# d3rlpy

Takuma Seno

Apr 27, 2022

# TUTORIALS

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.

CHAPTER

ONE

# TUTORIALS

## 1.1 Getting Started

This tutorial is also available on Google Colaboratory

### 1.1.1 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.

### 1.1.2 Prepare Dataset

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

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

```
from d3rlpy.datasets import get_cartpole # CartPole-v0 dataset
from d3rlpy.datasets import get_pendulum # Pendulum-v0 dataset
from d3rlpy.datasets import get_pybullet # PyBullet task datasets
from d3rlpy.datasets import get_atari    # Atari 2600 task datasets
from d3rlpy.datasets import get_d4rl     # D4RL datasets
```

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

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

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)
```

### 1.1.3 Setup Algorithm

There are many algorithms avaiable in d3rlpy. Since CartPole is the simple task, let's start from `DQN`, which is the Q-learnig 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*.

### 1.1.4 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*.

### 1.1.5 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]

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

### 1.1.6 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*.

## 1.2 Play with MDPDataset

d3rlpy provides `MDPDataset`, a dedicated dataset structure for offline RL. In this tutorial, you can learn how to play with `MDPDataset`. Check *MDPDataset* for more information.

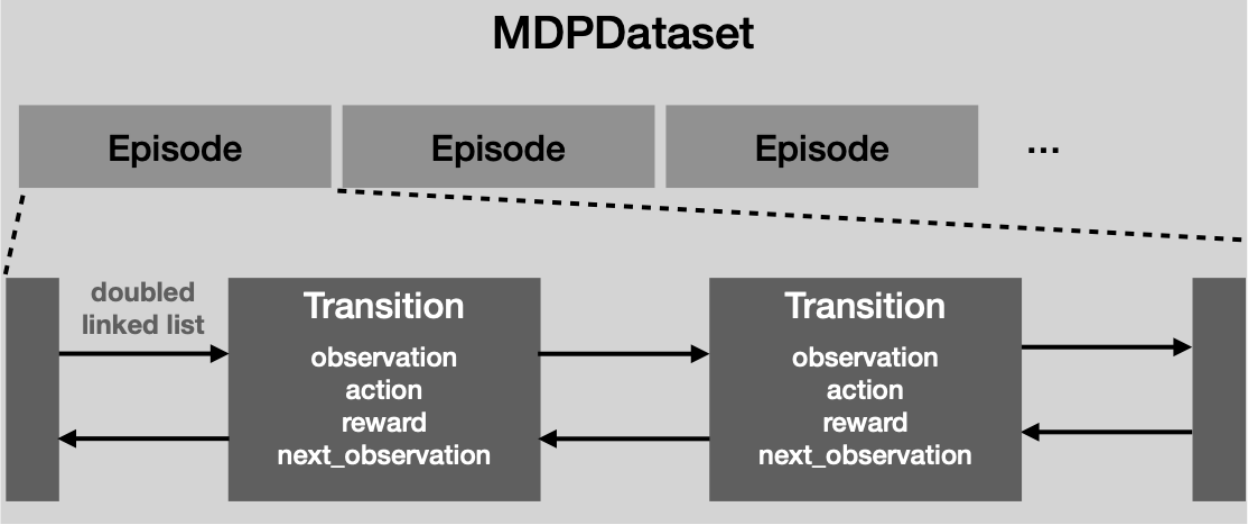
### 1.2.1 Prepare Dataset

In this tutorial, let's use a built-in dataset for CartPole-v0.

```
# prepare dataset
dataset, _ = d3rlpy.datasets.get_dataset("cartpole-random")
```

### 1.2.2 Understand Episode and Transition

`MDPDataset` hierarchically structures the dataset into `Episode` and `Transition`.




You can interact with this underlying data structure.

```
# first episode
episode = dataset.episodes[0]

# access to episode data
episode.observations
episode.actions
episode.rewards

# first transition
transition = episode.transitions[0]

# access to tuple
transition.observation
transition.action
transition.reward
transition.next_observation

# linked list structure
next_transition = transition.next_transition
assert transition is next_transition.prev_transition
```

### 1.2.3 Feed MDPDataset to Algorithm

There are multiple ways to feed datasets to algorithms for offline RL.

```
dqn = d3rlpy.algos.DQN()

# feed as MDPDataset
dqn.fit(dataset, n_steps=10000)

# feed as Episode
dqn.fit(dataset.episodes, n_steps=10000)

# feed as Transition
transitions = []
for episode in dataset.episodes:
    transitions.extend(episode.transitions)
dqn.fit(transitions, n_steps=10000)
```

The advantage of this design is that you can split datasets in both episode-wise and transition-wise. If you split datasets in episode-wise manner, you can completely remove all transitions included in test episodes, which makes valiadtion work better.

```
# use scikit-learn utility
from sklearn.model_selection import train_test_split

# episode-wise split
train_episodes, test_episodes = train_test_split(dataset.episodes)

# setup metrics
metrics = {
  "soft_opc": d3rlpy.metrics.scorer.soft_opc_scorer(return_threshold=180),
  "initial_value": d3rlpy.metrics.scorer.initial_state_value_estimation_scorer,
}

# start training with episode-wise splits
dqn.fit(
    train_episodes,
    n_steps=10000,
    scorers=metrics,
    eval_episodes=test_episodes,
)
```

### 1.2.4 Mix Datasets

You can also mix multiple datasets to train algorithms.

```
replay_dataset, _ = d3rlpy.datasets.get_dataset("cartpole-replay")

# extends replay dataset with random dataset
replay_dataset.extend(dataset)

# you can also save it and load it later
```

(continues on next page)

(continued from previous page)

```
replay_dataset.dump("mixed_dataset.h5")
mixed_dataset = MDPDataset.load("mixed_dataset.h5")
```

## 1.3 Data Collection

d3rlpy provides APIs to support data collection from environments. This feature is specifically useful if you want to build your own original datasets for research or practice purposes.

### 1.3.1 Prepare Environment

d3rlpy supports environments with OpenAI Gym interface. In this tutorial, let's use simple CartPole environment.

```
import gym

env = gym.make("CartPole-v0")
```

### 1.3.2 Data Collection with Random Policy

If you want to collect experiences with uniformly random policy, you can use `RandomPolicy` and `DiscreteRandomPolicy`. This procedure corresponds to `random` datasets in D4RL.

```
import d3rlpy

# setup algorithm
random_policy = d3rlpy.algos.DiscreteRandomPolicy()

# prepare experience replay buffer
buffer = d3rlpy.online.buffers.ReplayBuffer(maxlen=100000, env=env)

# start data collection
random_policy.collect(env, buffer, n_steps=100000)

# export as MDPDataset
dataset = buffer.to_mdp_dataset()

# save MDPDataset
dataset.dump("random_policy_dataset.h5")
```

### 1.3.3 Data Collection with Trained Policy

If you want to collect experiences with previously trained policy, you can still use the same set of APIs. This procedure corresponds to `medium` datasets in D4RL.

```
# setup algorithm
dqn = d3rlpy.algos.DQN()

# initialize neural networks before loading parameters
```

(continues on next page)

(continued from previous page)

```
dqn.build_with_env(env)

# load pretrained parameters
dqn.load_model("dqn_model.pt")

# prepare experience replay buffer
buffer = d3rlpy.online.buffers.ReplayBuffer(maxlen=100000, env=env)

# start data collection
dqn.collect(env, buffer, n_steps=100000)

# export as MDPDataset
dataset = buffer.to_mdp_dataset()

# save MDPDataset
dataset.dump("trained_policy_dataset.h5")
```

### 1.3.4 Data Collection while Training Policy

If you want to use experiences collected during training to build a new dataset, you can simply use `fit_online` and save the dataset. This procedure corresponds to `replay` datasets in D4RL.

