

---

**d3rlpy**

**Takuma Seno**

**Dec 18, 2021**



# TUTORIALS

<b>1</b>	<b>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</b>	<b>Jupyter Notebooks</b>	<b>7</b>
<b>3</b>	<b>API Reference</b>	<b>9</b>
3.1	Algorithms . . . . .	9
3.2	Q Functions . . . . .	308
3.3	MDPDataSet . . . . .	314
3.4	Datasets . . . . .	325
3.5	Preprocessing . . . . .	329
3.6	Optimizers . . . . .	344
3.7	Network Architectures . . . . .	348
3.8	Metrics . . . . .	355
3.9	Off-Policy Evaluation . . . . .	363
3.10	Save and Load . . . . .	387
3.11	Logging . . . . .	389
3.12	scikit-learn compatibility . . . . .	390
3.13	Online Training . . . . .	392
3.14	(experimental) Model-based Algorithms . . . . .	399
3.15	Stable-Baselines3 Wrapper . . . . .	407
<b>4</b>	<b>Command Line Interface</b>	<b>411</b>
4.1	plot . . . . .	411
4.2	plot-all . . . . .	412
4.3	export . . . . .	413
4.4	record . . . . .	413
4.5	play . . . . .	414
<b>5</b>	<b>Installation</b>	<b>415</b>
5.1	Recommended Platforms . . . . .	415
5.2	Install d3rlpy . . . . .	415
<b>6</b>	<b>Tips</b>	<b>417</b>
6.1	Reproducibility . . . . .	417
6.2	Create your own dataset . . . . .	417

6.3	Learning from image observation . . . . .	418
6.4	Improve performance beyond the original paper . . . . .	419
<b>7</b>	<b>Paper Reproductions</b>	<b>421</b>
<b>8</b>	<b>License</b>	<b>423</b>
<b>9</b>	<b>Indices and tables</b>	<b>425</b>
	<b>Python Module Index</b>	<b>427</b>
	<b>Index</b>	<b>429</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 [MDP Dataset](#).

`d3rlpy` provides suites of datasets for testing algorithms and research. See more documents at [Datasets](#).

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

## 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.IQL</code>	Implicit Q-Learning 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, policy_type='deterministic', 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.
- **policy\_type** (*str*) – the policy type. The available options are ['deterministic', 'stochastic'].
- **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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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*

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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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], List[d3rlpy.dataset.Transition], 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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(isodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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

**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`

**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*)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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  $\phi$ .

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

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

where  $\theta'$  and  $\phi'$  are the target network parameters. There target network parameters are updated every iteration.

$$\theta' \leftarrow \tau \theta + (1 - \tau) \theta'$$

$$\phi' \leftarrow \tau \phi + (1 - \tau) \phi'$$

## 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.preprocessingScaler* 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, deterministic=False, 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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from(*algo*)**

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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*

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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)
```

(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_optim_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], List[d3rlpy.dataset.Transition], 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.buffer.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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]`) – offline dataset 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]]

**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** None

**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*)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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=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, 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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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`

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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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], List[d3rlpy.dataset.Transition], 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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union*[*List*[*d3rlpy.dataset.Episode*], *List*[*d3rlpy.dataset.Transition*], *d3rlpy.dataset.MDPDataset*]) – offline dataset 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]]

**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** None

**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)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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=256, 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.

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

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

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

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.preprocessingScaler* 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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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_q_function_from(cql)
```

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

**Return type** `None`

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

Copies Q-function optimizer states 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_optim_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(episode, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], 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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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]]`**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`**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)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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, 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 algorithm. The major difference from DDPG is that the policy function is represented as combination of conditional VAE and perturbation function in order to remedy extrapolation error emerging from target value estimation.

The encoder and the decoder of the conditional VAE is represented as  $E_\omega$  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.
- **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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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*



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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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], List[d3rlpy.dataset.Transition], 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.buffer.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*, *random\_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union*[*List*[*d3rlpy.dataset.Episode*], *List*[*d3rlpy.dataset.Transition*], *d3rlpy.dataset.MDPDataset*]) – offline dataset 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]]

**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** None

**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)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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_policy_optim_from(algo)`**

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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`

**`copy_q_function_optim_from(algo)`**

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# 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_optim_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], List[d3rlpy.dataset.Transition], 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.buffer.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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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\[d3rlpy.dataset.Episode\]](#), [List\[d3rlpy.dataset.Transition\]](#), [d3rlpy.dataset.MDPDataset](#)]*) – offline dataset 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]]*

**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`

**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*)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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_type='hard', tau=0.005,
                       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.

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

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'].
- **target\_update\_type** (`str`) – target update type. The available options are ['hard', 'soft'].
- **tau** (`float`) – target network synchronization coefficient used with soft target update.
- **update\_actor\_interval** (`int`) – interval to update policy function used with hard target update.
- **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.CRRImpl`) – 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*, *deterministic=False*, *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.
- *deterministic* (`bool`) – flag to collect data with the greedy policy.
- *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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states 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_optim_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`

**`copy_q_function_optim_from(algo)`**

Copies Q-function optimizer states 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_optim_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(epochs, n_steps=1000000)
```

#### Parameters

- **dataset** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], 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, random\_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

**Parameters**

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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\[numpy.ndarray, numpy.ndarray\]`

**reset\_optimizer\_states**()

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`

**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*)

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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.

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

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.

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

where  $N$  is the number of sampled actions.

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

## References

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

### Parameters

- **actor\_learning\_rate** (*float*) – learning rate for policy function.
- **critic\_learning\_rate** (*float*) – learning rate for Q functions.
- **temp\_learning\_rate** (*float*) – learning rate for temperature parameter of SAC.
- **alpha\_learning\_rate** (*float*) – learning rate for  $\alpha$ .
- **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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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`

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

Copies Q-function optimizer states 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_optim_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`], *List*[`d3rlpy.dataset.Transition`], `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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]`) – offline dataset 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]]

**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** None

**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*)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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 TD( $\lambda$ ) to mitigate high variance issue.

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

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

where  $B$  is a constant factor.

## References

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

### Parameters

- **actor\_learning\_rate** (*float*) – learning rate for policy function.
- **critic\_learning\_rate** (*float*) – learning rate for value function.
- **actor\_optim\_factory** (*d3rlpy.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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from(*algo*)**

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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*

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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)
```

(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_optim_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], List[d3rlpy.dataset.Transition], 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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]`) – offline dataset 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*

**reset\_optimizer\_states**()

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** *None*

**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*)

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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*, *deterministic=False*, *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.
- *deterministic* (`bool`) – flag to collect data with the greedy policy.
- *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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states 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_optim_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`

**`copy_q_function_optim_from(algo)`**

Copies Q-function optimizer states 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_optim_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(epochs, n_steps=1000000)
```

### Parameters

- **dataset** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], 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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

**Parameters**

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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\[numpy.ndarray, numpy.ndarray\]`

**reset\_optimizer\_states**()

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`

**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*)

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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, deterministic=False, 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.
- `deterministic` (`bool`) – flag to collect data with the greedy policy.
- `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=1000000)

# 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\_policy\_optim\_from(*algo*)**

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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*

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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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], List[d3rlpy.dataset.Transition], 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.buffer.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, random_steps=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.buffer.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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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*], *List*[*d3rlpy.dataset.Transition*], *d3rlpy.dataset.MDPDataset*]) – offline dataset 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]]`

**reset\_optimizer\_states**()

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`

**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*)

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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.preprocessingScaler* 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*, *deterministic=False*, *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.buffer.Buffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.

- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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`

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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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(episode, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], 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*, *random\_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

#### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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]]`

**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`

**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)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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, 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),
    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.preprocessingScaler* 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*, *deterministic=False*, *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.buffer.Buffer]*) – replay buffer.
- **explorer** (*Optional[d3rlpy.online.explorers.Explorer]*) – action explorer.

- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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`

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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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(episode, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], 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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

#### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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]]`

**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`

**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)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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.IQL**

```
class d3rlpy.algos.IQL(*, actor_learning_rate=0.0003, critic_learning_rate=0.0003,
                        value_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),
                        value_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',
                        value_encoder_factory='default', batch_size=256, n_frames=1, n_steps=1,
                        gamma=0.99, tau=0.005, n_critics=2, expectile=0.7, weight_temp=3.0,
                        max_weight=100.0, use_gpu=False, scaler=None, action_scaler=None,
                        reward_scaler=None, impl=None, **kwargs)
```

Implicit Q-Learning algorithm.

IQL is the offline RL algorithm that avoids ever querying values of unseen actions while still being able to perform multi-step dynamic programming updates.

There are three functions to train in IQL. First the state-value function is trained via expectile regression.

$$L_V(\psi) = \mathbb{E}_{(s,a) \sim D} [L_2^\tau(Q_\theta(s, a) - V_\psi(s))]$$

where  $L_2^\tau(u) = |\tau - \mathbb{I}(u < 0)|u^2$ .

The Q-function is trained with the state-value function to avoid query the actions.

$$L_Q(\theta) = \mathbb{E}_{(s,a,r,a') \sim D} [(r + \gamma V_\psi(s') - Q_\theta(s, a))^2]$$

Finally, the policy function is trained by using advantage weighted regression.

$$L_\pi(\phi) = \mathbb{E}_{(s,a) \sim D} [\exp(\beta(Q_\theta - V_\psi(s))) \log \pi_\phi(a|s)]$$

**References**

- [Kostrikov et al., Offline Reinforcement Learning with Implicit Q-Learning.](#)

**Parameters**

- **actor\_learning\_rate** (*float*) – learning rate for policy function.
- **critic\_learning\_rate** (*float*) – learning rate for Q functions.
- **value\_learning\_rate** (*float*) – learning rate for state-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.

- **value\_optim\_factory** (`d3rlpy.models.optimizers.OptimizerFactory`) – optimizer factory for the value function.
- **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.
- **value\_encoder\_factory** (`d3rlpy.models.encoders.EncoderFactory` or `str`) – encoder factory for the value function.
- **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.
- **expectile** (`float`) – the expectile value for value function training.
- **weight\_temp** (`float`) – inverse temperature value represented as  $\beta$ .
- **max\_weight** (`float`) – the maximum advantage weight value to clip.
- **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.iql_impl.IQLImpl`) – 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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from(*algo*)**

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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`

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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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(episode, n_steps=1000000)
```

**Parameters**

- **dataset** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], 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*, *random\_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

#### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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
```

(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]]`**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`**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)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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.

$$r_{t+1}^{\sim} = r_{t+1} - \lambda \max_{i=1}^N \|\Sigma_i(s_t, a_t)\|$$

where  $\Sigma(s_t, a_t)$  is the estimated variance. Finally, the generated transitions  $(s_t, a_t, r_{t+1}^{\sim}, s_{t+1})$  are appended to dataset  $D$ . This generation process starts with randomly sampled `n_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 ratio 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, deterministic=False, 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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# transfer to online algorithm
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\_policy\_optim\_from**(`algo`)

Copies policy optimizer states 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_optim_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`

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

Copies Q-function optimizer states 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_optim_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`], *List*[`d3rlpy.dataset.Transition`], `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.buffer.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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(isodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[[List\[d3rlpy.dataset.Episode\]](#), [List\[d3rlpy.dataset.Transition\]](#), [d3rlpy.dataset.MDPDataset](#)]*) – offline dataset 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
```

(continues on next page)

(continued from previous page)

```

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]]`

**reset\_optimizer\_states**()

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`

**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*)

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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.

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

---

**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.preprocessingScaler* 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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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*

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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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], List[d3rlpy.dataset.Transition], 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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(isodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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
```

(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\[numpy.ndarray, numpy.ndarray\]]`**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`**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)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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, deterministic=False, 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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from(*algo*)**

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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`

**`copy_q_function_optim_from(algo)`**

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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(episode, n_steps=1000000)
```

**Parameters**

- **`dataset`** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], 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*, *random\_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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
```

(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\[numpy.ndarray, numpy.ndarray\]]`**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`**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)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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 version 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*) – 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.
- **beta** (*float*) – regularization factor.
- **use\_gpu** (*bool*, *int* or *d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.
- **scaler** (*d3rlpy.preprocessingScaler* or *str*) – preprocessor. The available options are ['pixel', 'min\_max', 'standard']
- **impl** (*d3rlpy.algos.torch.bc\_impl.DiscreteBCImpl*) – 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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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_policy_optim_from(algo)`**

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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`

**`copy_q_function_optim_from(algo)`**

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# 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_optim_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], List[d3rlpy.dataset.Transition], 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*, *random\_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[[List\[d3rlpy.dataset.Episode\]](#), [List\[d3rlpy.dataset.Transition\]](#), [d3rlpy.dataset.MDPDataset](#)]*) – offline dataset 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*

**reset\_optimizer\_states**()

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** *None*

**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*)

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from(*algo*)**

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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`

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

Copies Q-function optimizer states 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_optim_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`], *List*[`d3rlpy.dataset.Transition`], `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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]`) – offline dataset 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]]

**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** None

**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*)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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.

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

where  $\theta'$  is the target network parameter. The target network parameter is synchronized every *target\_update\_interval* iterations.

## References

- 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*, *deterministic=False*, *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.
- *deterministic* (`bool`) – flag to collect data with the greedy policy.
- *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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states 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_optim_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`

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

Copies Q-function optimizer states 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_optim_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], List[d3rlpy.dataset.Transition], 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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(isodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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]]`

**reset\_optimizer\_states**()

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`

**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*)

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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, ac-
                                tor_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*, *deterministic=False*, *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.
- *deterministic* (`bool`) – flag to collect data with the greedy policy.
- *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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states 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_optim_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`

**`copy_q_function_optim_from(algo)`**

Copies Q-function optimizer states 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_optim_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(epochs, n_steps=1000000)
```

### Parameters

- **dataset** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], 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, random\_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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]]`

**reset\_optimizer\_states**()

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`

**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*)

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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.preprocessingScaler` 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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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*

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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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(epochs, n_steps=1000000)
```

**Parameters**

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], 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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(isodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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
```

(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]]`**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`**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)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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.

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

**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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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`

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

Copies Q-function optimizer states 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_optim_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`], *List*[`d3rlpy.dataset.Transition`], `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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]`) – offline dataset 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]]

**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** None

**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*)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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 version of Advantage-Weighted Regression algorithm.

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

The value function is trained as a supervised regression problem.

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

where  $R_t$  is approximated using TD( $\lambda$ ) to mitigate high variance issue.

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

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

where  $B$  is a constant factor.

## References

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

### Parameters

- **actor\_learning\_rate** (*float*) – learning rate for policy function.
- **critic\_learning\_rate** (*float*) – learning rate for value function.
- **actor\_optim\_factory** (*d3rlpy.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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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`

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

Copies Q-function optimizer states 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_optim_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`], *List*[`d3rlpy.dataset.Transition`], `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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (`Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]`) – offline dataset 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*

**reset\_optimizer\_states**()

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** *None*

**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*)

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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*

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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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], List[d3rlpy.dataset.Transition], 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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(isodes):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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
```

(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\[numpy.ndarray, numpy.ndarray\]]`**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`**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)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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.

<code>d3rlpy.models.q_functions.</code> <code>MeanQFunctionFactory</code>	Standard Q function factory class.
<code>d3rlpy.models.q_functions.</code> <code>QRQFunctionFactory</code>	Quantile Regression Q function factory class.
<code>d3rlpy.models.q_functions.</code> <code>IQNQFunctionFactory</code>	Implicit Quantile Network Q function factory class.
<code>d3rlpy.models.q_functions.</code> <code>FQQFunctionFactory</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:
    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. See discussion in [issue #98](#).

<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)

# 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']
```

(continues on next page)



(continued from previous page)

```

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')

```

(continues on next page)

```
# 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`)

### **is\_discrete**

Returns flag of discrete action-space.

**Returns** True if action-space is discrete.

**Return type** `bool`

### **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`

**next\_rewards**

Returns mini-batch of rewards at  $t+n$ .



**Returns** rewards at  $t+n$ .

**Return type** `numpy.ndarray`

#### observations

Returns mini-batch of observations at  $t$ .

**Returns** observations at  $t$ .

**Return type** `numpy.ndarray` or `torch.Tensor`

#### rewards

Returns mini-batch of rewards at  $t$ .

**Returns** rewards at  $t$ .

**Return type** `numpy.ndarray`

#### terminals

Returns mini-batch of terminal flags at  $t+n$ .

**Returns** terminal flags at  $t+n$ .

**Return type** `numpy.ndarray`

#### transitions

Returns transitions.

**Returns** list of transitions.

**Return type** `d3rlpy.dataset.Transition`

## 3.4 Datasets

d3rlpy provides datasets for experimenting data-driven deep reinforcement learning algorithms.

<code>d3rlpy.datasets.get_cartpole</code>	Returns cartpole dataset and environment.
<code>d3rlpy.datasets.get_pendulum</code>	Returns pendulum dataset and environment.
<code>d3rlpy.datasets.get_pybullet</code>	Returns pybullet dataset and environment.
<code>d3rlpy.datasets.get_atari</code>	Returns atari dataset and environment.
<code>d3rlpy.datasets.get_d4rl</code>	Returns d4rl dataset and environment.
<code>d3rlpy.datasets.get_dataset</code>	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)

```
cql = CQL(scaler='pixel')  
cql.fit(dataset.episodes)
```

## Methods

### **fit**(*transitions*)

Estimates scaling parameters from dataset.

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

**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 transitions
transitions = []
for episode in dataset.episodes:
    transitions += episode.transitions
cql.fit(transitions)
```

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**(*transitions*)

Estimates scaling parameters from dataset.

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

**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
transitions = []
for episode in dataset.episodes:
    transitions += episode.transitions
cql.fit(transitions)
```

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**(transitions)

Estimates scaling parameters from dataset.

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

**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] = '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')
```

(continues on next page)

(continued from previous page)

```
# you don't need to take care of postprocessing at production
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)
```

---

*d3rlpy.preprocessing.MinMaxActionScaler*

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 transitions
transitions = []
for episode in dataset.episodes:
    transitions += episode.transitions
cql.fit(transitions)
```

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**(*transitions*)

Estimates scaling parameters from dataset.

