*), 341  
`save_params()` (*d3rlpy.algos.AWAC method*), 122  
`save_params()` (*d3rlpy.algos.AWR method*), 110  
`save_params()` (*d3rlpy.algos.BC method*), 17  
`save_params()` (*d3rlpy.algos.BCQ method*), 63  
`save_params()` (*d3rlpy.algos.BEAR method*), 75  
`save_params()` (*d3rlpy.algos.COMBO method*), 180  
`save_params()` (*d3rlpy.algos.CQL method*), 99  
`save_params()` (*d3rlpy.algos.CRR method*), 87  
`save_params()` (*d3rlpy.algos.DDPG method*), 28  
`save_params()` (*d3rlpy.algos.DiscreteAWR method*), 267  
`save_params()` (*d3rlpy.algos.DiscreteBC method*), 200  
`save_params()` (*d3rlpy.algos.DiscreteBCQ method*), 245  
`save_params()` (*d3rlpy.algos.DiscreteCQL method*), 256  
`save_params()` (*d3rlpy.algos.DiscreteRandomPolicy method*), 277  
`save_params()` (*d3rlpy.algos.DiscreteSAC method*), 234  
`save_params()` (*d3rlpy.algos.DoubleDQN method*), 222  
`save_params()` (*d3rlpy.algos.DQN method*), 211  
`save_params()` (*d3rlpy.algos.MOPO method*), 168  
`save_params()` (*d3rlpy.algos.PLAS method*), 133  
`save_params()` (*d3rlpy.algos.PLASWithPerturbation method*), 144  
`save_params()` (*d3rlpy.algos.RandomPolicy method*), 190  
`save_params()` (*d3rlpy.algos.SAC method*), 51  
`save_params()` (*d3rlpy.algos.TD3 method*), 40  
`save_params()` (*d3rlpy.algos.TD3PlusBC method*), 156  
`save_params()` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics method*), 373  
`save_policy()` (*d3rlpy.algos.DiscreteCQL method*), 256  
`save_policy()` (*d3rlpy.algos.DiscreteRandomPolicy method*), 277  
`save_policy()` (*d3rlpy.algos.DiscreteSAC method*), 234  
`save_policy()` (*d3rlpy.algos.DoubleDQN method*), 222  
`save_policy()` (*d3rlpy.algos.DQN method*), 211  
`save_policy()` (*d3rlpy.algos.MOPO method*), 168  
`save_policy()` (*d3rlpy.algos.PLAS method*), 133  
`save_policy()` (*d3rlpy.algos.PLASWithPerturbation method*), 144  
`save_policy()` (*d3rlpy.algos.RandomPolicy method*), 190