```
# setup algorithm
dqn = d3rlpy.algos.DQN()

# prepare experience replay buffer
buffer = d3rlpy.online.buffers.ReplayBuffer(maxlen=100000, env=env)

# prepare exploration strategy if necessary
explorer = d3rlpy.online.explorers.ConstantEpsilonGreedy(0.3)

# start data collection
dqn.fit_online(env, buffer, n_steps=100000)

# export as MDPDataset
dataset = buffer.to_mdp_dataset()

# save MDPDataset
dataset.dump("replay_dataset.h5")
```

## 1.4 Create Your Dataset

The data collection API is introduced in *Data Collection*. In this tutorial, you can learn how to build your dataset from logged data such as the user data collected in your web service.

### 1.4.1 Prepare Logged Data

First of all, you need to prepare your logged data. In this tutorial, let's use randomly generated data. `terminals` represents the last step of episodes. If `terminals[i] == 1.0`, i-th step is the terminal state. Otherwise you need to set zeros for non-terminal states.

```
import numpy as np

# vector observation
# 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)
```

### 1.4.2 Build MDPDataset

Once your logged data is ready, you can build `MDPDataset` object.

```
import d3rlpy

dataset = d3rlpy.dataset.MDPDataset(
    observations=observations,
    actions=actions,
    rewards=rewards,
    terminals=terminals,
)
```

### 1.4.3 Set Timeout Flags

In RL, there is the case where you want to stop an episode without a terminal state. For example, if you're collecting data of a 4-legged robot walking forward, the walking task basically never ends as long as the robot keeps walking while the logged episode must stop somewhere. In this case, you can use `episode_terminals` to represent this timeout states.

```
# terminal states
terminals = np.zeros(1000)

# timeout states
episode_terminals = np.random.randint(2, size=1000)

dataset = d3rlpy.dataest.MDPDataset(
    observations=observations,
    actions=actions,
    rewards=rewards,
    terminals=terminals,
```

(continues on next page)

(continued from previous page)

```
    episode_terminals=episode_terminals,
)
```

## 1.5 Preprocess / Postprocess

In this tutorial, you can learn how to preprocess datasets and postprocess continuous action outputs. Please check *Preprocessing* for more information.

### 1.5.1 Preprocess Observations

If your dataset includes unnormalized observations, you can normalize or standardize the observations by specifying `scaler` argument with a string alias. In this case, the statistics of the dataset will be computed at the beginning of offline training.

```
import d3rlpy

dataset, _ = d3rlpy.datasets.get_dataset("pendulum-random")

# specify by string alias
sac = d3rlpy.algos.SAC(scaler="standard")
```

Alternatively, you can manually instantiate preprocessing parameters.

```
# setup manually
mean = np.mean(dataset.observations, axis=0, keepdims=True)
std = np.std(dataset.observations, axis=0, keepdims=True)
scaler = d3rlpy.preprocessing.StandardScaler(mean=mean, std=std)

# specify by object
sac = d3rlpy.algos.SAC(scaler=scaler)
```

Please check *Preprocessing* for the full list of available observation preprocessors.

### 1.5.2 Preprocess / Postprocess Actions

In training with continuous action-space, the actions must be in the range between `[-1.0, 1.0]` due to the underlying `tanh` activation at the policy functions. In d3rlpy, you can easily normalize inputs and denormalize outpus instead of normalizing datasets by yourself.

```
# specify by string alias
sac = d3rlpy.algos.SAC(action_scaler="min_max")

# setup manually
minimum_action = np.min(dataset.actions, axis=0, keepdims=True)
maximum_action = np.max(dataset.actions, axis=0, keepdims=True)
action_scaler = d3rlpy.preprocessing.MinMaxActionScaler(
    minimum=minimum_action,
    maximum=maximum_action,
)
```

(continues on next page)

(continued from previous page)

```
# specify by object
sac = d3rlpy.algos.SAC(action_scaler=action_scaler)
```

Please check *Preprocessing* for the full list of available action preprocessors.

### 1.5.3 Preprocess Rewards

The effect of scaling rewards is not well studied yet in RL community, however, it's confirmed that the reward scale affects training performance.

```
# specify by string alias
sac = d3rlpy.algos.SAC(reward_scaler="standard")

# setup manuall
mean = np.mean(dataset.rewards, axis=0, keepdims=True)
std = np.std(dataset.rewards, axis=0, keepdims=True)
reward_scaler = StandardRewardScaler(mean=mean, std=std)

# specify by object
sac = d3rlpy.algos.SAC(reward_scaler=reward_scaler)
```

Please check *Preprocessing* for the full list of available reward preprocessors.

## 1.6 Customize Neural Network

In this tutorial, you can learn how to integrate your own neural network models to d3rlpy. Please check *Network Architectures* for more information.

### 1.6.1 Prepare PyTorch Model

If you're familiar with PyTorch, this step should be easy for you. Please note that your model must have `get_feature_size` method to tell the feature size to the final layer.

```
import torch
import torch.nn as nn
import d3rlpy

class CustomEncoder(nn.Module):
    def __init__(self, observation_shape, feature_size):
        self.feature_size = feature_size
        self.fc1 = nn.Linear(observation_shape[0], feature_size)
        self.fc2 = nn.Linear(feature_size, feature_size)

    def forward(self, x):
        h = torch.relu(self.fc1(x))
        h = torch.relu(self.fc2(x))
        return h
```

(continues on next page)

(continued from previous page)

```
    # THIS IS IMPORTANT!
    def get_feature_size(self):
        return self.feature_size
```

### 1.6.2 Setup EncoderFactory

Once you setup your PyTorch model, you need to setup `EncoderFactory`. In your `EncoderFactory` class, you need to define `create` and `get_params` methods as well as `TYPE` attribute. `TYPE` attribute and `get_params` method are used to serialize your customized neural network configuration.

```
class CustomEncoderFactory(d3rlpy.models.encoders.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}
```

Now, you can use your model with d3rlpy.

```
# integrate your model into d3rlpy algorithm
dqn = d3rlpy.algos.DQN(encoder_factory=CustomEncoderFactory(64))
```

### 1.6.3 Support Q-function for Actor-Critic

In the above example, your original model is designed for the network that takes an observation as an input. However, if you customize a Q-function of actor-critic algorithm (e.g. SAC), you need to prepare an action-conditioned model.

```
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, feature_size)
        self.fc2 = nn.Linear(feature_size, feature_size)

      def forward(self, x, action):
          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
```

Finally, you can update your `CustomEncoderFactory` as follows.

```
class CustomEncoderFactory(EncoderFactory):
    TYPE = "custom"

    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)

    def get_params(self, deep=False):
        return {"feature_size": self.feature_size}
```

Now, you can customize actor-critic algorithms.

```
encoder_factory = CustomEncoderFactory(64)

sac = d3rlpy.algos.SAC(
    actor_encoder_factory=encoder_factory,
    critic_encoder_factory=encoder_factory,
)
```

## 1.7 Online RL

### 1.7.1 Prepare Environment

d3rlpy supports environments with OpenAI Gym interface. In this tutorial, let's use simple CartPole environment.

```
import gym

# for training
env = gym.make("CartPole-v0")

# for evaluation
eval_env = gym.make("CartPole-v0")
```

### 1.7.2 Setup Algorithm

Just like offline RL training, you can setup an algorithm object.

```
import d3rlpy

# if you don't use GPU, set use_gpu=False instead.
dqn = d3rlpy.algos.DQN(
    batch_size=32,
    learning_rate=2.5e-4,
    target_update_interval=100,
```

(continues on next page)

(continued from previous page)

```
    use_gpu=True,
)

# initialize neural networks with the given environment object.
# this is not necessary when you directly call fit or fit_online method.
dqn.build_with_env(env)
```

### 1.7.3 Setup Online RL Utilities

Unlike offline RL training, you'll need to setup an experience replay buffer and an exploration strategy.

```
# experience replay buffer
buffer = d3rlpy.online.buffers.ReplayBuffer(maxlen=100000, env=env)

# exploration strategy
# in this tutorial, epsilon-greedy policy with static epsilon=0.3
explorer = d3rlpy.online.explorers.ConstantEpsilonGreedy(0.3)
```

### 1.7.4 Start Training

Now, you have everything you need to start online RL training. Let's put them together!