**Parameters** **transitions** (`List[d3rlpy.dataset.Transition]`) – a list of transition 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 and MultiplyRewardScaler must be initialized manually
reward_scaler = ClipRewardScaler(-1.0, 1.0)
cql = CQL(reward_scaler=reward_scaler)
```

<a href="#"><code>d3rlpy.preprocessing.MinMaxRewardScaler</code></a>	Min-Max reward normalization preprocessing.
<a href="#"><code>d3rlpy.preprocessing.StandardRewardScaler</code></a>	Reward standardization preprocessing.
<a href="#"><code>d3rlpy.preprocessing.ClipRewardScaler</code></a>	Reward clipping preprocessing.
<a href="#"><code>d3rlpy.preprocessing.MultiplyRewardScaler</code></a>	Multiplication reward preprocessing.

**d3rlpy.preprocessing.MinMaxRewardScaler**

**class** d3rlpy.preprocessing.MinMaxRewardScaler(*dataset=None, minimum=None, maximum=None, multiplier=1.0*)

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)
```

**Parameters**

- **dataset** (`d3rlpy.dataset.MDPDataset`) – dataset object.
- **minimum** (*float*) – minimum value.
- **maximum** (*float*) – maximum value.
- **multiplier** (*float*) – constant multiplication value.

**Methods**

**fit**(*transitions*)

Estimates scaling parameters from dataset.

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

**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, multiplier=1.0*)

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)
```

### Parameters

- **dataset** (`d3rlpy.dataset.MDPDataset`) – dataset object.
- **mean** (`float`) – mean value.
- **std** (`float`) – standard deviation value.
- **eps** (`float`) – constant value to avoid zero-division.
- **multiplier** (`float`) – constant multiplication value

### Methods

#### **fit**(*transitions*)

Estimates scaling parameters from dataset.

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

**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`, `multiplier=1.0`)

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.
- **multiplier** (`float`) – constant multiplication value.

### Methods

**fit**(`transitions`)

Estimates scaling parameters from dataset.

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

**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]` = `'clip'`

## d3rlpy.preprocessing.MultiplyRewardScaler

**class** d3rlpy.preprocessing.MultiplyRewardScaler(*multiplier=None*)  
 Multiplication reward preprocessing.

This preprocessor multiplies rewards by a constant number.

```
from d3rlpy.preprocessing import MultiplyRewardScaler

# multiply rewards by 10
reward_scaler = MultiplyRewardScaler(10.0)

cql = CQL(reward_scaler=reward_scaler)
```

**Parameters** *multiplier* (*float*) – constant multiplication value.

## Methods

**fit**(*transitions*)

Estimates scaling parameters from dataset.

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

**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]` = 'multiply'

## 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
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.

---

continues on next page

Table 9 – continued from previous page

<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 import Adam
from d3rlpy.optimizers import OptimizerFactory
from d3rlpy.algos import DQN

factory = OptimizerFactory(Adam, eps=0.001)

dqn = DQN(optim_factory=factory)
```

#### Parameters

- **optim\_cls** (`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 network 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, observation_shape, feature_size):
        self.feature_size = feature_size
        self.fc1 = nn.Linear(observation_shape[0], 64)
        self.fc2 = nn.Linear(64, feature_size)

    def forward(self, x):
        h = torch.relu(self.fc1(x))
```

(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, observation_shape, action_size, feature_size):
        self.feature_size = feature_size
        self.fc1 = nn.Linear(observation_shape[0] + action_size, 64)
        self.fc2 = nn.Linear(64, feature_size)

    def forward(self, x, action): # action is also given
        h = torch.cat([x, action], dim=1)
        h = torch.relu(self.fc1(h))
        h = torch.relu(self.fc2(h))
        return h

    def get_feature_size(self):
        return self.feature_size

class CustomEncoderFactory(EncoderFactory):
    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(self, 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.
<code>d3rlpy.metrics.scorer.discounted_sum_of_advantage_scorer</code>	Returns average of discounted sum of advantage.
<code>d3rlpy.metrics.scorer.average_value_estimation_scorer</code>	Returns average value estimation.
<code>d3rlpy.metrics.scorer.value_estimation_std_scorer</code>	Returns standard deviation of value estimation.
<code>d3rlpy.metrics.scorer.initial_state_value_estimation_scorer</code>	Returns mean estimated action-values at the initial states.
<code>d3rlpy.metrics.scorer.soft_opc_scorer</code>	Returns Soft Off-Policy Classification metrics.
<code>d3rlpy.metrics.scorer.continuous_action_diff_scorer</code>	Returns squared difference of actions between algorithm and dataset.
<code>d3rlpy.metrics.scorer.discrete_action_match_scorer</code>	Returns percentage of identical actions between algorithm and dataset.
<code>d3rlpy.metrics.scorer.evaluate_on_environment</code>	Returns scorer function of evaluation on environment.
<code>d3rlpy.metrics.comparer.compare_continuous_action_diff</code>	Returns scorer function of action difference between algorithms.
<code>d3rlpy.metrics.comparer.compare_discrete_action_match</code>	Returns scorer function of action matches between algorithms.

#### `d3rlpy.metrics.scorer.td_error_scorer`

`d3rlpy.metrics.scorer.td_error_scorer(algo, episodes)`

Returns average TD error.

This metrics suggests how Q functions overfit to training sets. If the TD error is large, the Q functions are overfitting.

$$\mathbb{E}_{s_t, a_t, r_{t+1}, s_{t+1} \sim D} [(Q_\theta(s_t, a_t) - r_{t+1} - \gamma \max_a Q_\theta(s_{t+1}, a))^2]$$

##### Parameters

- **algo** (`d3rlpy.metrics.scorer.AlgoProtocol`) – algorithm.
- **episodes** (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**Returns** 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.

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  $A(s_t, a_t) = Q_\theta(s_t, a_t) - \mathbb{E}_{a \sim \pi}[Q_\theta(s_t, a)]$ .



## References

- [Murphy., A generalization error for Q-Learning.](#)

### Parameters

- **algo** (`d3rlpy.metrics.scorer.AlgoProtocol`) – algorithm.
- **episodes** (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**Returns** 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.

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** 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.

This metrics suggests how confident Q functions are for the given episodes. This metrics will be more accurate with *bootstrap* enabled and the larger *n\_critics* at algorithm. If standard deviation of value estimation is large, the Q functions are overfitting to the training set.

$$\mathbb{E}_{s_t \sim D, a \sim \arg\max_a Q_\theta(s_t, a)} [Q_{\text{std}}(s_t, a)]$$

where  $Q_{\text{std}}(s, a)$  is a standard deviation of action-value estimation over ensemble functions.

### Parameters

- **algo** (`d3rlpy.metrics.scorer.AlgoProtocol`) – algorithm.
- **episodes** (`List[d3rlpy.dataset.Episode]`) – list of episodes.

**Returns** 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 function is evaluating gaps of action-value estimation between the success episodes and the all episodes. If the learned Q-function is optimal, action-values in success episodes are expected to be higher than the others. The success episode is defined as an episode with a return above the given threshold.

$$\mathbb{E}_{s,a \sim D_{success}}[Q(s,a)] - \mathbb{E}_{s,a \sim D}[Q(s,a)]$$

```
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** 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 episodes are near-optimal, the large percentage would be better.

$$\frac{1}{N} \sum^N \mathbb{I}\{a_t = \operatorname{argmax}_a Q_\theta(s_t, a)\}$$

**Parameters**

- `algo` (d3rlpy.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.dynamics_observation_prediction_error_scorer</code>	Returns MSE of observation prediction.
<code>d3rlpy.metrics.scorer.dynamics_reward_prediction_error_scorer</code>	Returns MSE of reward prediction.
<code>d3rlpy.metrics.scorer.dynamics_prediction_variance_scorer</code>	Returns prediction variance of ensemble dynamics.

**d3rlpy.metrics.scorer.dynamics\_observation\_prediction\_error\_scorer**

`d3rlpy.metrics.scorer.dynamics_observation_prediction_error_scorer(dynamics, episodes)`

Returns MSE of observation prediction.

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** 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.

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** 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.

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** 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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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*

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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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], List[d3rlpy.dataset.Transition], 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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union[List[d3rlpy.dataset.Episode], List[d3rlpy.dataset.Transition], d3rlpy.dataset.MDPDataset]*) – offline dataset 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
```

(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]]`**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** `None`**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)**

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

The format will be automatically detected by the file name.



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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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.preprocessingScaler* 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.FQEIml*) – 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*, *deterministic=False*, *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.
- **deterministic** (*bool*) – flag to collect data with the greedy policy.
- **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=1000000)

# 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\_policy\_optim\_from**(*algo*)

Copies policy optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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=1000000)

# 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*

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

Copies Q-function optimizer states from the given algorithm.

```
# pretrain with static dataset
cql = d3rlpy.algos.CQL()
cql.fit(dataset, n_steps=1000000)

# transfer to online algorithm
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_policy_optim_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], List[d3rlpy.dataset.Transition], 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, random_steps=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.
- **random\_steps** (*int*) – the steps for the initial random exploration.
- **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(epochs):
    my_plot(metrics)
    algo.save_model(my_path)
```

### Parameters

- **dataset** (*Union*[*List*[*d3rlpy.dataset.Episode*], *List*[*d3rlpy.dataset.Transition*], *d3rlpy.dataset.MDPDataset*]) – offline dataset 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]]

**reset\_optimizer\_states()**

Resets optimizer states.

This is especially useful when fine-tuning policies with setting initial optimizer states.

**Return type** None

**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)**

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

The format will be automatically detected by the file name.

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

# save as ONNX
algo.save_policy('policy.onnx')
```

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.

**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')
```

#### 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

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()
```

(continues on next page)

(continued from previous page)

```
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)
```

(continues on next page)

(continued from previous page)

```

# 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_steps=30000, # the number of total steps to train.
               n_steps_per_epoch=1000,
               update_interval=10) # update parameters every 10 steps.

```

## Replay Buffer

---

`d3rlpy.online.buffers.ReplayBuffer`

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***clip\_episode**()

Clips the current 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*

## Explorers

<i>d3rlpy.online.explorers.ConstantEpsilonGreedy</i>	$\epsilon$ -greedy explorer with constant $\epsilon$ .
<i>d3rlpy.online.explorers.LinearDecayEpsilonGreedy</i>	$\epsilon$ -greedy explorer with linear decay schedule.
<i>d3rlpy.online.explorers.NormalNoise</i>	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.buffer 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,
                                       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.buffer.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*

```
clip_episode()
```

Clips the current 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

```

(continues on next page)

(continued from previous page)

```

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, op-
    tim_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.Actionscalers` 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], List[d3rlpy.dataset.Transition], 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*], *List*[*d3rlpy.dataset.Transition*], *d3rlpy.dataset.MDPDataset*]) – offline dataset 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
```

(continues on next page)

(continued from previous page)

```
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:

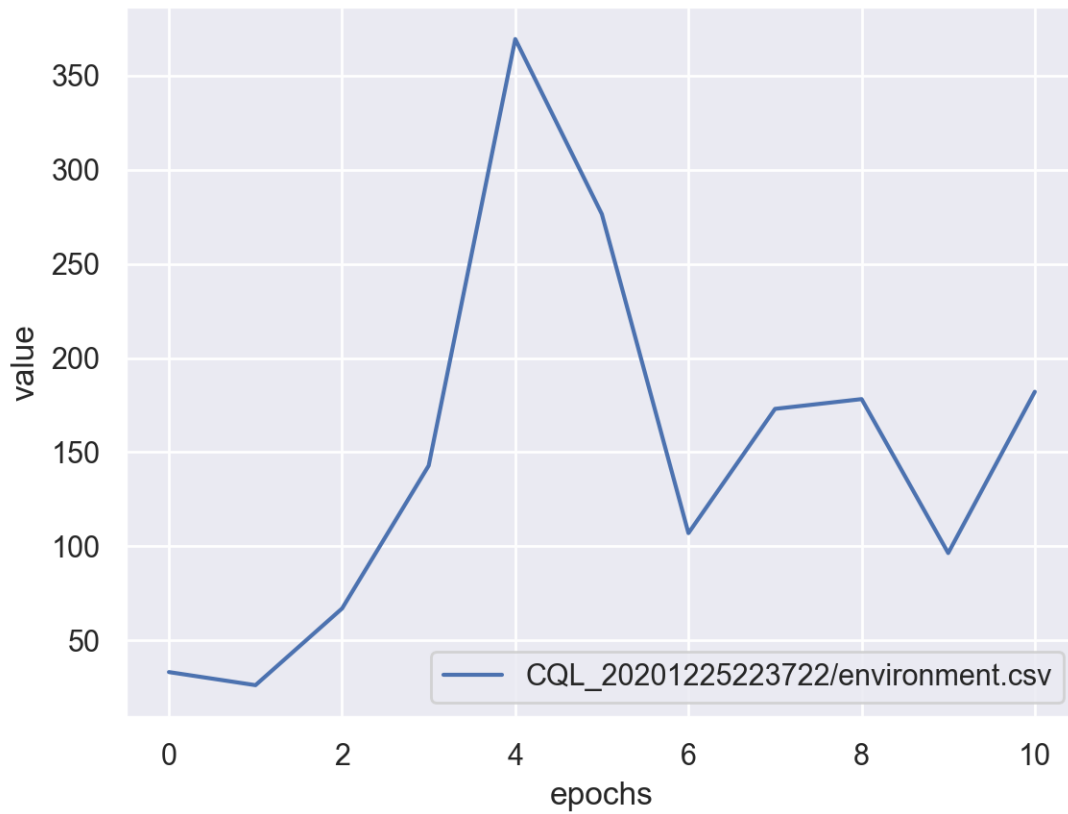
```
$ d3rlpy plot <path> [<path>...]
```

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.
--save	flag to save the plot as an image.

example:

```
$ d3rlpy plot d3rlpy_logs/CQL_20201224224314/environment.csv
```



## 4.2 plot-all

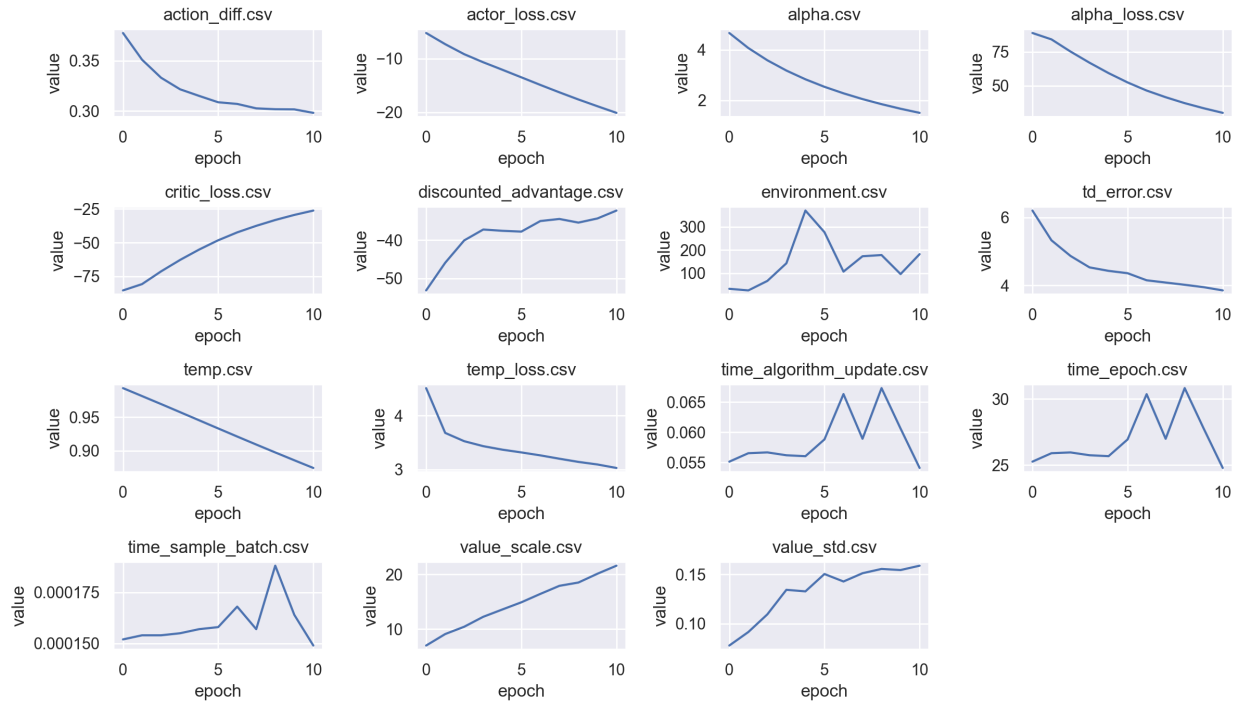
Plot the all metrics saved in the directory:

```
$ 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
<code>--format</code>	model format (torchscript, onnx).
<code>--params-json</code>	explicitly specify params.json.
<code>--out</code>	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 recored 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. See discussion in [issue #98](#).

If you have an access to the environment, you can automate the process.

```
import gym

import d3rlpy

env = gym.make("Hopper-v2")

# collect with random policy
random_policy = d3rlpy.algos.RandomPolicy()
random_buffer = d3rlpy.online.buffers.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.buffers.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.buffers.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)
```

(continues on next page)

(continued from previous page)

```
rewards = np.random.random(1000000)
terminals = np.random.randint(2, size=1000000)

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.

All the experimental results are available in [d3rlpy-benchmarks](#) repository.