<code>save_policy()</code> ( <i>d3rlpy.algos.SAC method</i> ), 51	<code>set_active_logger()</code> ( <i>d3rlpy.algos.DiscreteBCQ method</i> ), 246
<code>save_policy()</code> ( <i>d3rlpy.algos.TD3 method</i> ), 40	<code>set_active_logger()</code> ( <i>d3rlpy.algos.DiscreteCQL method</i> ), 257
<code>save_policy()</code> ( <i>d3rlpy.algos.TD3PlusBC method</i> ), 156	<code>set_active_logger()</code> ( <i>d3rlpy.algos.DiscreteRandomPolicy method</i> ), 278
<code>save_policy()</code> ( <i>d3rlpy.ope.DiscreteFQE method</i> ), 353	<code>set_active_logger()</code> ( <i>d3rlpy.algos.DiscreteSAC method</i> ), 234
<code>save_policy()</code> ( <i>d3rlpy.ope.FQE method</i> ), 342	<code>set_active_logger()</code> ( <i>d3rlpy.algos.DoubleDQN method</i> ), 223
<code>scaler</code> ( <i>d3rlpy.algos.AWAC attribute</i> ), 124	<code>set_active_logger()</code> ( <i>d3rlpy.algos.DQN method</i> ), 212
<code>scaler</code> ( <i>d3rlpy.algos.AWR attribute</i> ), 113	<code>set_active_logger()</code> ( <i>d3rlpy.algos.MOPO method</i> ), 168
<code>scaler</code> ( <i>d3rlpy.algos.BC attribute</i> ), 19	<code>set_active_logger()</code> ( <i>d3rlpy.algos.PLAS method</i> ), 134
<code>scaler</code> ( <i>d3rlpy.algos.BCQ attribute</i> ), 66	<code>set_active_logger()</code> ( <i>d3rlpy.algos.PLASWithPerturbation method</i> ), 145
<code>scaler</code> ( <i>d3rlpy.algos.BEAR attribute</i> ), 78	<code>set_active_logger()</code> ( <i>d3rlpy.algos.RandomPolicy method</i> ), 190
<code>scaler</code> ( <i>d3rlpy.algos.COMBO attribute</i> ), 182	<code>set_active_logger()</code> ( <i>d3rlpy.algos.SAC method</i> ), 52
<code>scaler</code> ( <i>d3rlpy.algos.CQL attribute</i> ), 102	<code>set_active_logger()</code> ( <i>d3rlpy.algos.TD3 method</i> ), 40
<code>scaler</code> ( <i>d3rlpy.algos.CRR attribute</i> ), 90	<code>set_active_logger()</code> ( <i>d3rlpy.algos.TD3PlusBC method</i> ), 156
<code>scaler</code> ( <i>d3rlpy.algos.DDPG attribute</i> ), 31	<code>set_active_logger()</code> ( <i>d3rlpy.dynamics.ProbabilisticEnsembleDynamics method</i> ), 374
<code>scaler</code> ( <i>d3rlpy.algos.DiscreteAWR attribute</i> ), 270	<code>set_active_logger()</code> ( <i>d3rlpy.ope.DiscreteFQE method</i> ), 353
<code>scaler</code> ( <i>d3rlpy.algos.DiscreteBC attribute</i> ), 203	<code>set_active_logger()</code> ( <i>d3rlpy.ope.FQE method</i> ), 342
<code>scaler</code> ( <i>d3rlpy.algos.DiscreteBCQ attribute</i> ), 248	<code>set_grad_step()</code> ( <i>d3rlpy.algos.AWAC method</i> ), 122
<code>scaler</code> ( <i>d3rlpy.algos.DiscreteCQL attribute</i> ), 259	<code>set_grad_step()</code> ( <i>d3rlpy.algos.AWR method</i> ), 111
<code>scaler</code> ( <i>d3rlpy.algos.DiscreteRandomPolicy attribute</i> ), 280	<code>set_grad_step()</code> ( <i>d3rlpy.algos.BC method</i> ), 17
<code>scaler</code> ( <i>d3rlpy.algos.DiscreteSAC attribute</i> ), 236	<code>set_grad_step()</code> ( <i>d3rlpy.algos.BCQ method</i> ), 64
<code>scaler</code> ( <i>d3rlpy.algos.DoubleDQN attribute</i> ), 225	<code>set_grad_step()</code> ( <i>d3rlpy.algos.BEAR method</i> ), 76
<code>scaler</code> ( <i>d3rlpy.algos.DQN attribute</i> ), 214	<code>set_grad_step()</code> ( <i>d3rlpy.algos.COMBO method</i> ), 180
<code>scaler</code> ( <i>d3rlpy.algos.MOPO attribute</i> ), 170	<code>set_grad_step()</code> ( <i>d3rlpy.algos.CQL method</i> ), 100
<code>scaler</code> ( <i>d3rlpy.algos.PLAS attribute</i> ), 135	<code>set_grad_step()</code> ( <i>d3rlpy.algos.CRR method</i> ), 88
<code>scaler</code> ( <i>d3rlpy.algos.PLASWithPerturbation attribute</i> ), 147	<code>set_grad_step()</code> ( <i>d3rlpy.algos.DDPG method</i> ), 29
<code>scaler</code> ( <i>d3rlpy.algos.RandomPolicy attribute</i> ), 192	<code>set_grad_step()</code> ( <i>d3rlpy.algos.DiscreteAWR method</i> ), 268
<code>scaler</code> ( <i>d3rlpy.algos.SAC attribute</i> ), 54	<code>set_grad_step()</code> ( <i>d3rlpy.algos.DiscreteBC method</i> ), 201
<code>scaler</code> ( <i>d3rlpy.algos.TD3 attribute</i> ), 42	<code>set_grad_step()</code> ( <i>d3rlpy.algos.DiscreteBCQ method</i> ), 246
<code>scaler</code> ( <i>d3rlpy.algos.TD3PlusBC attribute</i> ), 158	<code>set_grad_step()</code> ( <i>d3rlpy.algos.DiscreteCQL method</i> ), 257
<code>scaler</code> ( <i>d3rlpy.dynamics.ProbabilisticEnsembleDynamics attribute</i> ), 376	<code>set_grad_step()</code> ( <i>d3rlpy.algos.DiscreteRandomPolicy method</i> ), 278
<code>scaler</code> ( <i>d3rlpy.ope.DiscreteFQE attribute</i> ), 355	<code>set_grad_step()</code> ( <i>d3rlpy.algos.DiscreteSAC method</i> ), 234
<code>scaler</code> ( <i>d3rlpy.ope.FQE attribute</i> ), 344	<code>set_grad_step()</code> ( <i>d3rlpy.algos.DoubleDQN method</i> ),
<code>set_active_logger()</code> ( <i>d3rlpy.algos.AWAC method</i> ), 122	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.AWR method</i> ), 111	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.BC method</i> ), 17	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.BCQ method</i> ), 64	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.BEAR method</i> ), 76	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.COMBO method</i> ), 180	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.CQL method</i> ), 100	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.CRR method</i> ), 88	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.DDPG method</i> ), 29	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.DiscreteAWR method</i> ), 268	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.DiscreteBC method</i> ), 201	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.DiscreteBCQ method</i> ), 246	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.DiscreteCQL method</i> ), 257	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.DiscreteRandomPolicy method</i> ), 278	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.DiscreteSAC method</i> ), 234	
<code>set_active_logger()</code> ( <i>d3rlpy.algos.DoubleDQN method</i> ),	