```
dqn.fit_online(
    env,
    buffer,
    explorer,
    n_steps=100000,  # train for 100K steps
    eval_env=eval_env,
    n_steps_per_epoch=1000,  # evaluation is performed every 1K steps
    update_start_step=1000,  # parameter update starts after 1K steps
)
```

### 1.7.5 Train with Stochastic Policy

If the algorithm uses a stochastic policy (e.g. SAC), you can train algorithms without setting an exploration strategy.

```
sac = d3rlpy.algos.DiscreteSAC()
sac.fit_online(
    env,
    buffer,
    n_steps=100000,
    eval_env=eval_env,
    n_steps_per_epoch=1000,
    update_start_step=1000,
)
```

## 1.8 Finetuning

d3rlpy supports smooth transition from offline training to online training.

### 1.8.1 Prepare Dataset and Environment

In this tutorial, let's use a built-in dataset for CartPole-v0 environment.

```
import d3rlpy

# setup random CartPole-v0 dataset and environment
dataset, env = d3rlpy.datasets.get_dataset("cartpole-random")
```

### 1.8.2 Pretrain with Dataset

```
# setup algorithm
dqn = d3rlpy.algos.DQN()

# start offline training
dqn.fit(dataset, n_steps=100000)
```

### 1.8.3 Finetune with Environment

```
# setup experience replay buffer
buffer = d3rlpy.online.buffers.ReplayBuffer(maxlen=100000, env=env)

# setup exploration strategy if necessary
explorer = d3rlpy.online.explorers.ConstantEpsilonGreedy(0.1)

# start finetuning
dqn.fit_online(env, buffer, explorer, n_steps=100000)
```

### 1.8.4 Finetune with Saved Policy

If you want to finetune the saved policy, that's also easy to do with d3rlpy.

```
# setup algorithm
dqn = d3rlpy.algos.DQN()

# initialize neural networks before loading parameters
dqn.build_with_env(env)

# load pretrained policy
dqn.load_model("dqn_model.pt")

# start finetuning
dqn.fit_online(env, buffer, explorer, n_steps=100000)
```

## 1.9 Use Distributional Q-Function

The one of the unique features in d3rlpy is to use distributional Q-functions with arbitrary d3rlpy algorithms. The distributional Q-functions are powerful and potentially capable of improving performance of any algorithms. In this tutorial, you can learn how to use them. Check *Q Functions* for more information.

### 1.9.1 Specify by String Alias

The supported Q-functions can be specified by string alias. In this case, the default hyper-parameters will be used for the Q-function.

```
import d3rlpy

# default standard Q-function
sac = d3rlpy.algos.SAC(q_func_factory="mean")

# Quantile Regression Q-function
sac = d3rlpy.algos.SAC(q_func_factory="qr")

# Implicit Quantile Network Q-function
sac = d3rlpy.algos.SAC(q_func_factory="iqn")
```

### 1.9.2 Specify by instantiating QFunctionFactory

If you want to specify hyper-parameters, you need to instantiate a `QFunctionFactory` object.

```
# default standard Q-function
mean_q_function = d3rlpy.models.q_functions.MeanQFunctionFactory()
sac = d3rlpy.algos.SAC(q_func_factory=mean_q_function)

# Quantile Regression Q-function
qr_q_function = d3rlpy.models.q_functions.QRQFunctionFactory(n_quantiles=200)
sac = d3rlpy.algos.SAC(q_func_factory=qr_q_function)