## **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.



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### d

- `d3rlpy`, 9
- `d3rlpy.algos`, 9
- `d3rlpy.dataset`, 314
- `d3rlpy.datasets`, 325
- `d3rlpy.dynamics`, 399
- `d3rlpy.metrics`, 355
- `d3rlpy.models.encoders`, 348
- `d3rlpy.models.optimizers`, 344
- `d3rlpy.models.q_functions`, 308
- `d3rlpy.online`, 392
- `d3rlpy.ope`, 363
- `d3rlpy.preprocessing`, 329





## Symbols

`__getitem__()` (*d3rlpy.dataset.Episode method*), 319  
`__getitem__()` (*d3rlpy.dataset.MDPDataset method*), 316  
`__getitem__()` (*d3rlpy.dataset.TransitionMiniBatch method*), 323  
`__iter__()` (*d3rlpy.dataset.Episode method*), 319  
`__iter__()` (*d3rlpy.dataset.MDPDataset method*), 316  
`__iter__()` (*d3rlpy.dataset.TransitionMiniBatch method*), 323  
`__len__()` (*d3rlpy.dataset.Episode method*), 319  
`__len__()` (*d3rlpy.dataset.MDPDataset method*), 316  
`__len__()` (*d3rlpy.dataset.TransitionMiniBatch method*), 323  
`__len__()` (*d3rlpy.online.buffers.BatchReplayBuffer method*), 398  
`__len__()` (*d3rlpy.online.buffers.ReplayBuffer method*), 393

## A

`action` (*d3rlpy.dataset.Transition attribute*), 322  
`action_scaler` (*d3rlpy.algos.AWAC attribute*), 130  
`action_scaler` (*d3rlpy.algos.AWR attribute*), 118  
`action_scaler` (*d3rlpy.algos.BC attribute*), 19  
`action_scaler` (*d3rlpy.algos.BCQ attribute*), 68  
`action_scaler` (*d3rlpy.algos.BEAR attribute*), 81  
`action_scaler` (*d3rlpy.algos.COMBO attribute*), 204  
`action_scaler` (*d3rlpy.algos.CQL attribute*), 106  
`action_scaler` (*d3rlpy.algos.CRR attribute*), 94  
`action_scaler` (*d3rlpy.algos.DDPG attribute*), 31  
`action_scaler` (*d3rlpy.algos.DiscreteAWR attribute*), 296  
`action_scaler` (*d3rlpy.algos.DiscreteBC attribute*), 226  
`action_scaler` (*d3rlpy.algos.DiscreteBCQ attribute*), 273  
`action_scaler` (*d3rlpy.algos.DiscreteCQL attribute*), 284  
`action_scaler` (*d3rlpy.algos.DiscreteRandomPolicy attribute*), 307  
`action_scaler` (*d3rlpy.algos.DiscreteSAC attribute*), 261

`action_scaler` (*d3rlpy.algos.DoubleDQN attribute*), 249  
`action_scaler` (*d3rlpy.algos.DQN attribute*), 237  
`action_scaler` (*d3rlpy.algos.IQL attribute*), 179  
`action_scaler` (*d3rlpy.algos.MOPO attribute*), 191  
`action_scaler` (*d3rlpy.algos.PLAS attribute*), 142  
`action_scaler` (*d3rlpy.algos.PLASWithPerturbation attribute*), 154  
`action_scaler` (*d3rlpy.algos.RandomPolicy attribute*), 215  
`action_scaler` (*d3rlpy.algos.SAC attribute*), 56  
`action_scaler` (*d3rlpy.algos.TD3 attribute*), 43  
`action_scaler` (*d3rlpy.algos.TD3PlusBC attribute*), 166  
`action_scaler` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics attribute*), 406  
`action_scaler` (*d3rlpy.ope.DiscreteFQE attribute*), 385  
`action_scaler` (*d3rlpy.ope.FQE attribute*), 374  
`action_size` (*d3rlpy.algos.AWAC attribute*), 130  
`action_size` (*d3rlpy.algos.AWR attribute*), 118  
`action_size` (*d3rlpy.algos.BC attribute*), 19  
`action_size` (*d3rlpy.algos.BCQ attribute*), 68  
`action_size` (*d3rlpy.algos.BEAR attribute*), 81  
`action_size` (*d3rlpy.algos.COMBO attribute*), 204  
`action_size` (*d3rlpy.algos.CQL attribute*), 106  
`action_size` (*d3rlpy.algos.CRR attribute*), 94  
`action_size` (*d3rlpy.algos.DDPG attribute*), 31  
`action_size` (*d3rlpy.algos.DiscreteAWR attribute*), 296  
`action_size` (*d3rlpy.algos.DiscreteBC attribute*), 226  
`action_size` (*d3rlpy.algos.DiscreteBCQ attribute*), 273  
`action_size` (*d3rlpy.algos.DiscreteCQL attribute*), 284  
`action_size` (*d3rlpy.algos.DiscreteRandomPolicy attribute*), 307  
`action_size` (*d3rlpy.algos.DiscreteSAC attribute*), 261  
`action_size` (*d3rlpy.algos.DoubleDQN attribute*), 249  
`action_size` (*d3rlpy.algos.DQN attribute*), 237  
`action_size` (*d3rlpy.algos.IQL attribute*), 179  
`action_size` (*d3rlpy.algos.MOPO attribute*), 191  
`action_size` (*d3rlpy.algos.PLAS attribute*), 142  
`action_size` (*d3rlpy.algos.PLASWithPerturbation attribute*), 154  
`action_size` (*d3rlpy.algos.RandomPolicy attribute*),

- 215
- `action_size` (*d3rlpy.algos.SAC* attribute), 56
- `action_size` (*d3rlpy.algos.TD3* attribute), 43
- `action_size` (*d3rlpy.algos.TD3PlusBC* attribute), 166
- `action_size` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* attribute), 406
- `action_size` (*d3rlpy.ope.DiscreteFQE* attribute), 385
- `action_size` (*d3rlpy.ope.FQE* attribute), 374
- `actions` (*d3rlpy.dataset.Episode* attribute), 320
- `actions` (*d3rlpy.dataset.MDPDataset* attribute), 318
- `actions` (*d3rlpy.dataset.TransitionMiniBatch* attribute), 324
- `active_logger` (*d3rlpy.algos.AWAC* attribute), 130
- `active_logger` (*d3rlpy.algos.AWR* attribute), 118
- `active_logger` (*d3rlpy.algos.BC* attribute), 19
- `active_logger` (*d3rlpy.algos.BCQ* attribute), 68
- `active_logger` (*d3rlpy.algos.BEAR* attribute), 81
- `active_logger` (*d3rlpy.algos.COMBO* attribute), 204
- `active_logger` (*d3rlpy.algos.CQL* attribute), 106
- `active_logger` (*d3rlpy.algos.CRR* attribute), 94
- `active_logger` (*d3rlpy.algos.DDPG* attribute), 31
- `active_logger` (*d3rlpy.algos.DiscreteAWR* attribute), 296
- `active_logger` (*d3rlpy.algos.DiscreteBC* attribute), 226
- `active_logger` (*d3rlpy.algos.DiscreteBCQ* attribute), 273
- `active_logger` (*d3rlpy.algos.DiscreteCQL* attribute), 284
- `active_logger` (*d3rlpy.algos.DiscreteRandomPolicy* attribute), 307
- `active_logger` (*d3rlpy.algos.DiscreteSAC* attribute), 261
- `active_logger` (*d3rlpy.algos.DoubleDQN* attribute), 249
- `active_logger` (*d3rlpy.algos.DQN* attribute), 237
- `active_logger` (*d3rlpy.algos.IQL* attribute), 179
- `active_logger` (*d3rlpy.algos.MOPO* attribute), 191
- `active_logger` (*d3rlpy.algos.PLAS* attribute), 142
- `active_logger` (*d3rlpy.algos.PLASWithPerturbation* attribute), 155
- `active_logger` (*d3rlpy.algos.RandomPolicy* attribute), 215
- `active_logger` (*d3rlpy.algos.SAC* attribute), 56
- `active_logger` (*d3rlpy.algos.TD3* attribute), 43
- `active_logger` (*d3rlpy.algos.TD3PlusBC* attribute), 167
- `active_logger` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* attribute), 406
- `active_logger` (*d3rlpy.ope.DiscreteFQE* attribute), 385
- `active_logger` (*d3rlpy.ope.FQE* attribute), 374
- `AdamFactory` (class in *d3rlpy.models.optimizers*), 346
- `add_additional_data()` (*d3rlpy.dataset.TransitionMiniBatch* method), 323
- `append()` (*d3rlpy.dataset.MDPDataset* method), 316
- `append()` (*d3rlpy.online.buffers.BatchReplayBuffer* method), 398
- `append()` (*d3rlpy.online.buffers.ReplayBuffer* method), 393
- `append_episode()` (*d3rlpy.online.buffers.BatchReplayBuffer* method), 398
- `append_episode()` (*d3rlpy.online.buffers.ReplayBuffer* method), 394
- `average_value_estimation_scorer()` (in module *d3rlpy.metrics.scorer*), 357
- AWAC* (class in *d3rlpy.algos*), 119
- AWR* (class in *d3rlpy.algos*), 108
- ## B
- `batch_size` (*d3rlpy.algos.AWAC* attribute), 130
- `batch_size` (*d3rlpy.algos.AWR* attribute), 118
- `batch_size` (*d3rlpy.algos.BC* attribute), 19
- `batch_size` (*d3rlpy.algos.BCQ* attribute), 68
- `batch_size` (*d3rlpy.algos.BEAR* attribute), 81
- `batch_size` (*d3rlpy.algos.COMBO* attribute), 204
- `batch_size` (*d3rlpy.algos.CQL* attribute), 106
- `batch_size` (*d3rlpy.algos.CRR* attribute), 94
- `batch_size` (*d3rlpy.algos.DDPG* attribute), 31
- `batch_size` (*d3rlpy.algos.DiscreteAWR* attribute), 296
- `batch_size` (*d3rlpy.algos.DiscreteBC* attribute), 226
- `batch_size` (*d3rlpy.algos.DiscreteBCQ* attribute), 273
- `batch_size` (*d3rlpy.algos.DiscreteCQL* attribute), 284
- `batch_size` (*d3rlpy.algos.DiscreteRandomPolicy* attribute), 307
- `batch_size` (*d3rlpy.algos.DiscreteSAC* attribute), 261
- `batch_size` (*d3rlpy.algos.DoubleDQN* attribute), 249
- `batch_size` (*d3rlpy.algos.DQN* attribute), 237
- `batch_size` (*d3rlpy.algos.IQL* attribute), 179
- `batch_size` (*d3rlpy.algos.MOPO* attribute), 191
- `batch_size` (*d3rlpy.algos.PLAS* attribute), 142
- `batch_size` (*d3rlpy.algos.PLASWithPerturbation* attribute), 155
- `batch_size` (*d3rlpy.algos.RandomPolicy* attribute), 215
- `batch_size` (*d3rlpy.algos.SAC* attribute), 56
- `batch_size` (*d3rlpy.algos.TD3* attribute), 43
- `batch_size` (*d3rlpy.algos.TD3PlusBC* attribute), 167
- `batch_size` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* attribute), 406
- `batch_size` (*d3rlpy.ope.DiscreteFQE* attribute), 385
- `batch_size` (*d3rlpy.ope.FQE* attribute), 374
- `BatchReplayBuffer` (class in *d3rlpy.online.buffers*), 398
- BC* (class in *d3rlpy.algos*), 10
- BCQ* (class in *d3rlpy.algos*), 57
- BEAR* (class in *d3rlpy.algos*), 70
- `bootstrap` (*d3rlpy.models.q\_functions.FQFQFunctionFactory* attribute), 313

bootstrap(*d3rlpy.models.q\_functions.IQNQFunctionFactory* *method*), 205  
     *attribute*), 312  
 bootstrap(*d3rlpy.models.q\_functions.MeanQFunctionFactory* *method*), 46  
     *attribute*), 310  
 bootstrap(*d3rlpy.models.q\_functions.QRQFunctionFactory* *method*), 157  
     *attribute*), 311  
 build\_episodes() (*d3rlpy.dataset.MDPDataset* *method*), 316  
     (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* *method*), 401  
 build\_transitions() (*d3rlpy.dataset.Episode* *method*), 319  
 build\_with\_dataset() (*d3rlpy.algos.AWAC* *method*), 121  
 build\_with\_dataset() (*d3rlpy.algos.AWR* *method*), 109  
 build\_with\_dataset() (*d3rlpy.algos.BC* *method*), 10  
 build\_with\_dataset() (*d3rlpy.algos.BCQ* *method*), 59  
 build\_with\_dataset() (*d3rlpy.algos.BEAR* *method*), 72  
 build\_with\_dataset() (*d3rlpy.algos.COMBO* *method*), 194  
 build\_with\_dataset() (*d3rlpy.algos.CQL* *method*), 97  
 build\_with\_dataset() (*d3rlpy.algos.CRR* *method*), 85  
 build\_with\_dataset() (*d3rlpy.algos.DDPG* *method*), 22  
 build\_with\_dataset() (*d3rlpy.algos.DiscreteAWR* *method*), 287  
 build\_with\_dataset() (*d3rlpy.algos.DiscreteBC* *method*), 217  
 build\_with\_dataset() (*d3rlpy.algos.DiscreteBCQ* *method*), 263  
 build\_with\_dataset() (*d3rlpy.algos.DiscreteCQL* *method*), 275  
 build\_with\_dataset() (*d3rlpy.algos.DiscreteRandomPolicy* *method*), 297  
 build\_with\_dataset() (*d3rlpy.algos.DiscreteSAC* *method*), 252  
 build\_with\_dataset() (*d3rlpy.algos.DoubleDQN* *method*), 240  
 build\_with\_dataset() (*d3rlpy.algos.DQN* *method*), 228  
 build\_with\_dataset() (*d3rlpy.algos.IQL* *method*), 169  
 build\_with\_dataset() (*d3rlpy.algos.MOPO* *method*), 182  
 build\_with\_dataset() (*d3rlpy.algos.PLAS* *method*), 133  
 build\_with\_dataset() (*d3rlpy.algos.PLASWithPerturbation* *method*), 145  
 build\_with\_dataset() (*d3rlpy.algos.RandomPolicy* *method*), 205  
 build\_with\_dataset() (*d3rlpy.algos.SAC* *method*), 46  
 build\_with\_dataset() (*d3rlpy.algos.TD3* *method*), 34  
 build\_with\_dataset() (*d3rlpy.algos.TD3PlusBC* *method*), 157  
 build\_with\_dataset() (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* *method*), 401  
 build\_with\_dataset() (*d3rlpy.ope.DiscreteFQE* *method*), 376  
 build\_with\_dataset() (*d3rlpy.ope.FQE* *method*), 364  
 build\_with\_env() (*d3rlpy.algos.AWAC* *method*), 121  
 build\_with\_env() (*d3rlpy.algos.AWR* *method*), 109  
 build\_with\_env() (*d3rlpy.algos.BC* *method*), 10  
 build\_with\_env() (*d3rlpy.algos.BCQ* *method*), 59  
 build\_with\_env() (*d3rlpy.algos.BEAR* *method*), 72  
 build\_with\_env() (*d3rlpy.algos.COMBO* *method*), 194  
 build\_with\_env() (*d3rlpy.algos.CQL* *method*), 97  
 build\_with\_env() (*d3rlpy.algos.CRR* *method*), 85  
 build\_with\_env() (*d3rlpy.algos.DDPG* *method*), 22  
 build\_with\_env() (*d3rlpy.algos.DiscreteAWR* *method*), 287  
 build\_with\_env() (*d3rlpy.algos.DiscreteBC* *method*), 217  
 build\_with\_env() (*d3rlpy.algos.DiscreteBCQ* *method*), 263  
 build\_with\_env() (*d3rlpy.algos.DiscreteCQL* *method*), 275  
 build\_with\_env() (*d3rlpy.algos.DiscreteRandomPolicy* *method*), 297  
 build\_with\_env() (*d3rlpy.algos.DiscreteSAC* *method*), 252  
 build\_with\_env() (*d3rlpy.algos.DoubleDQN* *method*), 240  
 build\_with\_env() (*d3rlpy.algos.DQN* *method*), 228  
 build\_with\_env() (*d3rlpy.algos.IQL* *method*), 169  
 build\_with\_env() (*d3rlpy.algos.MOPO* *method*), 182  
 build\_with\_env() (*d3rlpy.algos.PLAS* *method*), 133  
 build\_with\_env() (*d3rlpy.algos.PLASWithPerturbation* *method*), 145  
 build\_with\_env() (*d3rlpy.algos.RandomPolicy* *method*), 205  
 build\_with\_env() (*d3rlpy.algos.SAC* *method*), 46  
 build\_with\_env() (*d3rlpy.algos.TD3* *method*), 34  
 build\_with\_env() (*d3rlpy.algos.TD3PlusBC* *method*), 157  
 build\_with\_env() (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* *method*), 401  
 build\_with\_env() (*d3rlpy.ope.DiscreteFQE* *method*), 376  
 build\_with\_env() (*d3rlpy.ope.FQE* *method*), 364