223  
**set\_grad\_step()** (*d3rlpy.algos.DQN* method), 212  
**set\_grad\_step()** (*d3rlpy.algos.MOPO* method), 168  
**set\_grad\_step()** (*d3rlpy.algos.PLAS* method), 134  
**set\_grad\_step()** (*d3rlpy.algos.PLASWithPerturbation method*), 145  
**set\_grad\_step()** (*d3rlpy.algos.RandomPolicy method*), 190  
**set\_grad\_step()** (*d3rlpy.algos.SAC* method), 52  
**set\_grad\_step()** (*d3rlpy.algos.TD3* method), 40  
**set\_grad\_step()** (*d3rlpy.algos.TD3PlusBC* method), 156  
**set\_grad\_step()** (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* method), 374  
**set\_grad\_step()** (*d3rlpy.ope.DiscreteFQE* method), 353  
**set\_grad\_step()** (*d3rlpy.ope.FQE* method), 342  
**set\_params()** (*d3rlpy.algos.AWAC* method), 122  
**set\_params()** (*d3rlpy.algos.AWR* method), 111  
**set\_params()** (*d3rlpy.algos.BC* method), 18  
**set\_params()** (*d3rlpy.algos.BCQ* method), 64  
**set\_params()** (*d3rlpy.algos.BEAR* method), 76  
**set\_params()** (*d3rlpy.algos.COMBO* method), 180  
**set\_params()** (*d3rlpy.algos.CQL* method), 100  
**set\_params()** (*d3rlpy.algos.CRR* method), 88  
**set\_params()** (*d3rlpy.algos-DDPG* method), 29  
**set\_params()** (*d3rlpy.algos.DiscreteAWR* method), 268  
**set\_params()** (*d3rlpy.algos.DiscreteBC* method), 201  
**set\_params()** (*d3rlpy.algos.DiscreteBCQ* method), 246  
**set\_params()** (*d3rlpy.algos.DiscreteCQL* method), 257  
**set\_params()** (*d3rlpy.algos.DiscreteRandomPolicy method*), 278  
**set\_params()** (*d3rlpy.algos.DiscreteSAC* method), 235  
**set\_params()** (*d3rlpy.algos.DoubleDQN* method), 223  
**set\_params()** (*d3rlpy.algos.DQN* method), 212  
**set\_params()** (*d3rlpy.algos.MOPO* method), 169  
**set\_params()** (*d3rlpy.algos.PLAS* method), 134  
**set\_params()** (*d3rlpy.algos.PLASWithPerturbation method*), 145  
**set\_params()** (*d3rlpy.algos.RandomPolicy* method), 191  
**set\_params()** (*d3rlpy.algos.SAC* method), 52  
**set\_params()** (*d3rlpy.algos.TD3* method), 41  
**set\_params()** (*d3rlpy.algos.TD3PlusBC* method), 157  
**set\_params()** (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* method), 374  
**set\_params()** (*d3rlpy.ope.DiscreteFQE* method), 353  
**set\_params()** (*d3rlpy.ope.FQE* method), 342  
**SGDFactory** (*class in d3rlpy.models.optimizers*), 316  
**share\_encoder** (*d3rlpy.models.q\_functions.FQFQFunctionFactory* attribute), 286  
**share\_encoder** (*d3rlpy.models.q\_functions.IQNQFunctionFactory* attribute), 284  
**share\_encoder** (*d3rlpy.models.q\_functions.MeanQFunctionFactory* attribute), 282  
**share\_encoder** (*d3rlpy.models.q\_functions.QRQFunctionFactory* attribute), 283  
**size()** (*d3rlpy.dataset.Episode* method), 292  
**size()** (*d3rlpy.dataset.MDPDataset* method), 290  
**size()** (*d3rlpy.dataset.TransitionMiniBatch* method), 296  
**size()** (*d3rlpy.online.buffers.BatchReplayBuffer* method), 367  
**size()** (*d3rlpy.online.buffers.ReplayBuffer* method), 363  
**soft\_opc\_scorer()** (*in module d3rlpy.metrics.scorer*), 328  
**StandardRewardScaler** (*class in d3rlpy.preprocessing*), 311  
**StandardScaler** (*class in d3rlpy.preprocessing*), 305