# Implicit Quantile Network Q-function
iqn_q_function = d3rlpy.models.q_functions.IQNQFunctionFactory(
    n_quantiles=32,
    n_greedy_quantiles=64,
    embed_size=64,
)
sac = d3rlpy.algos.SAC(q_func_factory=iqn_q_function)
```

CHAPTER

TWO

# JUPYTER NOTEBOOKS

- CartPole
- Discrete Control with Atari

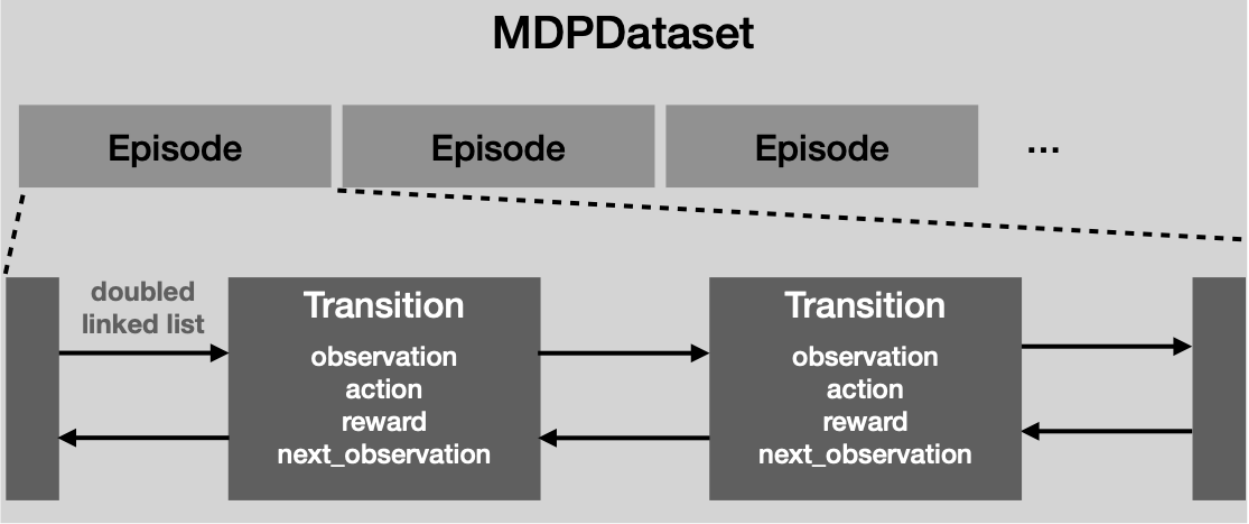
CHAPTER

# THREE

# SOFTWARE DESIGN

In this page, the software design of d3rlpy is explained.

## 3.1 MDPDataset



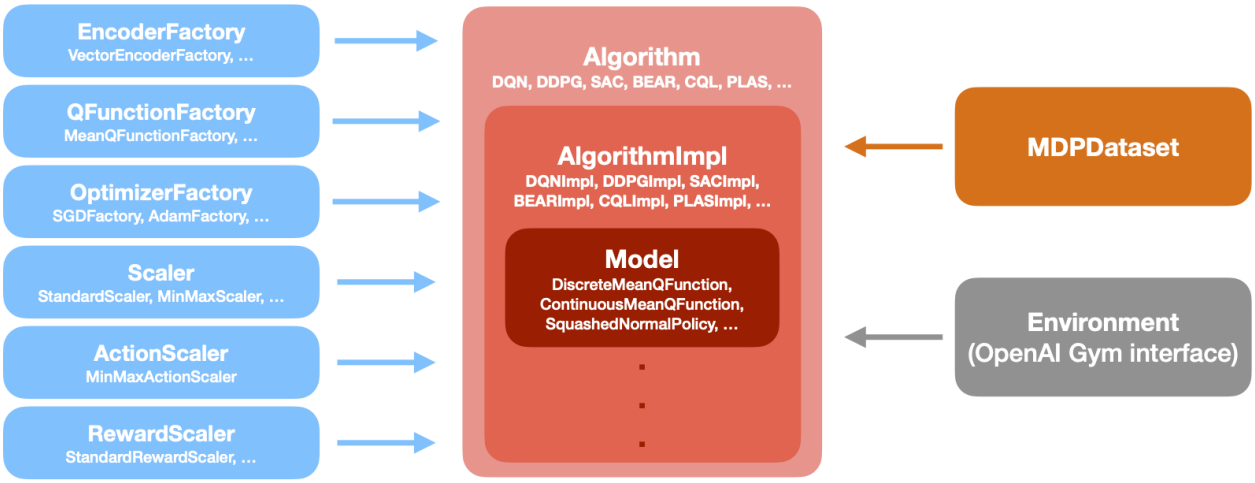

`MDPDataset` is a dedicated dataset structure for offline RL. `MDPDataset` automatically structures dataset based on `Episode` and `Transition`. `Episode` represents a single episode that includes multiple `Transition` objects collected in the episode. `Transition` represents a single tuple experience that consists of `observation`, `action`, `reward` and `next_observation`.

The advantage of this design is that you can split train and test datasets in an episode-wise manner. This feature is specifically useful for the offline RL training since holding out a continuous sequence of data is more making sense unlike a non-sequetial supervised training such as ImageNet classification models.

Regarding the engineering perspective, the underlying transition data is implemented by Cython, a Python-like language compiled to C language, to reduce the computational costs for the memory copies. This Cythonized implementation especially speeds up the cumulative returns for multi-step learning and frame-stacking for pixel observations.

Please check *Play with MDPDataset* for the tutorial and *MDPDataset* for the API reference.

## 3.2 Algorithm




The implemented algorithms are designed as above. The algorithm objects have a hierarchical structure where `Algorithm` provides the high-level API (e.g. `fit` and `fit_online`) for users and `AlgorithmImpl` provides the low-level API (e.g. `update_actor` and `update_critic`) used in the high-level API. The advantage of this design is to maximize the reusability of algorithm logics. For example, *delayed policy update* proposed in TD3 reduces the update frequency of the policy function. This mechanism can be implemented by changing the frequency of `update_actor` method calls in `Algorithm` layer without changing the underlying logics.

`Algorithm` class takes multiple components that configure the training. These are the links to the API reference.

Table 1: Algorithm Components

| Name | Reference |
|---|---|
| Algorithm | *Algorithms* |
| EncoderFactory | *Network Architectures* |
| QFunctionFactory | *Q Functions* |
| OptimizerFactory | *Optimizers* |
| Scaler | *Preprocessing* |
| ActionScaler | *Preprocessing* |
| RewardScaler | *Preprocessing* |

CHAPTER

FOUR

# API REFERENCE

## 4.1 Algorithms

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

### 4.1.1 Continuous control algorithms

| | |
|---|---|
| *d3rlpy.algos.BC* | Behavior Cloning algorithm. |
| *d3rlpy.algos.DDPG* | Deep Deterministic Policy Gradients algorithm. |
| *d3rlpy.algos.TD3* | Twin Delayed Deep Deterministic Policy Gradients algorithm. |
| *d3rlpy.algos.SAC* | Soft Actor-Critic algorithm. |
| *d3rlpy.algos.BCQ* | Batch-Constrained Q-learning algorithm. |
| *d3rlpy.algos.BEAR* | Bootstrapping Error Accumulation Reduction algorithm. |
| *d3rlpy.algos.CRR* | Critic Reguralized Regression algorithm. |
| *d3rlpy.algos.CQL* | Conservative Q-Learning algorithm. |
| *d3rlpy.algos.AWAC* | Advantage Weighted Actor-Critic algorithm. |
| *d3rlpy.algos.PLAS* | Policy in Latent Action Space algorithm. |
| *d3rlpy.algos.PLASWithPerturbation* | Policy in Latent Action Space algorithm with perturbation layer. |
| *d3rlpy.algos.TD3PlusBC* | TD3+BC algorithm. |
| *d3rlpy.algos.IQL* | Implicit Q-Learning algorithm. |
| *d3rlpy.algos.MOPO* | Model-based Offline Policy Optimization. |
| *d3rlpy.algos.COMBO* | Conservative Offline Model-Based Optimization. |
| *d3rlpy.algos.RandomPolicy* | 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*) – learing rate.
- **optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory.
- **encoder_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **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*) – implemenation of the algorithm.
- **kwargs** (*Any*) –

### Methods

**build_with_dataset**(*dataset*)
Instantiate implementation object with MDPDataset object.

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

**Return type** None

**build_with_env**(*env*)
Instantiate implementation object with OpenAI Gym object.

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

**Return type** None

**collect**(*env*, *buffer=None*, *explorer=None*, *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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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*)
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 inital 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 (for Python).
- 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*, *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 $\phi$.

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

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

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 coefficiency.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **use_gpu** (*bool, int or d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.
- **scaler** (*d3rlpy.preprocessing.Scaler or str*) – preprocessor. The available options are *['pixel', 'min_max', 'standard']*
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler or str*) – action preprocessor. The available options are `['min_max']`.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler or str*) – reward preprocessor. The available options are `['clip', 'min_max', 'standard']`.
- **impl** (*d3rlpy.algos.torch.ddpg_impl.DDPGImpl*) – algorithm implementation.
- **kwargs** (*Any*) –

### Methods

**build_with_dataset**(*dataset*)
Instantiate implementation object with MDPDataset object.

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

**Return type** None

**build_with_env**(*env*)
Instantiate implementation object with OpenAI Gym object.

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

**Return type** None

**collect**(*env*, *buffer=None*, *explorer=None*, *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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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 inital 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 (for Python).
- 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_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 $\epsilon \sim clip(N(0, \sigma), -c, c)$

### 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 coefficiency.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **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=100000)

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

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

**Return type** None

**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_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 explortion.
- **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 inital 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 (for Python).
- 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*, *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.

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

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

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

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

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

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 coefficiency.
- **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']*.
- **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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
```

(continues on next page)

(continued from previous page)

```
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_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 explortion.
- **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 inital 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 (for Python).
- 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 lgorithm. The major difference from DDPG is that the policy function is represented as combination of conditional VAE and perturbation function in order to remedy extrapolation error emerging from target value estimation.

The encoder and the decoder of the conditional VAE is represented as $E_\omega$ and $D_\omega$ respectively.

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

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

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

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

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

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

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

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

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

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

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

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

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

---

**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 coefficiency.

- **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 reguralization 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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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 inital 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 (for Python).
- 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 Discreptancy (MMD) between the policy function and the approximated beahvior 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 $k(x, y) = \exp\left((x - y)^2/(2\sigma^2)\right)$.

$\alpha$ is also adjustable through dual gradient decsent 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 coefficiency.
- **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 avaiable options are *['pixel', 'min_max', 'standard']*.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler or str*) – action preprocessor. The avaiable 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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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 inital 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 (for Python).
- 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*, *update_actor_interval=1*, *use_gpu=False*, *scaler=None*, *action_scaler=None*, *reward_scaler=None*, *impl=None*, ***kwargs*)

Critic Reguralized Regression algorithm.

CRR is a simple offline RL method similar to AWAC.

The policy is trained as a supervised regression.

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

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

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

The other is `exp` function.

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

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

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

The other one is `max`.

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

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

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

### References

- Wang et al., Critic Reguralized 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_update_type** (*str*) – target update type. The available options are `['hard', 'soft']`.
- **tau** (*float*) – target network synchronization coefficiency 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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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
```

(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 inital 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 (for Python).
- 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*, *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.

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

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.

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

where $N$ is the number of sampled actions.

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

#### References

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

**Parameters**

- **actor_learning_rate** (*float*) – learning rate for policy function.
- **critic_learning_rate** (*float*) – learning rate for Q functions.
- **temp_learning_rate** (*float*) – learning rate for temperature parameter of SAC.
- **alpha_learning_rate** (*float*) – learning rate for $\alpha$.
- **actor_optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory for the actor.