## C

`clear_links()` (*d3rlpy.dataset.Transition* method), 321  
`clip_episode()` (*d3rlpy.online.buffers.BatchReplayBuffer* method), 398  
`clip_episode()` (*d3rlpy.online.buffers.ReplayBuffer* method), 394  
`ClipRewardScaler` (class in *d3rlpy.preprocessing*), 341  
`collect()` (*d3rlpy.algos.AWAC* method), 121  
`collect()` (*d3rlpy.algos.AWR* method), 109  
`collect()` (*d3rlpy.algos.BC* method), 10  
`collect()` (*d3rlpy.algos.BCQ* method), 59  
`collect()` (*d3rlpy.algos.BEAR* method), 72  
`collect()` (*d3rlpy.algos.COMBO* method), 194  
`collect()` (*d3rlpy.algos.CQL* method), 97  
`collect()` (*d3rlpy.algos.CRR* method), 85  
`collect()` (*d3rlpy.algos.DDPG* method), 22  
`collect()` (*d3rlpy.algos.DiscreteAWR* method), 287  
`collect()` (*d3rlpy.algos.DiscreteBC* method), 217  
`collect()` (*d3rlpy.algos.DiscreteBCQ* method), 264  
`collect()` (*d3rlpy.algos.DiscreteCQL* method), 275  
`collect()` (*d3rlpy.algos.DiscreteRandomPolicy* method), 298  
`collect()` (*d3rlpy.algos.DiscreteSAC* method), 252  
`collect()` (*d3rlpy.algos.DoubleDQN* method), 240  
`collect()` (*d3rlpy.algos.DQN* method), 228  
`collect()` (*d3rlpy.algos.IQL* method), 169  
`collect()` (*d3rlpy.algos.MOPO* method), 182  
`collect()` (*d3rlpy.algos.PLAS* method), 133  
`collect()` (*d3rlpy.algos.PLASWithPerturbation* method), 145  
`collect()` (*d3rlpy.algos.RandomPolicy* method), 205  
`collect()` (*d3rlpy.algos.SAC* method), 46  
`collect()` (*d3rlpy.algos.TD3* method), 34  
`collect()` (*d3rlpy.algos.TD3PlusBC* method), 157  
`collect()` (*d3rlpy.ope.DiscreteFQE* method), 376  
`collect()` (*d3rlpy.ope.FQE* method), 365  
`COMBO` (class in *d3rlpy.algos*), 193  
`compare_continuous_action_diff()` (in module *d3rlpy.metrics.comparer*), 360  
`compare_discrete_action_match()` (in module *d3rlpy.metrics.comparer*), 361  
`compute_epsilon()` (*d3rlpy.online.explorers.LinearDecayEpsilonGreedy* method), 396  
`compute_return()` (*d3rlpy.dataset.Episode* method), 319  
`compute_stats()` (*d3rlpy.dataset.MDPDataset* method), 316  
`ConstantEpsilonGreedy` (class in *d3rlpy.online.explorers*), 395  
`continuous_action_diff_scorer()` (in module *d3rlpy.metrics.scorer*), 359  
`copy_policy_from()` (*d3rlpy.algos.AWAC* method), 121  
`copy_policy_from()` (*d3rlpy.algos.AWR* method), 109  
`copy_policy_from()` (*d3rlpy.algos.BC* method), 11  
`copy_policy_from()` (*d3rlpy.algos.BCQ* method), 60  
`copy_policy_from()` (*d3rlpy.algos.BEAR* method), 72  
`copy_policy_from()` (*d3rlpy.algos.COMBO* method), 195  
`copy_policy_from()` (*d3rlpy.algos.CQL* method), 98  
`copy_policy_from()` (*d3rlpy.algos.CRR* method), 85  
`copy_policy_from()` (*d3rlpy.algos.DDPG* method), 22  
`copy_policy_from()` (*d3rlpy.algos.DiscreteAWR* method), 288  
`copy_policy_from()` (*d3rlpy.algos.DiscreteBC* method), 217  
`copy_policy_from()` (*d3rlpy.algos.DiscreteBCQ* method), 264  
`copy_policy_from()` (*d3rlpy.algos.DiscreteCQL* method), 276  
`copy_policy_from()` (*d3rlpy.algos.DiscreteRandomPolicy* method), 298  
`copy_policy_from()` (*d3rlpy.algos.DiscreteSAC* method), 252  
`copy_policy_from()` (*d3rlpy.algos.DoubleDQN* method), 240  
`copy_policy_from()` (*d3rlpy.algos.DQN* method), 229  
`copy_policy_from()` (*d3rlpy.algos.IQL* method), 170  
`copy_policy_from()` (*d3rlpy.algos.MOPO* method), 182  
`copy_policy_from()` (*d3rlpy.algos.PLAS* method), 133  
`copy_policy_from()` (*d3rlpy.algos.PLASWithPerturbation* method), 146  
`copy_policy_from()` (*d3rlpy.algos.RandomPolicy* method), 206  
`copy_policy_from()` (*d3rlpy.algos.SAC* method), 47  
`copy_policy_from()` (*d3rlpy.algos.TD3* method), 35  
`copy_policy_from()` (*d3rlpy.algos.TD3PlusBC* method), 158  
`copy_policy_from()` (*d3rlpy.ope.DiscreteFQE* method), 377  
`copy_policy_from()` (*d3rlpy.ope.FQE* method), 365  
`copy_policy_optim_from()` (*d3rlpy.algos.AWAC* method), 121  
`copy_policy_optim_from()` (*d3rlpy.algos.AWR* method), 110  
`copy_policy_optim_from()` (*d3rlpy.algos.BC* method), 11  
`copy_policy_optim_from()` (*d3rlpy.algos.BCQ* method), 60  
`copy_policy_optim_from()` (*d3rlpy.algos.BEAR* method), 73  
`copy_policy_optim_from()` (*d3rlpy.algos.COMBO* method), 195  
`copy_policy_optim_from()` (*d3rlpy.algos.CQL* method), 98  
`copy_policy_optim_from()` (*d3rlpy.algos.CRR* method), 85



<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.DDPG method</i> ), 23	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.CRR method</i> ), 86
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.DiscreteAWR method</i> ), 288	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DDPG method</i> ), 23
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.DiscreteBC method</i> ), 218	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DiscreteAWR method</i> ), 288
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.DiscreteBCQ method</i> ), 264	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DiscreteBC method</i> ), 218
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.DiscreteCQL method</i> ), 276	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DiscreteBCQ method</i> ), 265
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.DiscreteRandomPolicy method</i> ), 298	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DiscreteCQL method</i> ), 276
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.DiscreteSAC method</i> ), 252	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DiscreteRandomPolicy method</i> ), 299
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.DoubleDQN method</i> ), 240	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DiscreteSAC method</i> ), 253
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.DQN method</i> ), 229	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DoubleDQN method</i> ), 241
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.IQL method</i> ), 170	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.DQN method</i> ), 229
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.MOPO method</i> ), 183	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.IQL method</i> ), 170
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.PLAS method</i> ), 134	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.MOPO method</i> ), 183
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.PLASWithPerturbation method</i> ), 146	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.PLAS method</i> ), 134
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.RandomPolicy method</i> ), 206	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.PLASWithPerturbation method</i> ), 146
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.SAC method</i> ), 47	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.RandomPolicy method</i> ), 206
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.TD3 method</i> ), 35	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.SAC method</i> ), 47
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.algos.TD3PlusBC method</i> ), 158	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.TD3 method</i> ), 35
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.ope.DiscreteFQE method</i> ), 377	<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.TD3PlusBC method</i> ), 158
<code>copy_policy_optim_from()</code> ( <i>d3rlpy.ope.FQE method</i> ), 365	<code>copy_q_function_from()</code> ( <i>d3rlpy.ope.DiscreteFQE method</i> ), 377
<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.AWAC method</i> ), 122	<code>copy_q_function_from()</code> ( <i>d3rlpy.ope.FQE method</i> ), 365
<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.AWR method</i> ), 110	<code>copy_q_function_optim_from()</code> ( <i>d3rlpy.algos.AWAC method</i> ), 122
<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.BC method</i> ), 11	<code>copy_q_function_optim_from()</code> ( <i>d3rlpy.algos.AWR method</i> ), 110
<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.BCQ method</i> ), 60	<code>copy_q_function_optim_from()</code> ( <i>d3rlpy.algos.BC method</i> ), 12
<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.BEAR method</i> ), 73	<code>copy_q_function_optim_from()</code> ( <i>d3rlpy.algos.BCQ method</i> ), 60
<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.COMBO method</i> ), 195	<code>copy_q_function_optim_from()</code> ( <i>d3rlpy.algos.BEAR method</i> ), 73
<code>copy_q_function_from()</code> ( <i>d3rlpy.algos.CQL method</i> ), 98	<code>copy_q_function_optim_from()</code> ( <i>d3rlpy.algos.COMBO method</i> ), 196

`copy_q_function_optim_from()` (*d3rlpy.algos.CQL method*), 98  
`copy_q_function_optim_from()` (*d3rlpy.algos.CRR method*), 86  
`copy_q_function_optim_from()` (*d3rlpy.algos.DDPG method*), 23  
`copy_q_function_optim_from()` (*d3rlpy.algos.DiscreteAWR method*), 288  
`copy_q_function_optim_from()` (*d3rlpy.algos.DiscreteBC method*), 218  
`copy_q_function_optim_from()` (*d3rlpy.algos.DiscreteBCQ method*), 265  
`copy_q_function_optim_from()` (*d3rlpy.algos.DiscreteCQL method*), 276  
`copy_q_function_optim_from()` (*d3rlpy.algos.DiscreteRandomPolicy method*), 299  
`copy_q_function_optim_from()` (*d3rlpy.algos.DiscreteSAC method*), 253  
`copy_q_function_optim_from()` (*d3rlpy.algos.DoubleDQN method*), 241  
`copy_q_function_optim_from()` (*d3rlpy.algos.DQN method*), 229  
`copy_q_function_optim_from()` (*d3rlpy.algos.IQL method*), 171  
`copy_q_function_optim_from()` (*d3rlpy.algos.MOPO method*), 183  
`copy_q_function_optim_from()` (*d3rlpy.algos.PLAS method*), 134  
`copy_q_function_optim_from()` (*d3rlpy.algos.PLASWithPerturbation method*), 147  
`copy_q_function_optim_from()` (*d3rlpy.algos.RandomPolicy method*), 207  
`copy_q_function_optim_from()` (*d3rlpy.algos.SAC method*), 48  
`copy_q_function_optim_from()` (*d3rlpy.algos.TD3 method*), 35  
`copy_q_function_optim_from()` (*d3rlpy.algos.TD3PlusBC method*), 159  
`copy_q_function_optim_from()` (*d3rlpy.ope.DiscreteFQE method*), 377  
`copy_q_function_optim_from()` (*d3rlpy.ope.FQE method*), 366  
`CQL` (class in *d3rlpy.algos*), 95  
`create()` (*d3rlpy.models.encoders.DefaultEncoderFactory method*), 351  
`create()` (*d3rlpy.models.encoders.DenseEncoderFactory method*), 354  
`create()` (*d3rlpy.models.encoders.PixelEncoderFactory method*), 352  
`create()` (*d3rlpy.models.encoders.VectorEncoderFactory method*), 353  
`create()` (*d3rlpy.models.optimizers.AdamFactory method*), 347  
`create()` (*d3rlpy.models.optimizers.OptimizerFactory method*), 345  
`create()` (*d3rlpy.models.optimizers.RMSpropFactory method*), 348  
`create()` (*d3rlpy.models.optimizers.SGDFactory method*), 346  
`create_continuous()` (*d3rlpy.models.q\_functions.FQFQFunctionFactory method*), 313  
`create_continuous()` (*d3rlpy.models.q\_functions.IQNQFunctionFactory method*), 311  
`create_continuous()` (*d3rlpy.models.q\_functions.MeanQFunctionFactory method*), 309  
`create_continuous()` (*d3rlpy.models.q\_functions.QRQFunctionFactory method*), 310  
`create_discrete()` (*d3rlpy.models.q\_functions.FQFQFunctionFactory method*), 313  
`create_discrete()` (*d3rlpy.models.q\_functions.IQNQFunctionFactory method*), 311  
`create_discrete()` (*d3rlpy.models.q\_functions.MeanQFunctionFactory method*), 309  
`create_discrete()` (*d3rlpy.models.q\_functions.QRQFunctionFactory method*), 310  
`create_impl()` (*d3rlpy.algos.AWAC method*), 122  
`create_impl()` (*d3rlpy.algos.AWR method*), 111  
`create_impl()` (*d3rlpy.algos.BC method*), 12  
`create_impl()` (*d3rlpy.algos.BCQ method*), 61  
`create_impl()` (*d3rlpy.algos.BEAR method*), 74  
`create_impl()` (*d3rlpy.algos.COMBO method*), 196  
`create_impl()` (*d3rlpy.algos.CQL method*), 99  
`create_impl()` (*d3rlpy.algos.CRR method*), 86  
`create_impl()` (*d3rlpy.algos.DDPG method*), 24  
`create_impl()` (*d3rlpy.algos.DiscreteAWR method*), 289  
`create_impl()` (*d3rlpy.algos.DiscreteBC method*), 219  
`create_impl()` (*d3rlpy.algos.DiscreteBCQ method*), 265  
`create_impl()` (*d3rlpy.algos.DiscreteCQL method*), 277  
`create_impl()` (*d3rlpy.algos.DiscreteRandomPolicy method*), 299  
`create_impl()` (*d3rlpy.algos.DiscreteSAC method*), 253  
`create_impl()` (*d3rlpy.algos.DoubleDQN method*), 241  
`create_impl()` (*d3rlpy.algos.DQN method*), 230  
`create_impl()` (*d3rlpy.algos.IQL method*), 171  
`create_impl()` (*d3rlpy.algos.MOPO method*), 184  
`create_impl()` (*d3rlpy.algos.PLAS method*), 134  
`create_impl()` (*d3rlpy.algos.PLASWithPerturbation method*), 147

- method*), 147
- `create_impl()` (*d3rlpy.algos.RandomPolicy method*), 207
- `create_impl()` (*d3rlpy.algos.SAC method*), 48
- `create_impl()` (*d3rlpy.algos.TD3 method*), 36
- `create_impl()` (*d3rlpy.algos.TD3PlusBC method*), 159
- `create_impl()` (*d3rlpy.dynamics.ProbabilisticEnsembleDP method*), 402
- `create_impl()` (*d3rlpy.ope.DiscreteFQE method*), 378
- `create_impl()` (*d3rlpy.ope.FQE method*), 366
- `create_with_action()` (*d3rlpy.models.encoders.DefaultEncoderFactory method*), 351
- `create_with_action()` (*d3rlpy.models.encoders.DenseEncoderFactory method*), 354
- `create_with_action()` (*d3rlpy.models.encoders.PixelEncoderFactory method*), 352
- `create_with_action()` (*d3rlpy.models.encoders.VectorEncoderFactory method*), 353
- CRR (*class in d3rlpy.algos*), 83
- D**
- d3rlpy
  - module, 9
- d3rlpy.algos
  - module, 9
- d3rlpy.dataset
  - module, 314
- d3rlpy.datasets
  - module, 325
- d3rlpy.dynamics
  - module, 399
- d3rlpy.metrics
  - module, 355
- d3rlpy.models.encoders
  - module, 348
- d3rlpy.models.optimizers
  - module, 344
- d3rlpy.models.q\_functions
  - module, 308
- d3rlpy.online
  - module, 392
- d3rlpy.ope
  - module, 363
- d3rlpy.preprocessing
  - module, 329
- DDPG (*class in d3rlpy.algos*), 21
- DefaultEncoderFactory (*class in d3rlpy.models.encoders*), 350
- DenseEncoderFactory (*class in d3rlpy.models.encoders*), 354
- discounted\_sum\_of\_advantage\_scorer() (*in module d3rlpy.metrics.scorer*), 356
- discrete\_action\_match\_scorer() (*in module d3rlpy.metrics.scorer*), 359
- DiscreteAWR (*class in d3rlpy.algos*), 286
- DiscreteBC (*class in d3rlpy.algos*), 216
- DiscreteBCQ (*class in d3rlpy.algos*), 262
- DiscreteCQL (*class in d3rlpy.algos*), 274
- DiscreteFQE (*class in d3rlpy.ope*), 375
- DiscreteRandomPolicy (*class in d3rlpy.algos*), 297
- DiscreteSAC (*class in d3rlpy.algos*), 250
- DoubleDQN (*class in d3rlpy.algos*), 239
- DQN (*class in d3rlpy.algos*), 227
- dump() (*d3rlpy.dataset.MDPDataset method*), 317
- dynamics\_observation\_prediction\_error\_scorer() (*in module d3rlpy.metrics.scorer*), 362
- dynamics\_prediction\_variance\_scorer() (*in module d3rlpy.metrics.scorer*), 362
- dynamics\_reward\_prediction\_error\_scorer() (*in module d3rlpy.metrics.scorer*), 362
- E**
- embed\_size (*d3rlpy.models.q\_functions.FQFQFunctionFactory attribute*), 313
- embed\_size (*d3rlpy.models.q\_functions.IQNQFunctionFactory attribute*), 312
- entropy\_coeff (*d3rlpy.models.q\_functions.FQFQFunctionFactory attribute*), 313
- Episode (*class in d3rlpy.dataset*), 319
- episode\_terminals (*d3rlpy.dataset.MDPDataset attribute*), 318
- episodes (*d3rlpy.dataset.MDPDataset attribute*), 318
- evaluate\_on\_environment() (*in module d3rlpy.metrics.scorer*), 360
- extend() (*d3rlpy.dataset.MDPDataset method*), 317
- F**
- fit() (*d3rlpy.algos.AWAC method*), 123
- fit() (*d3rlpy.algos.AWR method*), 111
- fit() (*d3rlpy.algos.BC method*), 12
- fit() (*d3rlpy.algos.BCQ method*), 61
- fit() (*d3rlpy.algos.BEAR method*), 74
- fit() (*d3rlpy.algos.COMBO method*), 196
- fit() (*d3rlpy.algos.CQL method*), 99
- fit() (*d3rlpy.algos.CRR method*), 87
- fit() (*d3rlpy.algos.DDPG method*), 24
- fit() (*d3rlpy.algos.DiscreteAWR method*), 289
- fit() (*d3rlpy.algos.DiscreteBC method*), 219
- fit() (*d3rlpy.algos.DiscreteBCQ method*), 265
- fit() (*d3rlpy.algos.DiscreteCQL method*), 277
- in* fit() (*d3rlpy.algos.DiscreteRandomPolicy method*), 299
- in* fit() (*d3rlpy.algos.DiscreteSAC method*), 254
- fit() (*d3rlpy.algos.DoubleDQN method*), 242
- fit() (*d3rlpy.algos.DQN method*), 230