**T**

**TD3** (*class in d3rlpy.algos*), 31  
**TD3PlusBC** (*class in d3rlpy.algos*), 147  
**td\_error\_scorer()** (*in module d3rlpy.metrics.scorer*), 326  
**terminal** (*d3rlpy.dataset.Episode* attribute), 293  
**terminal** (*d3rlpy.dataset.Transition* attribute), 295  
**terminals** (*d3rlpy.dataset.MDPDataset* attribute), 291  
**terminals** (*d3rlpy.dataset.TransitionMiniBatch* attribute), 297  
**to\_mdp\_dataset()** (*d3rlpy.online.buffers.BatchReplayBuffer* method), 367  
**to\_mdp\_dataset()** (*d3rlpy.online.buffers.ReplayBuffer* method), 363  
**transform()** (*d3rlpy.preprocessing.ClipRewardScaler* method), 314  
**transform()** (*d3rlpy.preprocessing.MinMaxActionScaler* method), 308  
**transform()** (*d3rlpy.preprocessing.MinMaxRewardScaler* method), 311  
**transform()** (*d3rlpy.preprocessing.MinMaxScaler* method), 304  
**transform()** (*d3rlpy.preprocessing.PixelScaler* method), 302  
**transform()** (*d3rlpy.preprocessing.StandardRewardScaler* method), 312  
**transform()** (*d3rlpy.preprocessing.StandardScaler* method), 306  
**transform\_numpy()** (*d3rlpy.preprocessing.ClipRewardScaler* method), 314  
**transform\_numpy()** (*d3rlpy.preprocessing.MinMaxRewardScaler* method), 311  
**transform\_numpy()** (*d3rlpy.preprocessing.StandardRewardScaler* method), 312  
**Transition** (*class in d3rlpy.dataset*), 293  
**TransitionMiniBatch** (*class in d3rlpy.dataset*), 295  
**transitions** (*d3rlpy.dataset.Episode* attribute), 293