- **critic_optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory for the critic.
- **temp_optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory for the temperature.
- **alpha_optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory for $\alpha$.
- **actor_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the critic.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory or str*) – Q function factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **n_steps** (*int*) – N-step TD calculation.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficiency.
- **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 $\tau$.
- **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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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 inital 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 (for Python).
- 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*, *n_critics=2*, *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 coefficiency.
- **lam** (*float*) – $\lambda$ for weight calculation.
- **n_action_samples** (*int*) – the number of sampled actions to calculate $A^\pi(s_t, a_t)$.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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
```

(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 inital 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 (for Python).
- 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*, *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 coefficiency.
- **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.
- **warmup_steps** (*int*) – the number of steps to warmup the VAE.
- **beta** (*float*) – KL reguralization 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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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 inital 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 (for Python).
- 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*, *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*, *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 coefficiency.
- **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.
- **action_flexibility** (*float*) – output scale of perturbation layer.
- **warmup_steps** (*int*) – the number of steps to warmup the VAE.
- **beta** (*float*) – KL reguralization 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=100000)

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

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

**Return type** None

**copy_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)
```

(continues on next page)

(continued from previous page)

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

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

**Return type** None

**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_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 explortion.
- **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,)
```

(continues on next page)

(continued from previous page)

```
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 inital 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 (for Python).
- 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*, *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_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

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

#### 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 coefficiency.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **target_smoothing_sigma** (*float*) – standard deviation for target noise.
- **target_smoothing_clip** (*float*) – clipping range for target noise.
- **alpha** (*float*) – $\alpha$ value.
- **update_actor_interval** (*int*) – interval to update policy function described as *delayed policy update* in the paper.
- **use_gpu** (*bool, int or d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.
- **scaler** (*d3rlpy.preprocessing.Scaler or str*) – preprocessor. The available options are *['pixel', 'min_max', 'standard']*.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler or str*) – action preprocessor. The available options are `['min_max']`.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler or str*) – reward preprocessor. The available options are `['clip', 'min_max', 'standard']`.
- **impl** (*d3rlpy.algos.torch.td3_impl.TD3Impl*) – algorithm implementation.
- **kwargs** (*Any*) –

### Methods

**build_with_dataset**(*dataset*)
Instantiate implementation object with MDPDataset object.

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

**Return type** None

**build_with_env**(*env*)
Instantiate implementation object with OpenAI Gym object.

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

**Return type** None

**collect**(*env*, *buffer=None*, *explorer=None*, *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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.

- **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 inital 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 (for Python).
- 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*, *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'*, *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 $L_2^\tau(u) = |\tau - \mathbb{1}(u < 0)|u^2$.

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.
- **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.
- **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 coefficiency.
- **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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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 inital 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 (for Python).
- 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*, *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 probablistic ensemble dynamics model to generate new dynamics data with uncertainty penalties. The ensemble dynamics model consists of $N$ probablistic 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 reguralizes this bias.

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

where $\Sigma(s_t, a_t)$ is the estimated variance. Finally, the generated transitions $(s_t, a_t, \tilde{r_{t+1}}, s_{t+1})$ are appended to dataset $D$. This generation process starts with randomly sampled `n_initial_transitions` transitions till `horizon` steps.

**Note:** Currently, MOPO only supports vector observations.

#### References

- Yu et al., MOPO: Model-based Offline Policy Optimization.

**Parameters**

- **actor_learning_rate** (*float*) – learning rate for policy function.
- **critic_learning_rate** (*float*) – learning rate for Q functions.
- **temp_learning_rate** (*float*) – learning rate for temperature parameter.
- **actor_optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory for the actor.
- **critic_optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory for the critic.

- **temp_optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory for the temperature.
- **actor_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the critic.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory or str*) – Q function factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **n_steps** (*int*) – N-step TD calculation.
- **gamma** (*float*) – discount factor.
- **tau** (*float*) – target network synchronization coefficiency.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **update_actor_interval** (*int*) – interval to update policy function.
- **initial_temperature** (*float*) – initial temperature value.
- **dynamics** (*d3rlpy.dynamics.DynamicsBase*) – dynamics object.
- **rollout_interval** (*int*) – the number of steps before rollout.
- **rollout_horizon** (*int*) – the rollout step length.
- **rollout_batch_size** (*int*) – the number of initial transitions for rollout.
- **lam** (*float*) – $\lambda$ for uncertainty penalties.
- **real_ratio** (*float*) – the real of dataset samples in a mini-batch.
- **generated_maxlen** (*int*) – the maximum number of generated samples.
- **use_gpu** (*bool, int or d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.
- **scaler** (*d3rlpy.preprocessing.Scaler or str*) – preprocessor. The available options are *['pixel', 'min_max', 'standard']*.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler or str*) – action preprocessor. The available options are `['min_max']`.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler or str*) – reward preprocessor. The available options are `['clip', 'min_max', 'standard']`.
- **impl** (*d3rlpy.algos.torch.sac_impl.SACImpl*) – algorithm implementation.
- **kwargs** (*Any*) –

### Methods

**build_with_dataset**(*dataset*)
Instantiate implementation object with MDPDataset object.

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

**Return type** None

**build_with_env**(*env*)
Instantiate implementation object with OpenAI Gym object.

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

**Return type** None

**collect**(*env*, *buffer=None*, *explorer=None*, *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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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 inital 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 (for Python).
- 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*, *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.

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

**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 coefficiency.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **update_actor_interval** (*int*) – interval to update policy function.
- **initial_temperature** (*float*) – initial temperature value.
- **conservative_weight** (*float*) – constant weight to scale conservative loss.
- **n_action_samples** (*int*) – the number of sampled actions to compute $\log \sum_a \exp Q(s, a)$.
- **soft_q_backup** (*bool*) – flag to use SAC-style backup.
- **dynamics** (*d3rlpy.dynamics.DynamicsBase*) – dynamics object.
- **rollout_interval** (*int*) – the number of steps before rollout.
- **rollout_horizon** (*int*) – the rollout step length.
- **rollout_batch_size** (*int*) – the number of initial transitions for rollout.
- **real_ratio** (*float*) – the real of dataset samples in a mini-batch.
- **generated_maxlen** (*int*) – the maximum number of generated samples.
- **use_gpu** (*bool, int or d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.
- **scaler** (*d3rlpy.preprocessing.Scaler or str*) – preprocessor. The available options are *['pixel', 'min_max', 'standard']*.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler or str*) – action preprocessor. The available options are `['min_max']`.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler or str*) – reward preprocessor. The available options are `['clip', 'min_max', 'standard']`.
- **impl** (*d3rlpy.algos.torch.combo_impl.COMBOImpl*) – algorithm implementation.
- **kwargs** (*Any*) –

### Methods

**build_with_dataset**(*dataset*)
Instantiate implementation object with MDPDataset object.

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

**Return type** None

**build_with_env**(*env*)
Instantiate implementation object with OpenAI Gym object.

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

**Return type** None

**collect**(*env*, *buffer=None*, *explorer=None*, *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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.

- **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 inital 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 (for Python).
- 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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.

- **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 inital 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 (for Python).
- 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]

### 4.1.2 Discrete control algorithms

| | |
|---|---|
| *d3rlpy.algos.DiscreteBC* | Behavior Cloning algorithm for discrete control. |
| *d3rlpy.algos.NFQ* | Neural Fitted Q Iteration algorithm. |
| *d3rlpy.algos.DQN* | Deep Q-Network algorithm. |
| *d3rlpy.algos.DoubleDQN* | Double Deep Q-Network algorithm. |
| *d3rlpy.algos.DiscreteSAC* | Soft Actor-Critic algorithm for discrete action-space. |
| *d3rlpy.algos.DiscreteBCQ* | Discrete version of Batch-Constrained Q-learning algorithm. |
| *d3rlpy.algos.DiscreteCQL* | Discrete version of Conservative Q-Learning algorithm. |
| *d3rlpy.algos.DiscreteRandomPolicy* | 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.

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

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

**Parameters**

- **learning_rate** (*float*) – learing rate.
- **optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory.
- **encoder_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **beta** (*float*) – reguralization factor.
- **use_gpu** (*bool, int or d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.
- **scaler** (*d3rlpy.preprocessing.Scaler or str*) – preprocessor. The available options are *['pixel', 'min_max', 'standard']*
- **impl** (*d3rlpy.algos.torch.bc_impl.DiscreteBCImpl*) – implemenation of the algorithm.
- **kwargs** (*Any*) –

### Methods

**build_with_dataset**(*dataset*)
Instantiate implementation object with MDPDataset object.

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

**Return type** None

**build_with_env**(*env*)
Instantiate implementation object with OpenAI Gym object.

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

**Return type** None

**collect**(*env*, *buffer=None*, *explorer=None*, *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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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*)
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 inital 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 (for Python).
- 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.NFQ

class d3rlpy.algos.**NFQ**(*, *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*, *use_gpu=False*, *scaler=None*, *reward_scaler=None*, *impl=None*, ***kwargs*)

Neural Fitted Q Iteration algorithm.