- [fit\(\)](#) ([d3rlpy.algos.IQL method](#)), 171  
[fit\(\)](#) ([d3rlpy.algos.MOPO method](#)), 184  
[fit\(\)](#) ([d3rlpy.algos.PLAS method](#)), 135  
[fit\(\)](#) ([d3rlpy.algos.PLASWithPerturbation method](#)), 147  
[fit\(\)](#) ([d3rlpy.algos.RandomPolicy method](#)), 207  
[fit\(\)](#) ([d3rlpy.algos.SAC method](#)), 48  
[fit\(\)](#) ([d3rlpy.algos.TD3 method](#)), 36  
[fit\(\)](#) ([d3rlpy.algos.TD3PlusBC method](#)), 159  
[fit\(\)](#) ([d3rlpy.dynamics.ProbabilisticEnsembleDynamics method](#)), 402  
[fit\(\)](#) ([d3rlpy.ope.DiscreteFQE method](#)), 378  
[fit\(\)](#) ([d3rlpy.ope.FQE method](#)), 366  
[fit\(\)](#) ([d3rlpy.preprocessing.ClipRewardScaler method](#)), 341  
[fit\(\)](#) ([d3rlpy.preprocessing.MinMaxActionScaler method](#)), 336  
[fit\(\)](#) ([d3rlpy.preprocessing.MinMaxRewardScaler method](#)), 338  
[fit\(\)](#) ([d3rlpy.preprocessing.MinMaxScaler method](#)), 332  
[fit\(\)](#) ([d3rlpy.preprocessing.MultiplyRewardScaler method](#)), 343  
[fit\(\)](#) ([d3rlpy.preprocessing.PixelScaler method](#)), 330  
[fit\(\)](#) ([d3rlpy.preprocessing.StandardRewardScaler method](#)), 340  
[fit\(\)](#) ([d3rlpy.preprocessing.StandardScaler method](#)), 333  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.AWAC method](#)), 123  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.AWR method](#)), 112  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.BC method](#)), 13  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.BCQ method](#)), 62  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.BEAR method](#)), 75  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.COMBO method](#)), 197  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.CQL method](#)), 100  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.CRR method](#)), 87  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.DDPG method](#)), 25  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.DiscreteAWR method](#)), 290  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.DiscreteBC method](#)), 220  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.DiscreteBCQ method](#)), 266  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.DiscreteCQL method](#)), 278  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.DiscreteRandomPolicy method](#)), 300  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.DiscreteSAC method](#)), 254  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.DoubleDQN method](#)), 242  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.DQN method](#)), 231  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.IQL method](#)), 172  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.MOPO method](#)), 185  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.PLAS method](#)), 135  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.PLASWithPerturbation method](#)), 148  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.RandomPolicy method](#)), 208  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.SAC method](#)), 49  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.TD3 method](#)), 37  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.algos.TD3PlusBC method](#)), 160  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.ope.DiscreteFQE method](#)), 379  
[fit\\_batch\\_online\(\)](#) ([d3rlpy.ope.FQE method](#)), 367  
[fit\\_online\(\)](#) ([d3rlpy.algos.AWAC method](#)), 124  
[fit\\_online\(\)](#) ([d3rlpy.algos.AWR method](#)), 113  
[fit\\_online\(\)](#) ([d3rlpy.algos.BC method](#)), 14  
[fit\\_online\(\)](#) ([d3rlpy.algos.BCQ method](#)), 63  
[fit\\_online\(\)](#) ([d3rlpy.algos.BEAR method](#)), 76  
[fit\\_online\(\)](#) ([d3rlpy.algos.COMBO method](#)), 198  
[fit\\_online\(\)](#) ([d3rlpy.algos.CQL method](#)), 101  
[fit\\_online\(\)](#) ([d3rlpy.algos.CRR method](#)), 88  
[fit\\_online\(\)](#) ([d3rlpy.algos.DDPG method](#)), 26  
[fit\\_online\(\)](#) ([d3rlpy.algos.DiscreteAWR method](#)), 291  
[fit\\_online\(\)](#) ([d3rlpy.algos.DiscreteBC method](#)), 221  
[fit\\_online\(\)](#) ([d3rlpy.algos.DiscreteBCQ method](#)), 267  
[fit\\_online\(\)](#) ([d3rlpy.algos.DiscreteCQL method](#)), 279  
[fit\\_online\(\)](#) ([d3rlpy.algos.DiscreteRandomPolicy method](#)), 301  
[fit\\_online\(\)](#) ([d3rlpy.algos.DiscreteSAC method](#)), 255  
[fit\\_online\(\)](#) ([d3rlpy.algos.DoubleDQN method](#)), 243  
[fit\\_online\(\)](#) ([d3rlpy.algos.DQN method](#)), 232  
[fit\\_online\(\)](#) ([d3rlpy.algos.IQL method](#)), 173  
[fit\\_online\(\)](#) ([d3rlpy.algos.MOPO method](#)), 185  
[fit\\_online\(\)](#) ([d3rlpy.algos.PLAS method](#)), 136  
[fit\\_online\(\)](#) ([d3rlpy.algos.PLASWithPerturbation method](#)), 149  
[fit\\_online\(\)](#) ([d3rlpy.algos.RandomPolicy method](#)), 209  
[fit\\_online\(\)](#) ([d3rlpy.algos.SAC method](#)), 50  
[fit\\_online\(\)](#) ([d3rlpy.algos.TD3 method](#)), 38  
[fit\\_online\(\)](#) ([d3rlpy.algos.TD3PlusBC method](#)), 161  
[fit\\_online\(\)](#) ([d3rlpy.ope.DiscreteFQE method](#)), 380  
[fit\\_online\(\)](#) ([d3rlpy.ope.FQE method](#)), 368  
[fit\\_with\\_env\(\)](#) ([d3rlpy.preprocessing.ClipRewardScaler method](#)), 341  
[fit\\_with\\_env\(\)](#) ([d3rlpy.preprocessing.MinMaxActionScaler method](#)), 336  
[fit\\_with\\_env\(\)](#) ([d3rlpy.preprocessing.MinMaxRewardScaler method](#)), 338  
[fit\\_with\\_env\(\)](#) ([d3rlpy.preprocessing.MinMaxScaler method](#)), 332



- `fit_with_env()` (`d3rlpy.preprocessing.MultiplyRewardScaler` class method), 343  
`fit_with_env()` (`d3rlpy.preprocessing.PixelScaler` class method), 330  
`fit_with_env()` (`d3rlpy.preprocessing.StandardRewardScaler` class method), 340  
`fit_with_env()` (`d3rlpy.preprocessing.StandardScaler` class method), 333  
`fitter()` (`d3rlpy.algos.AWAC` method), 125  
`fitter()` (`d3rlpy.algos.AWR` method), 114  
`fitter()` (`d3rlpy.algos.BC` method), 15  
`fitter()` (`d3rlpy.algos.BCQ` method), 64  
`fitter()` (`d3rlpy.algos.BEAR` method), 76  
`fitter()` (`d3rlpy.algos.COMBO` method), 199  
`fitter()` (`d3rlpy.algos.CQL` method), 102  
`fitter()` (`d3rlpy.algos.CRR` method), 89  
`fitter()` (`d3rlpy.algos.DDPG` method), 27  
`fitter()` (`d3rlpy.algos.DiscreteAWR` method), 292  
`fitter()` (`d3rlpy.algos.DiscreteBC` method), 221  
`fitter()` (`d3rlpy.algos.DiscreteBCQ` method), 268  
`fitter()` (`d3rlpy.algos.DiscreteCQL` method), 280  
`fitter()` (`d3rlpy.algos.DiscreteRandomPolicy` method), 302  
`fitter()` (`d3rlpy.algos.DiscreteSAC` method), 256  
`fitter()` (`d3rlpy.algos.DoubleDQN` method), 244  
`fitter()` (`d3rlpy.algos.DQN` method), 233  
`fitter()` (`d3rlpy.algos.IQL` method), 174  
`fitter()` (`d3rlpy.algos.MOPO` method), 186  
`fitter()` (`d3rlpy.algos.PLAS` method), 137  
`fitter()` (`d3rlpy.algos.PLASWithPerturbation` method), 150  
`fitter()` (`d3rlpy.algos.RandomPolicy` method), 210  
`fitter()` (`d3rlpy.algos.SAC` method), 51  
`fitter()` (`d3rlpy.algos.TD3` method), 39  
`fitter()` (`d3rlpy.algos.TD3PlusBC` method), 162  
`fitter()` (`d3rlpy.dynamics.ProbabilisticEnsembleDynamics` class method), 403  
`fitter()` (`d3rlpy.ope.DiscreteFQE` method), 381  
`fitter()` (`d3rlpy.ope.FQE` method), 369  
`FQE` (class in `d3rlpy.ope`), 363  
`FQFQFunctionFactory` (class in `d3rlpy.models.q_functions`), 312  
`from_json()` (`d3rlpy.algos.AWAC` class method), 126  
`from_json()` (`d3rlpy.algos.AWR` class method), 114  
`from_json()` (`d3rlpy.algos.BC` class method), 16  
`from_json()` (`d3rlpy.algos.BCQ` class method), 65  
`from_json()` (`d3rlpy.algos.BEAR` class method), 77  
`from_json()` (`d3rlpy.algos.COMBO` class method), 200  
`from_json()` (`d3rlpy.algos.CQL` class method), 102  
`from_json()` (`d3rlpy.algos.CRR` class method), 90  
`from_json()` (`d3rlpy.algos.DDPG` class method), 27  
`from_json()` (`d3rlpy.algos.DiscreteAWR` class method), 292  
`from_json()` (`d3rlpy.algos.DiscreteBC` class method), 222  
`from_json()` (`d3rlpy.algos.DiscreteBCQ` class method), 269  
`from_json()` (`d3rlpy.algos.DiscreteCQL` class method), 280  
`from_json()` (`d3rlpy.algos.DiscreteRandomPolicy` class method), 303  
`from_json()` (`d3rlpy.algos.DiscreteSAC` class method), 257  
`from_json()` (`d3rlpy.algos.DoubleDQN` class method), 245  
`from_json()` (`d3rlpy.algos.DQN` class method), 233  
`from_json()` (`d3rlpy.algos.IQL` class method), 175  
`from_json()` (`d3rlpy.algos.MOPO` class method), 187  
`from_json()` (`d3rlpy.algos.PLAS` class method), 138  
`from_json()` (`d3rlpy.algos.PLASWithPerturbation` class method), 151  
`from_json()` (`d3rlpy.algos.RandomPolicy` class method), 211  
`from_json()` (`d3rlpy.algos.SAC` class method), 52  
`from_json()` (`d3rlpy.algos.TD3` class method), 40  
`from_json()` (`d3rlpy.algos.TD3PlusBC` class method), 163  
`from_json()` (`d3rlpy.dynamics.ProbabilisticEnsembleDynamics` class method), 404  
`from_json()` (`d3rlpy.ope.DiscreteFQE` class method), 382  
`from_json()` (`d3rlpy.ope.FQE` class method), 370
- ## G
- `gamma` (`d3rlpy.algos.AWAC` attribute), 130  
`gamma` (`d3rlpy.algos.AWR` attribute), 118  
`gamma` (`d3rlpy.algos.BC` attribute), 19  
`gamma` (`d3rlpy.algos.BCQ` attribute), 69  
`gamma` (`d3rlpy.algos.BEAR` attribute), 82  
`gamma` (`d3rlpy.algos.COMBO` attribute), 204  
`gamma` (`d3rlpy.algos.CQL` attribute), 107  
`gamma` (`d3rlpy.algos.CRR` attribute), 94  
`gamma` (`d3rlpy.algos.DDPG` attribute), 32  
`gamma` (`d3rlpy.algos.DiscreteAWR` attribute), 296  
`gamma` (`d3rlpy.algos.DiscreteBC` attribute), 226  
`gamma` (`d3rlpy.algos.DiscreteBCQ` attribute), 273  
`gamma` (`d3rlpy.algos.DiscreteCQL` attribute), 285  
`gamma` (`d3rlpy.algos.DiscreteRandomPolicy` attribute), 307  
`gamma` (`d3rlpy.algos.DiscreteSAC` attribute), 261  
`gamma` (`d3rlpy.algos.DoubleDQN` attribute), 249  
`gamma` (`d3rlpy.algos.DQN` attribute), 238  
`gamma` (`d3rlpy.algos.IQL` attribute), 179  
`gamma` (`d3rlpy.algos.MOPO` attribute), 191  
`gamma` (`d3rlpy.algos.PLAS` attribute), 142  
`gamma` (`d3rlpy.algos.PLASWithPerturbation` attribute), 155

- `gamma` (*d3rlpy.algos.RandomPolicy* attribute), 215
- `gamma` (*d3rlpy.algos.SAC* attribute), 56
- `gamma` (*d3rlpy.algos.TD3* attribute), 44
- `gamma` (*d3rlpy.algos.TD3PlusBC* attribute), 167
- `gamma` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* attribute), 406
- `gamma` (*d3rlpy.ope.DiscreteFQE* attribute), 386
- `gamma` (*d3rlpy.ope.FQE* attribute), 374
- `generate_new_data()` (*d3rlpy.algos.AWAC* method), 127
- `generate_new_data()` (*d3rlpy.algos.AWR* method), 115
- `generate_new_data()` (*d3rlpy.algos.BC* method), 16
- `generate_new_data()` (*d3rlpy.algos.BCQ* method), 65
- `generate_new_data()` (*d3rlpy.algos.BEAR* method), 78
- `generate_new_data()` (*d3rlpy.algos.COMBO* method), 200
- `generate_new_data()` (*d3rlpy.algos.CQL* method), 103
- `generate_new_data()` (*d3rlpy.algos.CRR* method), 91
- `generate_new_data()` (*d3rlpy.algos.DDPG* method), 28
- `generate_new_data()` (*d3rlpy.algos.DiscreteAWR* method), 293
- `generate_new_data()` (*d3rlpy.algos.DiscreteBC* method), 223
- `generate_new_data()` (*d3rlpy.algos.DiscreteBCQ* method), 269
- `generate_new_data()` (*d3rlpy.algos.DiscreteCQL* method), 281
- `generate_new_data()` (*d3rlpy.algos.DiscreteRandomPolicy* method), 303
- `generate_new_data()` (*d3rlpy.algos.DiscreteSAC* method), 258
- `generate_new_data()` (*d3rlpy.algos.DoubleDQN* method), 246
- `generate_new_data()` (*d3rlpy.algos.DQN* method), 234
- `generate_new_data()` (*d3rlpy.algos.IQL* method), 175
- `generate_new_data()` (*d3rlpy.algos.MOPO* method), 188
- `generate_new_data()` (*d3rlpy.algos.PLAS* method), 139
- `generate_new_data()` (*d3rlpy.algos.PLASWithPerturbation* method), 151
- `generate_new_data()` (*d3rlpy.algos.RandomPolicy* method), 211
- `generate_new_data()` (*d3rlpy.algos.SAC* method), 52
- `generate_new_data()` (*d3rlpy.algos.TD3* method), 40
- `generate_new_data()` (*d3rlpy.algos.TD3PlusBC* method), 163
- `generate_new_data()` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* method), 404
- `generate_new_data()` (*d3rlpy.ope.DiscreteFQE* method), 382
- `generate_new_data()` (*d3rlpy.ope.FQE* method), 370
- `get_action_size()` (*d3rlpy.dataset.Episode* method), 320
- `get_action_size()` (*d3rlpy.dataset.MDPDataset* method), 317
- `get_action_size()` (*d3rlpy.dataset.Transition* method), 321
- `get_action_type()` (*d3rlpy.algos.AWAC* method), 127
- `get_action_type()` (*d3rlpy.algos.AWR* method), 115
- `get_action_type()` (*d3rlpy.algos.BC* method), 16
- `get_action_type()` (*d3rlpy.algos.BCQ* method), 65
- `get_action_type()` (*d3rlpy.algos.BEAR* method), 78
- `get_action_type()` (*d3rlpy.algos.COMBO* method), 200
- `get_action_type()` (*d3rlpy.algos.CQL* method), 103
- `get_action_type()` (*d3rlpy.algos.CRR* method), 91
- `get_action_type()` (*d3rlpy.algos.DDPG* method), 28
- `get_action_type()` (*d3rlpy.algos.DiscreteAWR* method), 293
- `get_action_type()` (*d3rlpy.algos.DiscreteBC* method), 223
- `get_action_type()` (*d3rlpy.algos.DiscreteBCQ* method), 269
- `get_action_type()` (*d3rlpy.algos.DiscreteCQL* method), 281
- `get_action_type()` (*d3rlpy.algos.DiscreteRandomPolicy* method), 303
- `get_action_type()` (*d3rlpy.algos.DiscreteSAC* method), 258
- `get_action_type()` (*d3rlpy.algos.DoubleDQN* method), 246
- `get_action_type()` (*d3rlpy.algos.DQN* method), 234
- `get_action_type()` (*d3rlpy.algos.IQL* method), 175
- `get_action_type()` (*d3rlpy.algos.MOPO* method), 188
- `get_action_type()` (*d3rlpy.algos.PLAS* method), 139
- `get_action_type()` (*d3rlpy.algos.PLASWithPerturbation* method), 151
- `get_action_type()` (*d3rlpy.algos.RandomPolicy* method), 211
- `get_action_type()` (*d3rlpy.algos.SAC* method), 52
- `get_action_type()` (*d3rlpy.algos.TD3* method), 40
- `get_action_type()` (*d3rlpy.algos.TD3PlusBC* method), 163
- `get_action_type()` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* method), 404
- `get_action_type()` (*d3rlpy.ope.DiscreteFQE* method), 382
- `get_action_type()` (*d3rlpy.ope.FQE* method), 370
- `get_additional_data()`