transitions (*d3rlpy.dataset.TransitionMiniBatch* attribute), 297  
transitions (*d3rlpy.online.buffers.BatchReplayBuffer* attribute), 368  
transitions (*d3rlpy.online.buffers.ReplayBuffer* attribute), 363  
TYPE (*d3rlpy.models.encoders.DefaultEncoderFactory* attribute), 321  
TYPE (*d3rlpy.models.encoders.DenseEncoderFactory* attribute), 325  
TYPE (*d3rlpy.models.encoders.PixelEncoderFactory* attribute), 322  
TYPE (*d3rlpy.models.encoders.VectorEncoderFactory* attribute), 323  
TYPE (*d3rlpy.models.q\_functions.FQFQFunctionFactory* attribute), 286  
TYPE (*d3rlpy.models.q\_functions.IQNQFunctionFactory* attribute), 284  
TYPE (*d3rlpy.models.q\_functions.MeanQFunctionFactory* attribute), 282  
TYPE (*d3rlpy.models.q\_functions.QRQFunctionFactory* attribute), 283  
TYPE (*d3rlpy.preprocessing.ClipRewardScaler* attribute), 314  
TYPE (*d3rlpy.preprocessing.MinMaxActionScaler* attribute), 309  
TYPE (*d3rlpy.preprocessing.MinMaxRewardScaler* attribute), 311  
TYPE (*d3rlpy.preprocessing.MinMaxScaler* attribute), 304  
TYPE (*d3rlpy.preprocessing.PixelScaler* attribute), 303  
TYPE (*d3rlpy.preprocessing.StandardRewardScaler* attribute), 313  
TYPE (*d3rlpy.preprocessing.StandardScaler* attribute), 306

## U

update() (*d3rlpy.algos.AWAC* method), 123  
update() (*d3rlpy.algos.AWR* method), 111  
update() (*d3rlpy.algos.BC* method), 18  
update() (*d3rlpy.algos.BCQ* method), 64  
update() (*d3rlpy.algos.BEAR* method), 76  
update() (*d3rlpy.algos.COMBO* method), 181  
update() (*d3rlpy.algos.CQL* method), 100  
update() (*d3rlpy.algos.CRR* method), 88  
update() (*d3rlpy.algos.DDPG* method), 29  
update() (*d3rlpy.algos.DiscreteAWR* method), 268  
update() (*d3rlpy.algos.DiscreteBC* method), 201  
update() (*d3rlpy.algos.DiscreteBCQ* method), 246  
update() (*d3rlpy.algos.DiscreteCQL* method), 257  
update() (*d3rlpy.algos.DiscreteRandomPolicy* method), 278  
update() (*d3rlpy.algos.DiscreteSAC* method), 235  
update() (*d3rlpy.algos.DoubleDQN* method), 223

update() (*d3rlpy.algos.DQN* method), 212  
update() (*d3rlpy.algos.MOPO* method), 169  
update() (*d3rlpy.algos.PLAS* method), 134  
update() (*d3rlpy.algos.PLASWithPerturbation* method), 145  
update() (*d3rlpy.algos.RandomPolicy* method), 191  
update() (*d3rlpy.algos.SAC* method), 52  
update() (*d3rlpy.algos.TD3* method), 41  
update() (*d3rlpy.algos.TD3PlusBC* method), 157  
update() (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* method), 374  
update() (*d3rlpy.ope.DiscreteFQE* method), 354  
update() (*d3rlpy.ope.FQE* method), 343

## V

value\_estimation\_std\_scorer() (in module *d3rlpy.metrics.scorer*), 327  
VectorEncoderFactory (class in *d3rlpy.models.encoders*), 322