This NFQ implementation in d3rlpy is practically same as DQN, but excluding the target network mechanism.

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

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

### References

- Riedmiller., Neural Fitted Q Iteration - first experiences with a data efficient neural reinforcement learning method.

**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.
- **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=100000)

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

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

**Return type** None

**copy_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 algorithmn
```

(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(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_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 explortion.
- **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,)
```

(continues on next page)

(continued from previous page)

```
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 inital 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 (for Python).
- 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_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_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 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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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
```

(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_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 explortion.
- **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 inital 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 (for Python).
- 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_update_interval=8000*, *use_gpu=False*, *scaler=None*, *reward_scaler=None*, *impl=None*, ***kwargs*)

Double Deep Q-Network algorithm.

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

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

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

#### References

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

**Parameters**

- **learning_rate** (*float*) – learning rate.
- **optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory.
- **encoder_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory or str*) – Q function factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **n_steps** (*int*) – N-step TD calculation.
- **gamma** (*float*) – discount factor.
- **n_critics** (*int*) – the number of Q functions.
- **target_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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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 inital 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 (for Python).
- 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*, *actor_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam', betas=(0.9, 0.999), eps=0.0001, weight_decay=0, amsgrad=False)*, *critic_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam', betas=(0.9, 0.999), eps=0.0001, weight_decay=0, amsgrad=False)*, *temp_optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam', betas=(0.9, 0.999), eps=0.0001, weight_decay=0, amsgrad=False)*, *actor_encoder_factory='default'*, *critic_encoder_factory='default'*, *q_func_factory='mean'*, *batch_size=64*, *n_frames=1*, *n_steps=1*, *gamma=0.99*, *n_critics=2*, *initial_temperature=1.0*, *target_update_interval=8000*, *use_gpu=False*, *scaler=None*, *reward_scaler=None*, *impl=None*, ***kwargs*)

Soft Actor-Critic algorithm for discrete action-space.

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

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

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

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

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

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

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

#### References

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

**Parameters**

- **actor_learning_rate** (*float*) – learning rate for policy function.
- **critic_learning_rate** (*float*) – learning rate for Q functions.
- **temp_learning_rate** (*float*) – learning rate for temperature parameter.
- **actor_optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory for the actor.
- **critic_optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory for the critic.
- **temp_optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory for the temperature.
- **actor_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory for the critic.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory or str*) – Q function factory.

- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **n_steps** (*int*) – N-step TD calculation.
- **gamma** (*float*) – discount factor.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **initial_temperature** (*float*) – initial temperature value.
- **use_gpu** (*bool, int or d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.
- **scaler** (*d3rlpy.preprocessing.Scaler or str*) – preprocessor. The available options are *['pixel', 'min_max', 'standard']*
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler or str*) – reward preprocessor. The available options are `['clip', 'min_max', 'standard']`.
- **impl** (*d3rlpy.algos.torch.sac_impl.DiscreteSACImpl*) – algorithm implementation.
- **target_update_interval** (*int*) –
- **kwargs** (*Any*) –

### Methods

**build_with_dataset**(*dataset*)
Instantiate implementation object with MDPDataset object.

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

**Return type** None

**build_with_env**(*env*)
Instantiate implementation object with OpenAI Gym object.

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

**Return type** None

**collect**(*env*, *buffer=None*, *explorer=None*, *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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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 inital 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 (for Python).
- 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*, *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.

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

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

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

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.
- **action_flexibility** (*float*) – probability threshold represented as $\tau$.
- **beta** (*float*) – reguralization term for imitation function.
- **target_update_interval** (*int*) – interval to update the target network.
- **use_gpu** (*bool, int or d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.
- **scaler** (*d3rlpy.preprocessing.Scaler or str*) – preprocessor. The available options are *['pixel', 'min_max', 'standard']*
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler or str*) – reward preprocessor. The available options are `['clip', 'min_max', 'standard']`.
- **impl** (*d3rlpy.algos.torch.bcq_impl.DiscreteBCQImpl*) – algorithm implementation.
- **kwargs** (*Any*) –

### Methods

**build_with_dataset**(*dataset*)
Instantiate implementation object with MDPDataset object.

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

**Return type** None

**build_with_env**(*env*)
Instantiate implementation object with OpenAI Gym object.

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

**Return type** None

**collect**(*env*, *buffer=None*, *explorer=None*, *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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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)
```

(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_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 explortion.
- **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 inital 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 (for Python).
- 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_update_interval=8000*, *alpha=1.0*, *use_gpu=False*, *scaler=None*, *reward_scaler=None*, *impl=None*, ***kwargs*)

Discrete version of Conservative Q-Learning algorithm.

Discrete version of CQL is a DoubleDQN-based data-driven deep reinforcement learning algorithm (the original paper uses DQN), which achieves state-of-the-art performance in offline RL problems.

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

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

#### References

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

**Parameters**

- **learning_rate** (*float*) – learning rate.
- **optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – optimizer factory.
- **encoder_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory or str*) – Q function factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **n_steps** (*int*) – N-step TD calculation.
- **gamma** (*float*) – discount factor.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **target_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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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
```

(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 inital 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 (for Python).
- 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=100000)

# transfer to online algorithm
```

(continues on next page)

(continued from previous page)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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 inital 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 (for Python).
- 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]

## 4.2 Q Functions

d3rlpy provides various Q functions including state-of-the-arts, which are internally used in algorithm objects. You can switch Q functions by passing `q_func_factory` argument at algorithm initialization.

```
from d3rlpy.algos import CQL

cql = CQL(q_func_factory='qr') # use Quantile Regression Q function
```

Also you can change hyper parameters.

```
from d3rlpy.models.q_functions import QRQFunctionFactory

q_func = QRQFunctionFactory(n_quantiles=32)

cql = CQL(q_func_factory=q_func)
```

The default Q function is `mean` approximator, which estimates expected scalar action-values. However, in recent advancements of deep reinforcement learning, the new type of action-value approximators has been proposed, which is called *distributional* Q functions.

Unlike the `mean` approximator, the *distributional* Q functions estimate distribution of action-values. This *distributional* approaches have shown consistently much stronger performance than the `mean` approximator.

Here is a list of available Q functions in the order of performance ascendingly. Currently, as a trade-off between performance and computational complexity, the higher performance requires the more expensive computational costs.

| | |
|---|---|
| *d3rlpy.models.q_functions.MeanQFunctionFactory* | Standard Q function factory class. |
| *d3rlpy.models.q_functions.QRQFunctionFactory* | Quantile Regression Q function factory class. |
| *d3rlpy.models.q_functions.IQNQFunctionFactory* | Implicit Quantile Network Q function factory class. |
| *d3rlpy.models.q_functions.FQFQFunctionFactory* | Fully parameterized Quantile Function Q function factory. |

### 4.2.1 d3rlpy.models.q_functions.MeanQFunctionFactory

**class** d3rlpy.models.q_functions.**MeanQFunctionFactory**(*share_encoder=False*, ***kwargs*)

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**

- **share_encoder** (*bool*) – flag to share encoder over multiple Q functions.
- **kwargs** (*Any*) –

### 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'**

**share_encoder**

### 4.2.2 d3rlpy.models.q_functions.QRQFunctionFactory

**class** d3rlpy.models.q_functions.**QRQFunctionFactory**(*share_encoder=False*, *n_quantiles=32*, ***kwargs*)

Quantile Regression Q function factory class.

#### References

- Dabney et al., Distributional reinforcement learning with quantile regression.

**Parameters**

- **share_encoder** (*bool*) – flag to share encoder over multiple Q functions.
- **n_quantiles** (*int*) – the number of quantiles.
- **kwargs** (*Any*) –

#### 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'**

**n_quantiles**

**share_encoder**

### 4.2.3 d3rlpy.models.q_functions.IQNQFunctionFactory

**class** d3rlpy.models.q_functions.**IQNQFunctionFactory**(*share_encoder=False*, *n_quantiles=64*, *n_greedy_quantiles=32*, *embed_size=64*, ***kwargs*)

Implicit Quantile Network Q function factory class.

#### References

- Dabney et al., Implicit quantile networks for distributional reinforcement learning.

**Parameters**

- **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.
- **kwargs** (*Any*) –

#### 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'**

**embed_size**

**n_greedy_quantiles**

**n_quantiles**

**share_encoder**

### 4.2.4 d3rlpy.models.q_functions.FQFQFunctionFactory

**class** d3rlpy.models.q_functions.**FQFQFunctionFactory**(*share_encoder=False*, *n_quantiles=32*, *embed_size=64*, *entropy_coeff=0.0*, ***kwargs*)
Fully parameterized Quantile Function Q function factory.

#### References

- Yang et al., Fully parameterized quantile function for distributional reinforcement learning.

**Parameters**

- **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 coefficiency of entropy penalty term.
- **kwargs** (*Any*) –

### 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'**

**embed_size**

**entropy_coeff**

**n_quantiles**

**share_encoder**

## 4.3 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 $(s_t, a_t, r_t, s_{t+1})$ 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].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, ...]  # r1 = r(s1, a1)
terminals    = [t1, t2, t3, ...]  # t1 = t(s1, a1)
```