(*d3rlpy.dataset.TransitionMiniBatch* method), 324  
*get\_atari()* (in module *d3rlpy.datasets*), 327  
*get\_cartpole()* (in module *d3rlpy.datasets*), 325  
*get\_d4rl()* (in module *d3rlpy.datasets*), 327  
*get\_dataset()* (in module *d3rlpy.datasets*), 328  
*get\_observation\_shape()* (*d3rlpy.dataset.Episode* method), 320  
*get\_observation\_shape()* (*d3rlpy.dataset.MDPDataset* method), 317  
*get\_observation\_shape()* (*d3rlpy.dataset.Transition* method), 321  
*get\_params()* (*d3rlpy.algos.AWAC* method), 127  
*get\_params()* (*d3rlpy.algos.AWR* method), 115  
*get\_params()* (*d3rlpy.algos.BC* method), 17  
*get\_params()* (*d3rlpy.algos.BCQ* method), 65  
*get\_params()* (*d3rlpy.algos.BEAR* method), 78  
*get\_params()* (*d3rlpy.algos.COMBO* method), 201  
*get\_params()* (*d3rlpy.algos.CQL* method), 103  
*get\_params()* (*d3rlpy.algos.CRR* method), 91  
*get\_params()* (*d3rlpy.algos.DDPG* method), 28  
*get\_params()* (*d3rlpy.algos.DiscreteAWR* method), 293  
*get\_params()* (*d3rlpy.algos.DiscreteBC* method), 223  
*get\_params()* (*d3rlpy.algos.DiscreteBCQ* method), 270  
*get\_params()* (*d3rlpy.algos.DiscreteCQL* method), 281  
*get\_params()* (*d3rlpy.algos.DiscreteRandomPolicy* method), 304  
*get\_params()* (*d3rlpy.algos.DiscreteSAC* method), 258  
*get\_params()* (*d3rlpy.algos.DoubleDQN* method), 246  
*get\_params()* (*d3rlpy.algos.DQN* method), 234  
*get\_params()* (*d3rlpy.algos.IQL* method), 176  
*get\_params()* (*d3rlpy.algos.MOPO* method), 188  
*get\_params()* (*d3rlpy.algos.PLAS* method), 139  
*get\_params()* (*d3rlpy.algos.PLASWithPerturbation* method), 151  
*get\_params()* (*d3rlpy.algos.RandomPolicy* method), 212  
*get\_params()* (*d3rlpy.algos.SAC* method), 52  
*get\_params()* (*d3rlpy.algos.TD3* method), 40  
*get\_params()* (*d3rlpy.algos.TD3PlusBC* method), 163  
*get\_params()* (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* method), 404  
*get\_params()* (*d3rlpy.models.encoders.DefaultEncoderFactory* method), 351  
*get\_params()* (*d3rlpy.models.encoders.DenseEncoderFactory* method), 354  
*get\_params()* (*d3rlpy.models.encoders.PixelEncoderFactory* method), 352  
*get\_params()* (*d3rlpy.models.encoders.VectorEncoderFactory* method), 353  
*get\_params()* (*d3rlpy.models.optimizers.AdamFactory* method), 347  
*get\_params()* (*d3rlpy.models.optimizers.OptimizerFactory* method), 345  
*get\_params()* (*d3rlpy.models.optimizers.RMSpropFactory* method), 348  
*get\_params()* (*d3rlpy.models.optimizers.SGDFactory* method), 346  
*get\_params()* (*d3rlpy.models.q\_functions.FQFQFunctionFactory* method), 313  
*get\_params()* (*d3rlpy.models.q\_functions.IQNQFunctionFactory* method), 312  
*get\_params()* (*d3rlpy.models.q\_functions.MeanQFunctionFactory* method), 309  
*get\_params()* (*d3rlpy.models.q\_functions.QRQFunctionFactory* method), 310  
*get\_params()* (*d3rlpy.ope.DiscreteFQE* method), 382  
*get\_params()* (*d3rlpy.ope.FQE* method), 371  
*get\_params()* (*d3rlpy.preprocessing.ClipRewardScaler* method), 342  
*get\_params()* (*d3rlpy.preprocessing.MinMaxActionScaler* method), 336  
*get\_params()* (*d3rlpy.preprocessing.MinMaxRewardScaler* method), 338  
*get\_params()* (*d3rlpy.preprocessing.MinMaxScaler* method), 332  
*get\_params()* (*d3rlpy.preprocessing.MultiplyRewardScaler* method), 343  
*get\_params()* (*d3rlpy.preprocessing.PixelScaler* method), 330  
*get\_params()* (*d3rlpy.preprocessing.StandardRewardScaler* method), 340  
*get\_params()* (*d3rlpy.preprocessing.StandardScaler* method), 334  
*get\_pendulum()* (in module *d3rlpy.datasets*), 326  
*get\_pybullet()* (in module *d3rlpy.datasets*), 326  
*get\_type()* (*d3rlpy.models.encoders.DefaultEncoderFactory* method), 351  
*get\_type()* (*d3rlpy.models.encoders.DenseEncoderFactory* method), 354  
*get\_type()* (*d3rlpy.models.encoders.PixelEncoderFactory* method), 352  
*get\_type()* (*d3rlpy.models.encoders.VectorEncoderFactory* method), 353  
*get\_type()* (*d3rlpy.models.q\_functions.FQFQFunctionFactory* method), 313  
*get\_type()* (*d3rlpy.models.q\_functions.IQNQFunctionFactory* method), 312  
*get\_type()* (*d3rlpy.models.q\_functions.MeanQFunctionFactory* method), 309  
*get\_type()* (*d3rlpy.models.q\_functions.QRQFunctionFactory* method), 310  
*get\_type()* (*d3rlpy.preprocessing.ClipRewardScaler* method), 342  
*get\_type()* (*d3rlpy.preprocessing.MinMaxActionScaler* method), 336  
*get\_type()* (*d3rlpy.preprocessing.MinMaxRewardScaler* method), 339



`get_type()` (*d3rlpy.preprocessing.MinMaxScaler method*), 332  
`get_type()` (*d3rlpy.preprocessing.MultiplyRewardScaler method*), 343  
`get_type()` (*d3rlpy.preprocessing.PixelScaler method*), 330  
`get_type()` (*d3rlpy.preprocessing.StandardRewardScaler method*), 340  
`get_type()` (*d3rlpy.preprocessing.StandardScaler method*), 334  
`grad_step` (*d3rlpy.algos.AWAC attribute*), 130  
`grad_step` (*d3rlpy.algos.AWR attribute*), 118  
`grad_step` (*d3rlpy.algos.BC attribute*), 20  
`grad_step` (*d3rlpy.algos.BCQ attribute*), 69  
`grad_step` (*d3rlpy.algos.BEAR attribute*), 82  
`grad_step` (*d3rlpy.algos.COMBO attribute*), 204  
`grad_step` (*d3rlpy.algos.CQL attribute*), 107  
`grad_step` (*d3rlpy.algos.CRR attribute*), 94  
`grad_step` (*d3rlpy.algos.DDPG attribute*), 32  
`grad_step` (*d3rlpy.algos.DiscreteAWR attribute*), 296  
`grad_step` (*d3rlpy.algos.DiscreteBC attribute*), 226  
`grad_step` (*d3rlpy.algos.DiscreteBCQ attribute*), 273  
`grad_step` (*d3rlpy.algos.DiscreteCQL attribute*), 285  
`grad_step` (*d3rlpy.algos.DiscreteRandomPolicy attribute*), 307  
`grad_step` (*d3rlpy.algos.DiscreteSAC attribute*), 261  
`grad_step` (*d3rlpy.algos.DoubleDQN attribute*), 249  
`grad_step` (*d3rlpy.algos.DQN attribute*), 238  
`grad_step` (*d3rlpy.algos.IQL attribute*), 179  
`grad_step` (*d3rlpy.algos.MOPO attribute*), 192  
`grad_step` (*d3rlpy.algos.PLAS attribute*), 142  
`grad_step` (*d3rlpy.algos.PLASWithPerturbation attribute*), 155  
`grad_step` (*d3rlpy.algos.RandomPolicy attribute*), 215  
`grad_step` (*d3rlpy.algos.SAC attribute*), 56  
`grad_step` (*d3rlpy.algos.TD3 attribute*), 44  
`grad_step` (*d3rlpy.algos.TD3PlusBC attribute*), 167  
`grad_step` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics attribute*), 406  
`grad_step` (*d3rlpy.ope.DiscreteFQE attribute*), 386  
`grad_step` (*d3rlpy.ope.FQE attribute*), 374  
  
**I**  
`impl` (*d3rlpy.algos.AWAC attribute*), 130  
`impl` (*d3rlpy.algos.AWR attribute*), 118  
`impl` (*d3rlpy.algos.BC attribute*), 20  
`impl` (*d3rlpy.algos.BCQ attribute*), 69  
`impl` (*d3rlpy.algos.BEAR attribute*), 82  
`impl` (*d3rlpy.algos.COMBO attribute*), 204  
`impl` (*d3rlpy.algos.CQL attribute*), 107  
`impl` (*d3rlpy.algos.CRR attribute*), 94  
`impl` (*d3rlpy.algos.DDPG attribute*), 32  
`impl` (*d3rlpy.algos.DiscreteAWR attribute*), 296  
`impl` (*d3rlpy.algos.DiscreteBC attribute*), 226  
`impl` (*d3rlpy.algos.DiscreteBCQ attribute*), 273  
`impl` (*d3rlpy.algos.DiscreteCQL attribute*), 285  
`impl` (*d3rlpy.algos.DiscreteRandomPolicy attribute*), 307  
`impl` (*d3rlpy.algos.DiscreteSAC attribute*), 261  
`impl` (*d3rlpy.algos.DoubleDQN attribute*), 249  
`impl` (*d3rlpy.algos.DQN attribute*), 238  
`impl` (*d3rlpy.algos.IQL attribute*), 179  
`impl` (*d3rlpy.algos.MOPO attribute*), 192  
`impl` (*d3rlpy.algos.PLAS attribute*), 143  
`impl` (*d3rlpy.algos.PLASWithPerturbation attribute*), 155  
`impl` (*d3rlpy.algos.RandomPolicy attribute*), 215  
`impl` (*d3rlpy.algos.SAC attribute*), 56  
`impl` (*d3rlpy.algos.TD3 attribute*), 44  
`impl` (*d3rlpy.algos.TD3PlusBC attribute*), 167  
`impl` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics attribute*), 407  
`impl` (*d3rlpy.ope.DiscreteFQE attribute*), 386  
`impl` (*d3rlpy.ope.FQE attribute*), 374  
`initial_state_value_estimation_scorer()` (*in module d3rlpy.metrics.scorer*), 358  
IQL (*class in d3rlpy.algos*), 168  
IQNQFunctionFactory (*class in d3rlpy.models.q\_functions*), 311  
`is_action_discrete()` (*d3rlpy.dataset.MDPDataset method*), 317  
`is_discrete` (*d3rlpy.dataset.Transition attribute*), 322  
  
**L**  
LinearDecayEpsilonGreedy (*class in d3rlpy.online.explorers*), 395  
`load()` (*d3rlpy.dataset.MDPDataset class method*), 317  
`load_model()` (*d3rlpy.algos.AWAC method*), 127  
`load_model()` (*d3rlpy.algos.AWR method*), 115  
`load_model()` (*d3rlpy.algos.BC method*), 17  
`load_model()` (*d3rlpy.algos.BCQ method*), 66  
`load_model()` (*d3rlpy.algos.BEAR method*), 78  
`load_model()` (*d3rlpy.algos.COMBO method*), 201  
`load_model()` (*d3rlpy.algos.CQL method*), 103  
`load_model()` (*d3rlpy.algos.CRR method*), 91  
`load_model()` (*d3rlpy.algos.DDPG method*), 28  
`load_model()` (*d3rlpy.algos.DiscreteAWR method*), 293  
`load_model()` (*d3rlpy.algos.DiscreteBC method*), 223  
`load_model()` (*d3rlpy.algos.DiscreteBCQ method*), 270  
`load_model()` (*d3rlpy.algos.DiscreteCQL method*), 281  
`load_model()` (*d3rlpy.algos.DiscreteRandomPolicy method*), 304  
`load_model()` (*d3rlpy.algos.DiscreteSAC method*), 258  
`load_model()` (*d3rlpy.algos.DoubleDQN method*), 246  
`load_model()` (*d3rlpy.algos.DQN method*), 234  
`load_model()` (*d3rlpy.algos.IQL method*), 176  
`load_model()` (*d3rlpy.algos.MOPO method*), 188  
`load_model()` (*d3rlpy.algos.PLAS method*), 139  
`load_model()` (*d3rlpy.algos.PLASWithPerturbation method*), 152

- load\_model() (*d3rlpy.algos.RandomPolicy* method), 212
- load\_model() (*d3rlpy.algos.SAC* method), 53
- load\_model() (*d3rlpy.algos.TD3* method), 41
- load\_model() (*d3rlpy.algos.TD3PlusBC* method), 164
- load\_model() (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* method), 405
- load\_model() (*d3rlpy.ope.DiscreteFQE* method), 383
- load\_model() (*d3rlpy.ope.FQE* method), 371
- ## M
- mask (*d3rlpy.dataset.Transition* attribute), 322
- masks (*d3rlpy.dataset.TransitionMiniBatch* attribute), 324
- MDPDataSet (class in *d3rlpy.dataset*), 315
- MeanQFunctionFactory (class in *d3rlpy.models.q\_functions*), 309
- MinMaxActionScaler (class in *d3rlpy.preprocessing*), 335
- MinMaxRewardScaler (class in *d3rlpy.preprocessing*), 338
- MinMaxScaler (class in *d3rlpy.preprocessing*), 331
- module
- d3rlpy, 9
  - d3rlpy.algos, 9
  - d3rlpy.dataset, 314
  - d3rlpy.datasets, 325
  - d3rlpy.dynamics, 399
  - d3rlpy.metrics, 355
  - d3rlpy.models.encoders, 348
  - d3rlpy.models.optimizers, 344
  - d3rlpy.models.q\_functions, 308
  - d3rlpy.online, 392
  - d3rlpy.ope, 363
  - d3rlpy.preprocessing, 329
- MOPO (class in *d3rlpy.algos*), 180
- MultiplyRewardScaler (class in *d3rlpy.preprocessing*), 343
- ## N
- n\_frames (*d3rlpy.algos.AWAC* attribute), 130
- n\_frames (*d3rlpy.algos.AWR* attribute), 119
- n\_frames (*d3rlpy.algos.BC* attribute), 20
- n\_frames (*d3rlpy.algos.BCQ* attribute), 69
- n\_frames (*d3rlpy.algos.BEAR* attribute), 82
- n\_frames (*d3rlpy.algos.COMBO* attribute), 204
- n\_frames (*d3rlpy.algos.CQL* attribute), 107
- n\_frames (*d3rlpy.algos.CRR* attribute), 94
- n\_frames (*d3rlpy.algos.DDPG* attribute), 32
- n\_frames (*d3rlpy.algos.DiscreteAWR* attribute), 297
- n\_frames (*d3rlpy.algos.DiscreteBC* attribute), 226
- n\_frames (*d3rlpy.algos.DiscreteBCQ* attribute), 273
- n\_frames (*d3rlpy.algos.DiscreteCQL* attribute), 285
- n\_frames (*d3rlpy.algos.DiscreteRandomPolicy* attribute), 307
- n\_frames (*d3rlpy.algos.DiscreteSAC* attribute), 261
- n\_frames (*d3rlpy.algos.DoubleDQN* attribute), 249
- n\_frames (*d3rlpy.algos.DQN* attribute), 238
- n\_frames (*d3rlpy.algos.IQL* attribute), 179
- n\_frames (*d3rlpy.algos.MOPO* attribute), 192
- n\_frames (*d3rlpy.algos.PLAS* attribute), 143
- n\_frames (*d3rlpy.algos.PLASWithPerturbation* attribute), 155
- n\_frames (*d3rlpy.algos.RandomPolicy* attribute), 215
- n\_frames (*d3rlpy.algos.SAC* attribute), 56
- n\_frames (*d3rlpy.algos.TD3* attribute), 44
- n\_frames (*d3rlpy.algos.TD3PlusBC* attribute), 167
- n\_frames (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* attribute), 407
- n\_frames (*d3rlpy.ope.DiscreteFQE* attribute), 386
- n\_frames (*d3rlpy.ope.FQE* attribute), 374
- n\_greedy\_quantiles (*d3rlpy.models.q\_functions.IQNQFunctionFactory* attribute), 312
- n\_quantiles (*d3rlpy.models.q\_functions.FQFQFunctionFactory* attribute), 313
- n\_quantiles (*d3rlpy.models.q\_functions.IQNQFunctionFactory* attribute), 312
- n\_quantiles (*d3rlpy.models.q\_functions.QRQFunctionFactory* attribute), 311
- n\_steps (*d3rlpy.algos.AWAC* attribute), 131
- n\_steps (*d3rlpy.algos.AWR* attribute), 119
- n\_steps (*d3rlpy.algos.BC* attribute), 20
- n\_steps (*d3rlpy.algos.BCQ* attribute), 69
- n\_steps (*d3rlpy.algos.BEAR* attribute), 82
- n\_steps (*d3rlpy.algos.COMBO* attribute), 204
- n\_steps (*d3rlpy.algos.CQL* attribute), 107
- n\_steps (*d3rlpy.algos.CRR* attribute), 95
- n\_steps (*d3rlpy.algos.DDPG* attribute), 32
- n\_steps (*d3rlpy.algos.DiscreteAWR* attribute), 297
- n\_steps (*d3rlpy.algos.DiscreteBC* attribute), 226
- n\_steps (*d3rlpy.algos.DiscreteBCQ* attribute), 273
- n\_steps (*d3rlpy.algos.DiscreteCQL* attribute), 285
- n\_steps (*d3rlpy.algos.DiscreteRandomPolicy* attribute), 307
- n\_steps (*d3rlpy.algos.DiscreteSAC* attribute), 262
- n\_steps (*d3rlpy.algos.DoubleDQN* attribute), 250
- n\_steps (*d3rlpy.algos.DQN* attribute), 238
- n\_steps (*d3rlpy.algos.IQL* attribute), 179
- n\_steps (*d3rlpy.algos.MOPO* attribute), 192
- n\_steps (*d3rlpy.algos.PLAS* attribute), 143
- n\_steps (*d3rlpy.algos.PLASWithPerturbation* attribute), 155
- n\_steps (*d3rlpy.algos.RandomPolicy* attribute), 215
- n\_steps (*d3rlpy.algos.SAC* attribute), 56
- n\_steps (*d3rlpy.algos.TD3* attribute), 44
- n\_steps (*d3rlpy.algos.TD3PlusBC* attribute), 167

- n\_steps (*d3rlpy.dataset.TransitionMiniBatch* attribute), 324  
 n\_steps (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* attribute), 407  
 n\_steps (*d3rlpy.ope.DiscreteFQE* attribute), 386  
 n\_steps (*d3rlpy.ope.FQE* attribute), 374  
 next\_action (*d3rlpy.dataset.Transition* attribute), 322  
 next\_actions (*d3rlpy.dataset.TransitionMiniBatch* attribute), 324  
 next\_observation (*d3rlpy.dataset.Transition* attribute), 322  
 next\_observations (*d3rlpy.dataset.TransitionMiniBatch* attribute), 324  
 next\_reward (*d3rlpy.dataset.Transition* attribute), 322  
 next\_rewards (*d3rlpy.dataset.TransitionMiniBatch* attribute), 324  
 next\_transition (*d3rlpy.dataset.Transition* attribute), 322  
 NormalNoise (class in *d3rlpy.online.explorers*), 396
- ## O
- observation (*d3rlpy.dataset.Transition* attribute), 322  
 observation\_shape (*d3rlpy.algos.AWAC* attribute), 131  
 observation\_shape (*d3rlpy.algos.AWR* attribute), 119  
 observation\_shape (*d3rlpy.algos.BC* attribute), 20  
 observation\_shape (*d3rlpy.algos.BCQ* attribute), 69  
 observation\_shape (*d3rlpy.algos.BEAR* attribute), 82  
 observation\_shape (*d3rlpy.algos.COMBO* attribute), 204  
 observation\_shape (*d3rlpy.algos.CQL* attribute), 107  
 observation\_shape (*d3rlpy.algos.CRR* attribute), 95  
 observation\_shape (*d3rlpy.algos.DDPG* attribute), 32  
 observation\_shape (*d3rlpy.algos.DiscreteAWR* attribute), 297  
 observation\_shape (*d3rlpy.algos.DiscreteBC* attribute), 227  
 observation\_shape (*d3rlpy.algos.DiscreteBCQ* attribute), 273  
 observation\_shape (*d3rlpy.algos.DiscreteCQL* attribute), 285  
 observation\_shape (*d3rlpy.algos.DiscreteRandomPolicy* attribute), 307  
 observation\_shape (*d3rlpy.algos.DiscreteSAC* attribute), 262  
 observation\_shape (*d3rlpy.algos.DoubleDQN* attribute), 250  
 observation\_shape (*d3rlpy.algos.DQN* attribute), 238  
 observation\_shape (*d3rlpy.algos.IQL* attribute), 179  
 observation\_shape (*d3rlpy.algos.MOPO* attribute), 192  
 observation\_shape (*d3rlpy.algos.PLAS* attribute), 143  
 observation\_shape (*d3rlpy.algos.PLASWithPerturbation* attribute), 155  
 observation\_shape (*d3rlpy.algos.RandomPolicy* attribute), 215  
 observation\_shape (*d3rlpy.algos.SAC* attribute), 56  
 observation\_shape (*d3rlpy.algos.TD3* attribute), 44  
 observation\_shape (*d3rlpy.algos.TD3PlusBC* attribute), 167  
 observation\_shape (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* attribute), 407  
 observation\_shape (*d3rlpy.ope.DiscreteFQE* attribute), 386  
 observation\_shape (*d3rlpy.ope.FQE* attribute), 374  
 observations (*d3rlpy.dataset.Episode* attribute), 320  
 observations (*d3rlpy.dataset.MDPDataset* attribute), 318  
 observations (*d3rlpy.dataset.TransitionMiniBatch* attribute), 325  
 OptimizerFactory (class in *d3rlpy.models.optimizers*), 345
- ## P
- PixelEncoderFactory (class in *d3rlpy.models.encoders*), 351  
 PixelScaler (class in *d3rlpy.preprocessing*), 329  
 PLAS (class in *d3rlpy.algos*), 131  
 PLASWithPerturbation (class in *d3rlpy.algos*), 144  
 predict() (*d3rlpy.algos.AWAC* method), 127  
 predict() (*d3rlpy.algos.AWR* method), 116  
 predict() (*d3rlpy.algos.BC* method), 17  
 predict() (*d3rlpy.algos.BCQ* method), 66  
 predict() (*d3rlpy.algos.BEAR* method), 79  
 predict() (*d3rlpy.algos.COMBO* method), 201  
 predict() (*d3rlpy.algos.CQL* method), 104  
 predict() (*d3rlpy.algos.CRR* method), 91  
 predict() (*d3rlpy.algos.DDPG* method), 29  
 predict() (*d3rlpy.algos.DiscreteAWR* method), 294  
 predict() (*d3rlpy.algos.DiscreteBC* method), 224  
 predict() (*d3rlpy.algos.DiscreteBCQ* method), 270  
 predict() (*d3rlpy.algos.DiscreteCQL* method), 282  
 predict() (*d3rlpy.algos.DiscreteRandomPolicy* method), 304  
 predict() (*d3rlpy.algos.DiscreteSAC* method), 258  
 predict() (*d3rlpy.algos.DoubleDQN* method), 246  
 predict() (*d3rlpy.algos.DQN* method), 235  
 predict() (*d3rlpy.algos.IQL* method), 176  
 predict() (*d3rlpy.algos.MOPO* method), 189  
 predict() (*d3rlpy.algos.PLAS* method), 139  
 predict() (*d3rlpy.algos.PLASWithPerturbation* method), 152  
 predict() (*d3rlpy.algos.RandomPolicy* method), 212  
 predict() (*d3rlpy.algos.SAC* method), 53  
 predict() (*d3rlpy.algos.TD3* method), 41  
 predict() (*d3rlpy.algos.TD3PlusBC* method), 164  
 predict() (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* method), 405



predict() (*d3rlpy.ope.DiscreteFQE method*), 383  
 predict() (*d3rlpy.ope.FQE method*), 371  
 predict\_value() (*d3rlpy.algos.AWAC method*), 128  
 predict\_value() (*d3rlpy.algos.AWR method*), 116  
 predict\_value() (*d3rlpy.algos.BC method*), 17  
 predict\_value() (*d3rlpy.algos.BCQ method*), 66  
 predict\_value() (*d3rlpy.algos.BEAR method*), 79  
 predict\_value() (*d3rlpy.algos.COMBO method*), 201  
 predict\_value() (*d3rlpy.algos.CQL method*), 104  
 predict\_value() (*d3rlpy.algos.CRR method*), 92  
 predict\_value() (*d3rlpy.algos.DDPG method*), 29  
 predict\_value() (*d3rlpy.algos.DiscreteAWR method*), 294  
 predict\_value() (*d3rlpy.algos.DiscreteBC method*), 224  
 predict\_value() (*d3rlpy.algos.DiscreteBCQ method*), 270  
 predict\_value() (*d3rlpy.algos.DiscreteCQL method*), 282  
 predict\_value() (*d3rlpy.algos.DiscreteRandomPolicy method*), 304  
 predict\_value() (*d3rlpy.algos.DiscreteSAC method*), 259  
 predict\_value() (*d3rlpy.algos.DoubleDQN method*), 247  
 predict\_value() (*d3rlpy.algos.DQN method*), 235  
 predict\_value() (*d3rlpy.algos.IQL method*), 176  
 predict\_value() (*d3rlpy.algos.MOPO method*), 189  
 predict\_value() (*d3rlpy.algos.PLAS method*), 140  
 predict\_value() (*d3rlpy.algos.PLASWithPerturbation method*), 152  
 predict\_value() (*d3rlpy.algos.RandomPolicy method*), 212  
 predict\_value() (*d3rlpy.algos.SAC method*), 53  
 predict\_value() (*d3rlpy.algos.TD3 method*), 41  
 predict\_value() (*d3rlpy.algos.TD3PlusBC method*), 164  
 predict\_value() (*d3rlpy.ope.DiscreteFQE method*), 383  
 predict\_value() (*d3rlpy.ope.FQE method*), 371  
 prev\_transition (*d3rlpy.dataset.Transition attribute*), 322  
 ProbabilisticEnsembleDynamics (*class in d3rlpy.dynamics*), 400  
**Q**  
 QRQFunctionFactory (*class in d3rlpy.models.q\_functions*), 310  
**R**  
 RandomPolicy (*class in d3rlpy.algos*), 205  
 ReplayBuffer (*class in d3rlpy.online.buffers*), 393  
 reset\_optimizer\_states() (*d3rlpy.algos.AWAC method*), 128  
 reset\_optimizer\_states() (*d3rlpy.algos.AWR method*), 116  
 reset\_optimizer\_states() (*d3rlpy.algos.BC method*), 17  
 reset\_optimizer\_states() (*d3rlpy.algos.BCQ method*), 67  
 reset\_optimizer\_states() (*d3rlpy.algos.BEAR method*), 79  
 reset\_optimizer\_states() (*d3rlpy.algos.COMBO method*), 202  
 reset\_optimizer\_states() (*d3rlpy.algos.CQL method*), 105  
 reset\_optimizer\_states() (*d3rlpy.algos.CRR method*), 92  
 reset\_optimizer\_states() (*d3rlpy.algos.DDPG method*), 30  
 reset\_optimizer\_states() (*d3rlpy.algos.DiscreteAWR method*), 294  
 reset\_optimizer\_states() (*d3rlpy.algos.DiscreteBC method*), 224  
 reset\_optimizer\_states() (*d3rlpy.algos.DiscreteBCQ method*), 271  
 reset\_optimizer\_states() (*d3rlpy.algos.DiscreteCQL method*), 283  
 reset\_optimizer\_states() (*d3rlpy.algos.DiscreteRandomPolicy method*), 305  
 reset\_optimizer\_states() (*d3rlpy.algos.DiscreteSAC method*), 259  
 reset\_optimizer\_states() (*d3rlpy.algos.DoubleDQN method*), 247  
 reset\_optimizer\_states() (*d3rlpy.algos.DQN method*), 236  
 reset\_optimizer\_states() (*d3rlpy.algos.IQL method*), 177  
 reset\_optimizer\_states() (*d3rlpy.algos.MOPO method*), 189  
 reset\_optimizer\_states() (*d3rlpy.algos.PLAS method*), 140  
 reset\_optimizer\_states() (*d3rlpy.algos.PLASWithPerturbation method*), 153  
 reset\_optimizer\_states() (*d3rlpy.algos.RandomPolicy method*), 213  
 reset\_optimizer\_states() (*d3rlpy.algos.SAC method*), 54  
 reset\_optimizer\_states() (*d3rlpy.algos.TD3 method*), 42  
 reset\_optimizer\_states() (*d3rlpy.algos.TD3PlusBC method*), 165  
 reset\_optimizer\_states() (*d3rlpy.ope.DiscreteFQE method*), 384  
 reset\_optimizer\_states() (*d3rlpy.ope.FQE method*), 372

- `reverse_transform()`  
(*d3rlpy.preprocessing.ClipRewardScaler* method), 342
  - `reverse_transform()`  
(*d3rlpy.preprocessing.MinMaxActionScaler* method), 336
  - `reverse_transform()`  
(*d3rlpy.preprocessing.MinMaxRewardScaler* method), 339
  - `reverse_transform()`  
(*d3rlpy.preprocessing.MinMaxScaler* method), 332
  - `reverse_transform()`  
(*d3rlpy.preprocessing.MultiplyRewardScaler* method), 343
  - `reverse_transform()`  
(*d3rlpy.preprocessing.PixelScaler* method), 330
  - `reverse_transform()`  
(*d3rlpy.preprocessing.StandardRewardScaler* method), 340
  - `reverse_transform()`  
(*d3rlpy.preprocessing.StandardScaler* method), 334
  - `reverse_transform_numpy()`  
(*d3rlpy.preprocessing.MinMaxActionScaler* method), 336
  - `reward` (*d3rlpy.dataset.Transition* attribute), 322
  - `reward_scaler` (*d3rlpy.algos.AWAC* attribute), 131
  - `reward_scaler` (*d3rlpy.algos.AWR* attribute), 119
  - `reward_scaler` (*d3rlpy.algos.BC* attribute), 20
  - `reward_scaler` (*d3rlpy.algos.BCQ* attribute), 69
  - `reward_scaler` (*d3rlpy.algos.BEAR* attribute), 82
  - `reward_scaler` (*d3rlpy.algos.COMBO* attribute), 205
  - `reward_scaler` (*d3rlpy.algos.CQL* attribute), 107
  - `reward_scaler` (*d3rlpy.algos.CRR* attribute), 95
  - `reward_scaler` (*d3rlpy.algos.DDPG* attribute), 32
  - `reward_scaler` (*d3rlpy.algos.DiscreteAWR* attribute), 297
  - `reward_scaler` (*d3rlpy.algos.DiscreteBC* attribute), 227
  - `reward_scaler` (*d3rlpy.algos.DiscreteBCQ* attribute), 274
  - `reward_scaler` (*d3rlpy.algos.DiscreteCQL* attribute), 285
  - `reward_scaler` (*d3rlpy.algos.DiscreteRandomPolicy* attribute), 308
  - `reward_scaler` (*d3rlpy.algos.DiscreteSAC* attribute), 262
  - `reward_scaler` (*d3rlpy.algos.DoubleDQN* attribute), 250
  - `reward_scaler` (*d3rlpy.algos.DQN* attribute), 238
  - `reward_scaler` (*d3rlpy.algos.IQL* attribute), 180
  - `reward_scaler` (*d3rlpy.algos.MOPO* attribute), 192
  - `reward_scaler` (*d3rlpy.algos.PLAS* attribute), 143
  - `reward_scaler` (*d3rlpy.algos.PLASWithPerturbation* attribute), 155
  - `reward_scaler` (*d3rlpy.algos.RandomPolicy* attribute), 216
  - `reward_scaler` (*d3rlpy.algos.SAC* attribute), 57
  - `reward_scaler` (*d3rlpy.algos.TD3* attribute), 44
  - `reward_scaler` (*d3rlpy.algos.TD3PlusBC* attribute), 167
  - `reward_scaler` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* attribute), 407
  - `reward_scaler` (*d3rlpy.ope.DiscreteFQE* attribute), 386
  - `reward_scaler` (*d3rlpy.ope.FQE* attribute), 375
  - `rewards` (*d3rlpy.dataset.Episode* attribute), 320
  - `rewards` (*d3rlpy.dataset.MDPDataset* attribute), 318
  - `rewards` (*d3rlpy.dataset.TransitionMiniBatch* attribute), 325
  - `RMSpropFactory` (class in *d3rlpy.models.optimizers*), 347
- ## S
- `SAC` (class in *d3rlpy.algos*), 45
  - `sample()` (*d3rlpy.online.buffers.BatchReplayBuffer* method), 398
  - `sample()` (*d3rlpy.online.buffers.ReplayBuffer* method), 394
  - `sample()` (*d3rlpy.online.explorers.ConstantEpsilonGreedy* method), 395
  - `sample()` (*d3rlpy.online.explorers.LinearDecayEpsilonGreedy* method), 396
  - `sample()` (*d3rlpy.online.explorers.NormalNoise* method), 396
  - `sample_action()` (*d3rlpy.algos.AWAC* method), 128
  - `sample_action()` (*d3rlpy.algos.AWR* method), 116
  - `sample_action()` (*d3rlpy.algos.BC* method), 18
  - `sample_action()` (*d3rlpy.algos.BCQ* method), 67
  - `sample_action()` (*d3rlpy.algos.BEAR* method), 80
  - `sample_action()` (*d3rlpy.algos.COMBO* method), 202
  - `sample_action()` (*d3rlpy.algos.CQL* method), 105
  - `sample_action()` (*d3rlpy.algos.CRR* method), 92
  - `sample_action()` (*d3rlpy.algos.DDPG* method), 30
  - `sample_action()` (*d3rlpy.algos.DiscreteAWR* method), 294
  - `sample_action()` (*d3rlpy.algos.DiscreteBC* method), 224
  - `sample_action()` (*d3rlpy.algos.DiscreteBCQ* method), 271
  - `sample_action()` (*d3rlpy.algos.DiscreteCQL* method), 283
  - `sample_action()` (*d3rlpy.algos.DiscreteRandomPolicy* method), 305
  - `sample_action()` (*d3rlpy.algos.DiscreteSAC* method), 259



`sample_action()` (*d3rlpy.algos.DoubleDQN method*), 247  
`sample_action()` (*d3rlpy.algos.DQN method*), 236  
`sample_action()` (*d3rlpy.algos.IQL method*), 177  
`sample_action()` (*d3rlpy.algos.MOPO method*), 190  
`sample_action()` (*d3rlpy.algos.PLAS method*), 140  
`sample_action()` (*d3rlpy.algos.PLASWithPerturbation method*), 153  
`sample_action()` (*d3rlpy.algos.RandomPolicy method*), 213  
`sample_action()` (*d3rlpy.algos.SAC method*), 54  
`sample_action()` (*d3rlpy.algos.TD3 method*), 42  
`sample_action()` (*d3rlpy.algos.TD3PlusBC method*), 165  
`sample_action()` (*d3rlpy.ope.DiscreteFQE method*), 384  
`sample_action()` (*d3rlpy.ope.FQE method*), 372  
`save_model()` (*d3rlpy.algos.AWAC method*), 128  
`save_model()` (*d3rlpy.algos.AWR method*), 116  
`save_model()` (*d3rlpy.algos.BC method*), 18  
`save_model()` (*d3rlpy.algos.BCQ method*), 67  
`save_model()` (*d3rlpy.algos.BEAR method*), 80  
`save_model()` (*d3rlpy.algos.COMBO method*), 202  
`save_model()` (*d3rlpy.algos.CQL method*), 105  
`save_model()` (*d3rlpy.algos.CRR method*), 92  
`save_model()` (*d3rlpy.algos.DDPG method*), 30  
`save_model()` (*d3rlpy.algos.DiscreteAWR method*), 294  
`save_model()` (*d3rlpy.algos.DiscreteBC method*), 224  
`save_model()` (*d3rlpy.algos.DiscreteBCQ method*), 271  
`save_model()` (*d3rlpy.algos.DiscreteCQL method*), 283  
`save_model()` (*d3rlpy.algos.DiscreteRandomPolicy method*), 305  
`save_model()` (*d3rlpy.algos.DiscreteSAC method*), 259  
`save_model()` (*d3rlpy.algos.DoubleDQN method*), 247  
`save_model()` (*d3rlpy.algos.DQN method*), 236  
`save_model()` (*d3rlpy.algos.IQL method*), 177  
`save_model()` (*d3rlpy.algos.MOPO method*), 190  
`save_model()` (*d3rlpy.algos.PLAS method*), 141  
`save_model()` (*d3rlpy.algos.PLASWithPerturbation method*), 153  
`save_model()` (*d3rlpy.algos.RandomPolicy method*), 213  
`save_model()` (*d3rlpy.algos.SAC method*), 54  
`save_model()` (*d3rlpy.algos.TD3 method*), 42  
`save_model()` (*d3rlpy.algos.TD3PlusBC method*), 165  
`save_model()` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics method*), 405  
`save_model()` (*d3rlpy.ope.DiscreteFQE method*), 384  
`save_model()` (*d3rlpy.ope.FQE method*), 372  
`save_params()` (*d3rlpy.algos.AWAC method*), 129  
`save_params()` (*d3rlpy.algos.AWR method*), 117  
`save_params()` (*d3rlpy.algos.BC method*), 18  
`save_params()` (*d3rlpy.algos.BCQ method*), 67  
`save_params()` (*d3rlpy.algos.BEAR method*), 80  
`save_params()` (*d3rlpy.algos.COMBO method*), 202  
`save_params()` (*d3rlpy.algos.CQL method*), 105  
`save_params()` (*d3rlpy.algos.CRR method*), 93  
`save_params()` (*d3rlpy.algos.DDPG method*), 30  
`save_params()` (*d3rlpy.algos.DiscreteAWR method*), 295  
`save_params()` (*d3rlpy.algos.DiscreteBC method*), 225  
`save_params()` (*d3rlpy.algos.DiscreteBCQ method*), 271  
`save_params()` (*d3rlpy.algos.DiscreteCQL method*), 283  
`save_params()` (*d3rlpy.algos.DiscreteRandomPolicy method*), 305  
`save_params()` (*d3rlpy.algos.DiscreteSAC method*), 260  
`save_params()` (*d3rlpy.algos.DoubleDQN method*), 248  
`save_params()` (*d3rlpy.algos.DQN method*), 236  
`save_params()` (*d3rlpy.algos.IQL method*), 177  
`save_params()` (*d3rlpy.algos.MOPO method*), 190  
`save_params()` (*d3rlpy.algos.PLAS method*), 141  
`save_params()` (*d3rlpy.algos.PLASWithPerturbation method*), 153  
`save_params()` (*d3rlpy.algos.RandomPolicy method*), 213  
`save_params()` (*d3rlpy.algos.SAC method*), 54  
`save_params()` (*d3rlpy.algos.TD3 method*), 42  
`save_params()` (*d3rlpy.algos.TD3PlusBC method*), 165  
`save_params()` (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics method*), 405  
`save_params()` (*d3rlpy.ope.DiscreteFQE method*), 384  
`save_params()` (*d3rlpy.ope.FQE method*), 372  
`save_policy()` (*d3rlpy.algos.AWAC method*), 129  
`save_policy()` (*d3rlpy.algos.AWR method*), 117  
`save_policy()` (*d3rlpy.algos.BC method*), 18  
`save_policy()` (*d3rlpy.algos.BCQ method*), 67  
`save_policy()` (*d3rlpy.algos.BEAR method*), 80  
`save_policy()` (*d3rlpy.algos.COMBO method*), 202  
`save_policy()` (*d3rlpy.algos.CQL method*), 105  
`save_policy()` (*d3rlpy.algos.CRR method*), 93  
`save_policy()` (*d3rlpy.algos.DDPG method*), 30  
`save_policy()` (*d3rlpy.algos.DiscreteAWR method*), 295  
`save_policy()` (*d3rlpy.algos.DiscreteBC method*), 225  
`save_policy()` (*d3rlpy.algos.DiscreteBCQ method*), 271  
`save_policy()` (*d3rlpy.algos.DiscreteCQL method*), 283  
`save_policy()` (*d3rlpy.algos.DiscreteRandomPolicy method*), 305  
`save_policy()` (*d3rlpy.algos.DiscreteSAC method*), 260  
`save_policy()` (*d3rlpy.algos.DoubleDQN method*), 248