| | |
|---|---|
| *d3rlpy.dataset.MDPDataset* | Markov-Decision Process Dataset class. |
| *d3rlpy.dataset.Episode* | Episode class. |
| *d3rlpy.dataset.Transition* | Transition class. |
| *d3rlpy.dataset.TransitionMiniBatch* | mini-batch of Transition objects. |

### 4.3.1 d3rlpy.dataset.MDPDataset

**class** d3rlpy.dataset.**MDPDataset**(*observations*, *actions*, *rewards*, *terminals*, *episode_terminals=None*, *discrete_action=None*)

Markov-Decision Process Dataset class.

MDPDataset is deisnged for reinforcement learning datasets to use them like supervised learning datasets.

```
from d3rlpy.dataset import MDPDataset

# 1000 steps of observations with shape of (100,)
observations = np.random.random((1000, 100))
# 1000 steps of actions with shape of (4,)
actions = np.random.random((1000, 4))
# 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, 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, dim_action)*. If the action-space is discrete, the shape should be *(N,)*.
- **rewards** (*numpy.ndarray*) – array of scalar rewards. The reward function should be defined as $r_t = r(s_t, a_t)$.
- **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.

### Methods

**__getitem__**(*index*)

**__len__**()

**__iter__**()

**append**(*observations*, *actions*, *rewards*, *terminals*, *episode_terminals=None*)
Appends new data.

**Parameters**

- **observations** (*numpy.ndarray*) – N-D array.
- **actions** (*numpy.ndarray*) – actions.
- **rewards** (*numpy.ndarray*) – rewards.
- **terminals** (*numpy.ndarray*) – terminals.
- **episode_terminals** (*numpy.ndarray*) – episode terminals.

**build_episodes**()
Builds episode objects.

This method will be internally called when accessing the episodes property at the first time.

**compute_stats**()
Computes statistics of the dataset.

```
stats = dataset.compute_stats()

# return statistics
stats['return']['mean']
stats['return']['std']
stats['return']['min']
stats['return']['max']

# reward statistics
stats['reward']['mean']
stats['reward']['std']
stats['reward']['min']
stats['reward']['max']

# action (only with continuous control actions)
stats['action']['mean']
stats['action']['std']
stats['action']['min']
stats['action']['max']
```

(continues on next page)

(continued from previous page)

```
# observation (only with numpy.ndarray observations)
stats['observation']['mean']
stats['observation']['std']
stats['observation']['min']
stats['observation']['max']
```

**Returns** statistics of the dataset.

**Return type** dict

**dump**(*fname*)
Saves dataset as HDF5.

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

**extend**(*dataset*)
Extend dataset by another dataset.

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

**get_action_size**()
Returns dimension of action-space.

If *discrete_action=True*, the return value will be the maximum index +1 in the give actions.

**Returns** dimension of action-space.

**Return type** int

**get_observation_shape**()
Returns observation shape.

**Returns** observation shape.

**Return type** tuple

**is_action_discrete**()
Returns *discrete_action* flag.

**Returns** *discrete_action* flag.

**Return type** bool

**classmethod load**(*fname*)
Loads dataset from HDF5.

```
import numpy as np
from d3rlpy.dataset import MDPDataset

dataset = MDPDataset(np.random.random(10, 4),
                     np.random.random(10, 2),
                     np.random.random(10),
                     np.random.randint(2, size=10))

# save as HDF5
dataset.dump('dataset.h5')

# load from HDF5
new_dataset = MDPDataset.load('dataset.h5')
```

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

**size**()
Returns the number of episodes in the dataset.

**Returns** the number of episodes.

**Return type** int

### Attributes

**actions**
Returns the actions.

**Returns** array of actions.

**Return type** numpy.ndarray

**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

### 4.3.2 d3rlpy.dataset.Episode

class d3rlpy.dataset.**Episode**(*observation_shape*, *action_size*, *observations*, *actions*, *rewards*, *terminal=True*)

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).

#### 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*)

### 4.3.3 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*.
- **terminal** (*int*) – terminal flag at *t+1*.
- **prev_transition** (d3rlpy.dataset.Transition) – pointer to the previous transition.
- **next_transition** (d3rlpy.dataset.Transition) – pointer to the next transition.

#### Methods

**clear_links**()

Clears links to the next and previous transitions.

This method is necessary to call when freeing this instance by GC.

**get_action_size**()

Returns dimension of action-space.

**Returns** dimension of action-space.

**Return type** int

**get_observation_shape**()

Returns observation shape.

**Returns** observation shape.

**Return type** tuple

#### Attributes

**action**

Returns action at *t*.

**Returns** action at *t*.

**Return type** (numpy.ndarray or int)

**is_discrete**

Returns flag of discrete action-space.

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

**Return type** bool

**next_observation**
Returns observation at *t+1*.

**Returns** observation at *t+1*.

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

**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

### 4.3.4 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)
```

(continues on next page)

(continued from previous page)

```
# 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).

**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

**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_observations**
Returns mini-batch of observations at *t*+*n*.

**Returns** observations at *t*+*n*.

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

**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*

## 4.4 Datasets

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

| | |
|---|---|
| *d3rlpy.datasets.get_cartpole* | Returns cartpole dataset and environment. |
| *d3rlpy.datasets.get_pendulum* | Returns pendulum dataset and environment. |
| *d3rlpy.datasets.get_atari* | Returns atari dataset and envrironment. |
| *d3rlpy.datasets.get_atari_transitions* | Returns atari dataset as a list of Transition objects and envrironment. |
| *d3rlpy.datasets.get_d4rl* | Returns d4rl dataset and envrironment. |
| *d3rlpy.datasets.get_dataset* | Returns dataset and envrironment by guessing from name. |

### 4.4.1 d3rlpy.datasets.get_cartpole

d3rlpy.datasets.**get_cartpole**(*dataset_type='replay'*)
Returns cartpole dataset and environment.

The dataset is automatically downloaded to `d3rlpy_data/cartpole.h5` if it does not exist.

**Parameters** **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]

### 4.4.2 d3rlpy.datasets.get_pendulum

d3rlpy.datasets.**get_pendulum**(*dataset_type='replay'*)

Returns pendulum dataset and environment.

The dataset is automatically downloaded to `d3rlpy_data/pendulum.h5` if it does not exist.

**Parameters** **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]

### 4.4.3 d3rlpy.datasets.get_atari

d3rlpy.datasets.**get_atari**(*env_name*)

Returns atari dataset and envrironment.

The dataset is provided through d4rl-atari. See more details including available dataset from its GitHub page.