- save\_policy() (*d3rlpy.algos.DQN method*), 236  
 save\_policy() (*d3rlpy.algos.IQL method*), 177  
 save\_policy() (*d3rlpy.algos.MOPO method*), 190  
 save\_policy() (*d3rlpy.algos.PLAS method*), 141  
 save\_policy() (*d3rlpy.algos.PLASWithPerturbation method*), 153  
 save\_policy() (*d3rlpy.algos.RandomPolicy method*), 213  
 save\_policy() (*d3rlpy.algos.SAC method*), 54  
 save\_policy() (*d3rlpy.algos.TD3 method*), 42  
 save\_policy() (*d3rlpy.algos.TD3PlusBC method*), 165  
 save\_policy() (*d3rlpy.ope.DiscreteFQE method*), 384  
 save\_policy() (*d3rlpy.ope.FQE method*), 372  
 scaler (*d3rlpy.algos.AWAC attribute*), 131  
 scaler (*d3rlpy.algos.AWR attribute*), 119  
 scaler (*d3rlpy.algos.BC attribute*), 20  
 scaler (*d3rlpy.algos.BCQ attribute*), 69  
 scaler (*d3rlpy.algos.BEAR attribute*), 82  
 scaler (*d3rlpy.algos.COMBO attribute*), 205  
 scaler (*d3rlpy.algos.CQL attribute*), 107  
 scaler (*d3rlpy.algos.CRR attribute*), 95  
 scaler (*d3rlpy.algos.DDPG attribute*), 32  
 scaler (*d3rlpy.algos.DiscreteAWR attribute*), 297  
 scaler (*d3rlpy.algos.DiscreteBC attribute*), 227  
 scaler (*d3rlpy.algos.DiscreteBCQ attribute*), 274  
 scaler (*d3rlpy.algos.DiscreteCQL attribute*), 285  
 scaler (*d3rlpy.algos.DiscreteRandomPolicy attribute*), 308  
 scaler (*d3rlpy.algos.DiscreteSAC attribute*), 262  
 scaler (*d3rlpy.algos.DoubleDQN attribute*), 250  
 scaler (*d3rlpy.algos.DQN attribute*), 238  
 scaler (*d3rlpy.algos.IQL attribute*), 180  
 scaler (*d3rlpy.algos.MOPO attribute*), 192  
 scaler (*d3rlpy.algos.PLAS attribute*), 143  
 scaler (*d3rlpy.algos.PLASWithPerturbation attribute*), 155  
 scaler (*d3rlpy.algos.RandomPolicy attribute*), 216  
 scaler (*d3rlpy.algos.SAC attribute*), 57  
 scaler (*d3rlpy.algos.TD3 attribute*), 44  
 scaler (*d3rlpy.algos.TD3PlusBC attribute*), 167  
 scaler (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics attribute*), 407  
 scaler (*d3rlpy.ope.DiscreteFQE attribute*), 386  
 scaler (*d3rlpy.ope.FQE attribute*), 375  
 set\_active\_logger() (*d3rlpy.algos.AWAC method*), 129  
 set\_active\_logger() (*d3rlpy.algos.AWR method*), 117  
 set\_active\_logger() (*d3rlpy.algos.BC method*), 18  
 set\_active\_logger() (*d3rlpy.algos.BCQ method*), 67  
 set\_active\_logger() (*d3rlpy.algos.BEAR method*), 80  
 set\_active\_logger() (*d3rlpy.algos.COMBO method*), 203  
 set\_active\_logger() (*d3rlpy.algos.CQL method*), 105  
 set\_active\_logger() (*d3rlpy.algos.CRR method*), 93  
 set\_active\_logger() (*d3rlpy.algos.DDPG method*), 30  
 set\_active\_logger() (*d3rlpy.algos.DiscreteAWR method*), 295  
 set\_active\_logger() (*d3rlpy.algos.DiscreteBC method*), 225  
 set\_active\_logger() (*d3rlpy.algos.DiscreteBCQ method*), 272  
 set\_active\_logger() (*d3rlpy.algos.DiscreteCQL method*), 283  
 set\_active\_logger() (*d3rlpy.algos.DiscreteRandomPolicy method*), 306  
 set\_active\_logger() (*d3rlpy.algos.DiscreteSAC method*), 260  
 set\_active\_logger() (*d3rlpy.algos.DoubleDQN method*), 248  
 set\_active\_logger() (*d3rlpy.algos.DQN method*), 236  
 set\_active\_logger() (*d3rlpy.algos.IQL method*), 178  
 set\_active\_logger() (*d3rlpy.algos.MOPO method*), 190  
 set\_active\_logger() (*d3rlpy.algos.PLAS method*), 141  
 set\_active\_logger() (*d3rlpy.algos.PLASWithPerturbation method*), 154  
 set\_active\_logger() (*d3rlpy.algos.RandomPolicy method*), 214  
 set\_active\_logger() (*d3rlpy.algos.SAC method*), 55  
 set\_active\_logger() (*d3rlpy.algos.TD3 method*), 42  
 set\_active\_logger() (*d3rlpy.algos.TD3PlusBC method*), 166  
 set\_active\_logger() (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics method*), 405  
 set\_active\_logger() (*d3rlpy.ope.DiscreteFQE method*), 384  
 set\_active\_logger() (*d3rlpy.ope.FQE method*), 373  
 set\_grad\_step() (*d3rlpy.algos.AWAC method*), 129  
 set\_grad\_step() (*d3rlpy.algos.AWR method*), 117  
 set\_grad\_step() (*d3rlpy.algos.BC method*), 18  
 set\_grad\_step() (*d3rlpy.algos.BCQ method*), 68  
 set\_grad\_step() (*d3rlpy.algos.BEAR method*), 81  
 set\_grad\_step() (*d3rlpy.algos.COMBO method*), 203  
 set\_grad\_step() (*d3rlpy.algos.CQL method*), 106  
 set\_grad\_step() (*d3rlpy.algos.CRR method*), 93  
 set\_grad\_step() (*d3rlpy.algos.DDPG method*), 31  
 set\_grad\_step() (*d3rlpy.algos.DiscreteAWR method*), 295  
 set\_grad\_step() (*d3rlpy.algos.DiscreteBC method*),

- 225  
 set\_grad\_step() (*d3rlpy.algos.DiscreteBCQ method*), 272  
 set\_grad\_step() (*d3rlpy.algos.DiscreteCQL method*), 284  
 set\_grad\_step() (*d3rlpy.algos.DiscreteRandomPolicy method*), 306  
 set\_grad\_step() (*d3rlpy.algos.DiscreteSAC method*), 260  
 set\_grad\_step() (*d3rlpy.algos.DoubleDQN method*), 248  
 set\_grad\_step() (*d3rlpy.algos.DQN method*), 237  
 set\_grad\_step() (*d3rlpy.algos.IQL method*), 178  
 set\_grad\_step() (*d3rlpy.algos.MOPO method*), 190  
 set\_grad\_step() (*d3rlpy.algos.PLAS method*), 141  
 set\_grad\_step() (*d3rlpy.algos.PLASWithPerturbation method*), 154  
 set\_grad\_step() (*d3rlpy.algos.RandomPolicy method*), 214  
 set\_grad\_step() (*d3rlpy.algos.SAC method*), 55  
 set\_grad\_step() (*d3rlpy.algos.TD3 method*), 43  
 set\_grad\_step() (*d3rlpy.algos.TD3PlusBC method*), 166  
 set\_grad\_step() (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics method*), 405  
 set\_grad\_step() (*d3rlpy.ope.DiscreteFQE method*), 385  
 set\_grad\_step() (*d3rlpy.ope.FQE method*), 373  
 set\_params() (*d3rlpy.algos.AWAC method*), 129  
 set\_params() (*d3rlpy.algos.AWR method*), 117  
 set\_params() (*d3rlpy.algos.BC method*), 19  
 set\_params() (*d3rlpy.algos.BCQ method*), 68  
 set\_params() (*d3rlpy.algos.BEAR method*), 81  
 set\_params() (*d3rlpy.algos.COMBO method*), 203  
 set\_params() (*d3rlpy.algos.CQL method*), 106  
 set\_params() (*d3rlpy.algos.CRR method*), 93  
 set\_params() (*d3rlpy.algos.DDPG method*), 31  
 set\_params() (*d3rlpy.algos.DiscreteAWR method*), 295  
 set\_params() (*d3rlpy.algos.DiscreteBC method*), 225  
 set\_params() (*d3rlpy.algos.DiscreteBCQ method*), 272  
 set\_params() (*d3rlpy.algos.DiscreteCQL method*), 284  
 set\_params() (*d3rlpy.algos.DiscreteRandomPolicy method*), 306  
 set\_params() (*d3rlpy.algos.DiscreteSAC method*), 260  
 set\_params() (*d3rlpy.algos.DoubleDQN method*), 248  
 set\_params() (*d3rlpy.algos.DQN method*), 237  
 set\_params() (*d3rlpy.algos.IQL method*), 178  
 set\_params() (*d3rlpy.algos.MOPO method*), 191  
 set\_params() (*d3rlpy.algos.PLAS method*), 142  
 set\_params() (*d3rlpy.algos.PLASWithPerturbation method*), 154  
 set\_params() (*d3rlpy.algos.RandomPolicy method*), 214  
 set\_params() (*d3rlpy.algos.SAC method*), 55  
 set\_params() (*d3rlpy.algos.TD3 method*), 43  
 set\_params() (*d3rlpy.algos.TD3PlusBC method*), 166  
 set\_params() (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics method*), 406  
 set\_params() (*d3rlpy.ope.DiscreteFQE method*), 385  
 set\_params() (*d3rlpy.ope.FQE method*), 373  
 SGDFactory (*class in d3rlpy.models.optimizers*), 346  
 share\_encoder (*d3rlpy.models.q\_functions.FQFQFunctionFactory attribute*), 313  
 share\_encoder (*d3rlpy.models.q\_functions.IQNQFunctionFactory attribute*), 312  
 share\_encoder (*d3rlpy.models.q\_functions.MeanQFunctionFactory attribute*), 310  
 share\_encoder (*d3rlpy.models.q\_functions.QRQFunctionFactory attribute*), 311  
 size() (*d3rlpy.dataset.Episode method*), 320  
 size() (*d3rlpy.dataset.MDPDataset method*), 318  
 size() (*d3rlpy.dataset.TransitionMiniBatch method*), 324  
 size() (*d3rlpy.online.buffers.BatchReplayBuffer method*), 399  
 size() (*d3rlpy.online.buffers.ReplayBuffer method*), 394  
 soft\_opc\_scorer() (*in module d3rlpy.metrics.scorer*), 358  
 StandardRewardScaler (*class in d3rlpy.preprocessing*), 339  
 StandardScaler (*class in d3rlpy.preprocessing*), 333
- ## T
- TD3 (*class in d3rlpy.algos*), 33  
 TD3PlusBC (*class in d3rlpy.algos*), 156  
 td\_error\_scorer() (*in module d3rlpy.metrics.scorer*), 356  
 terminal (*d3rlpy.dataset.Episode attribute*), 320  
 terminal (*d3rlpy.dataset.Transition attribute*), 323  
 terminals (*d3rlpy.dataset.MDPDataset attribute*), 318  
 terminals (*d3rlpy.dataset.TransitionMiniBatch attribute*), 325  
 to\_mdp\_dataset() (*d3rlpy.online.buffers.BatchReplayBuffer method*), 399  
 to\_mdp\_dataset() (*d3rlpy.online.buffers.ReplayBuffer method*), 394  
 transform() (*d3rlpy.preprocessing.ClipRewardScaler method*), 342  
 transform() (*d3rlpy.preprocessing.MinMaxActionScaler method*), 336  
 transform() (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 339  
 transform() (*d3rlpy.preprocessing.MinMaxScaler method*), 332  
 transform() (*d3rlpy.preprocessing.MultiplyRewardScaler method*), 344  
 transform() (*d3rlpy.preprocessing.PixelScaler method*), 330



transform() (*d3rlpy.preprocessing.StandardRewardScaler* method), 341  
 transform() (*d3rlpy.preprocessing.StandardScaler* method), 334  
 transform\_numpy() (*d3rlpy.preprocessing.ClipRewardScaler* method), 342  
 transform\_numpy() (*d3rlpy.preprocessing.MinMaxRewardScaler* method), 339  
 transform\_numpy() (*d3rlpy.preprocessing.MultiplyRewardScaler* method), 344  
 transform\_numpy() (*d3rlpy.preprocessing.StandardRewardScaler* method), 341  
 Transition (class in *d3rlpy.dataset*), 321  
 TransitionMiniBatch (class in *d3rlpy.dataset*), 323  
 transitions (*d3rlpy.dataset.Episode* attribute), 320  
 transitions (*d3rlpy.dataset.TransitionMiniBatch* attribute), 325  
 transitions (*d3rlpy.online.buffers.BatchReplayBuffer* attribute), 399  
 transitions (*d3rlpy.online.buffers.ReplayBuffer* attribute), 395  
 TYPE (*d3rlpy.models.encoders.DefaultEncoderFactory* attribute), 351  
 TYPE (*d3rlpy.models.encoders.DenseEncoderFactory* attribute), 355  
 TYPE (*d3rlpy.models.encoders.PixelEncoderFactory* attribute), 352  
 TYPE (*d3rlpy.models.encoders.VectorEncoderFactory* attribute), 353  
 TYPE (*d3rlpy.models.q\_functions.FQFQFunctionFactory* attribute), 313  
 TYPE (*d3rlpy.models.q\_functions.IQNQFunctionFactory* attribute), 312  
 TYPE (*d3rlpy.models.q\_functions.MeanQFunctionFactory* attribute), 310  
 TYPE (*d3rlpy.models.q\_functions.QRQFunctionFactory* attribute), 311  
 TYPE (*d3rlpy.preprocessing.ClipRewardScaler* attribute), 343  
 TYPE (*d3rlpy.preprocessing.MinMaxActionScaler* attribute), 337  
 TYPE (*d3rlpy.preprocessing.MinMaxRewardScaler* attribute), 339  
 TYPE (*d3rlpy.preprocessing.MinMaxScaler* attribute), 332  
 TYPE (*d3rlpy.preprocessing.MultiplyRewardScaler* attribute), 344  
 TYPE (*d3rlpy.preprocessing.PixelScaler* attribute), 331  
 TYPE (*d3rlpy.preprocessing.StandardRewardScaler* attribute), 341  
 TYPE (*d3rlpy.preprocessing.StandardScaler* attribute), 334  
 update() (*d3rlpy.algos.AWAC* method), 130  
 update() (*d3rlpy.algos.AWR* method), 118  
 update() (*d3rlpy.algos.BC* method), 19  
 update() (*d3rlpy.algos.BCQ* method), 68  
 update() (*d3rlpy.algos.BEAR* method), 81  
 update() (*d3rlpy.algos.COMBO* method), 203  
 update() (*d3rlpy.algos.CQL* method), 106  
 update() (*d3rlpy.algos.CRR* method), 94  
 update() (*d3rlpy.algos.DDPG* method), 31  
 update() (*d3rlpy.algos.DiscreteAWR* method), 296  
 update() (*d3rlpy.algos.DiscreteBC* method), 225  
 update() (*d3rlpy.algos.DiscreteBCQ* method), 272  
 update() (*d3rlpy.algos.DiscreteCQL* method), 284  
 update() (*d3rlpy.algos.DiscreteRandomPolicy* method), 306  
 update() (*d3rlpy.algos.DiscreteSAC* method), 261  
 update() (*d3rlpy.algos.DoubleDQN* method), 249  
 update() (*d3rlpy.algos.DQN* method), 237  
 update() (*d3rlpy.algos.IQL* method), 178  
 update() (*d3rlpy.algos.MOPO* method), 191  
 update() (*d3rlpy.algos.PLAS* method), 142  
 update() (*d3rlpy.algos.PLASWithPerturbation* method), 154  
 update() (*d3rlpy.algos.RandomPolicy* method), 214  
 update() (*d3rlpy.algos.SAC* method), 55  
 update() (*d3rlpy.algos.TD3* method), 43  
 update() (*d3rlpy.algos.TD3PlusBC* method), 166  
 update() (*d3rlpy.dynamics.ProbabilisticEnsembleDynamics* method), 406  
 update() (*d3rlpy.ope.DiscreteFQE* method), 385  
 update() (*d3rlpy.ope.FQE* method), 373  
 V  
 value\_estimation\_std\_scorer() (in module *d3rlpy.metrics.scorer*), 357  
 VectorEncoderFactory (class in *d3rlpy.models.encoders*), 352