```
from d3rlpy.datasets import get_atari

dataset, env = get_atari('breakout-mixed-v0')
```

#### References

- https://github.com/takuseno/d4rl-atari

**Parameters** **env_name** (*str*) – environment id of d4rl-atari dataset.

**Returns** tuple of *d3rlpy.dataset.MDPDataset* and gym environment.

**Return type** Tuple[*d3rlpy.dataset.MDPDataset*, gym.core.Env]

### 4.4.4 d3rlpy.datasets.get_atari_transitions

d3rlpy.datasets.**get_atari_transitions**(*game_name*, *fraction=0.01*, *index=0*)

Returns atari dataset as a list of Transition objects and envrironment.

The dataset is provided through d4rl-atari. The difference from `get_atari` function is that this function will sample transitions from all epochs. This function is necessary for reproducing Atari experiments.

```
from d3rlpy.datasets import get_atari_transitions

# get 1% of transitions from all epochs (1M x 50 epoch x 1% = 0.5M)
dataset, env = get_atari_transitions('breakout', fraction=0.01)
```

#### References

- https://github.com/takuseno/d4rl-atari

**Parameters**

- **game_name** (*str*) – Atari 2600 game name in lower_snake_case.
- **fraction** (*float*) – fraction of sampled transitions.
- **index** (*int*) – index to specify which trial to load.

**Returns** tuple of a list of *d3rlpy.dataset.Transition* and gym environment.

**Return type** Tuple[List[*d3rlpy.dataset.Transition*], gym.core.Env]

### 4.4.5 d3rlpy.datasets.get_d4rl

d3rlpy.datasets.**get_d4rl**(*env_name*)

Returns d4rl dataset and envrironment.

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.

**Returns** tuple of *d3rlpy.dataset.MDPDataset* and gym environment.

**Return type** Tuple[*d3rlpy.dataset.MDPDataset*, gym.core.Env]

### 4.4.6 d3rlpy.datasets.get_dataset

d3rlpy.datasets.**get_dataset**(*env_name*)

Returns dataset and envrironment 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-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.

**Returns** tuple of *d3rlpy.dataset.MDPDataset* and gym environment.

**Return type** Tuple[*d3rlpy.dataset.MDPDataset*, gym.core.Env]

## 4.5 Preprocessing

### 4.5.1 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)

# preprocesing 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)
```

| | |
|---|---|
| *d3rlpy.preprocessing.PixelScaler* | Pixel normalization preprocessing. |
| *d3rlpy.preprocessing.MinMaxScaler* | Min-Max normalization preprocessing. |
| *d3rlpy.preprocessing.StandardScaler* | 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
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'**

### 4.5.2 Action

d3rlpy also provides the feature that preprocesses continuous action. With this preprocessing, you don't need to normalize actions in advance or implement normalization in the environment side.

```
from d3rlpy.algos import CQL
from d3rlpy.dataset import MDPDataset

dataset = MDPDataset(...)

# 'min_max' or None
cql = CQL(action_scaler='min_max')

# action scaler is fitted from the given episodes
cql.fit(dataset.episodes)

# postprocessing is included in TorchScript
cql.save_policy('policy.pt')

# you don't need to take care of postprocessing at production
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 = []
```

(continues on next page)

(continued from previous page)

```
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'**

### 4.5.3 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=...)
```

(continues on next page)

(continued from previous page)

```
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)
```

| | |
|---|---|
| *d3rlpy.preprocessing.MinMaxRewardScaler* | Min-Max reward normalization preprocessing. |
| *d3rlpy.preprocessing.StandardRewardScaler* | Reward standardization preprocessing. |
| *d3rlpy.preprocessing.ClipRewardScaler* | Reward clipping preprocessing. |
| *d3rlpy.preprocessing.MultiplyRewardScaler* | Multiplication reward preprocessing. |
| *d3rlpy.preprocessing.ReturnBasedRewardScaler* | Reward normalization preprocessing based on return scale. |

### 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'**

## d3rlpy.preprocessing.ReturnBasedRewardScaler

**class** d3rlpy.preprocessing.**ReturnBasedRewardScaler**(*dataset=None*, *return_max=None*, *return_min=None*, *multiplier=1.0*)
Reward normalization preprocessing based on return scale.

$$r' = r/(R_{max} - R_{min})$$

```
from d3rlpy.algos import CQL

cql = CQL(reward_scaler="return")
```

You can also initialize with *d3rlpy.dataset.MDPDataset* object or manually.

```
from d3rlpy.preprocessing import ReturnBasedRewardScaler

# initialize with dataset
scaler = ReturnBasedRewardScaler(dataset)

# initialize manually
scaler = ReturnBasedRewardScaler(return_max=100.0, return_min=1.0)

cql = CQL(scaler=scaler)
```

### References

- Kostrikov et al., Offline Reinforcement Learning with Implicit Q-Learning.

**Parameters**

- **dataset** (d3rlpy.dataset.MDPDataset) – dataset object.
- **return_max** (*float*) – the maximum return value.
- **return_min** (*float*) – standard deviation 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] = 'return'**

## 4.6 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 alises.

```
from d3rlpy.models.optimizers import AdamFactory

# alias for Adam optimizer
optim_factory = AdamFactory(weight_decay=1e-4)

dqn = DQN(optim_factory=optim_factory)
```

| | |
|---|---|
| *d3rlpy.models.optimizers.OptimizerFactory* | A factory class that creates an optimizer object in a lazy way. |
| *d3rlpy.models.optimizers.SGDFactory* | An alias for SGD optimizer. |
| *d3rlpy.models.optimizers.AdamFactory* | An alias for Adam optimizer. |
| *d3rlpy.models.optimizers.RMSpropFactory* | An alias for RMSprop optimizer. |

### 4.6.1 d3rlpy.models.optimizers.OptimizerFactory

class d3rlpy.models.optimizers.**OptimizerFactory**(*optim_cls*, ***kwargs*)

A factory class that creates an optimizer object in a lazy way.

The optimizers in algorithms can be configured through this factory class.

```
from torch.optim Adam
from d3rlpy.optimizers import OptimizerFactory
from d3rlpy.algos import DQN

factory = OptimizerFactory(Adam, eps=0.001)

dqn = DQN(optim_factory=factory)
```

**Parameters**

- **optim_cls** (*Union[Type[torch.optim.optimizer.Optimizer], str]*) – An optimizer class.
- **kwargs** (*Any*) – arbitrary keyword-arguments.

#### Methods

**create**(*params*, *lr*)

Returns an optimizer object.

**Parameters**

- **params** (*list*) – a list of PyTorch parameters.
- **lr** (*float*) – learning rate.

**Returns** an optimizer object.

**Return type** torch.optim.Optimizer

**get_params**(*deep=False*)

Returns optimizer parameters.

**Parameters** **deep** (*bool*) – flag to deeply copy the parameters.

**Returns** optimizer parameters.

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

### 4.6.2 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]

### 4.6.3 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]

## 4.6.4 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]

## 4.7 Network Architectures

In d3rlpy, the neural network architecture is automatically selected based on observation shape. If the observation is image, the algorithm uses the `Nature DQN`-based encoder at each function. Otherwise, the standard MLP architecture that consists with two linear layers with `256` hidden units.

Furthermore, d3rlpy provides `EncoderFactory` that gives you flexible control over this neural netowrk architectures.

```
from d3rlpy.algos import DQN
from d3rlpy.models.encoders import VectorEncoderFactory

# encoder factory
encoder_factory = VectorEncoderFactory(hidden_units=[300, 400], activation='tanh')

# set EncoderFactory
dqn = DQN(encoder_factory=encoder_factory)
```

You can also build your own encoder factory.

```
import torch
import torch.nn as nn

from d3rlpy.models.encoders import EncoderFactory

# your own neural network
class CustomEncoder(nn.Module):
    def __init__(self, obsevation_shape, feature_size):
        self.feature_size = feature_size
        self.fc1 = nn.Linear(observation_shape[0], 64)
        self.fc2 = nn.Linear(64, feature_size)

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

(continues on next page)

(continued from previous page)

```
    def __init__(self, feature_size):
        self.feature_size = feature_size

    def create(self, observation_shape):
        return CustomEncoder(observation_shape, self.feature_size)

    def get_params(self, deep=False):
        return {'feature_size': self.feature_size}

dqn = DQN(encoder_factory=CustomEncoderFactory(feature_size=64))
```

You can also define action-conditioned networks such as Q-functions for continuous controls. `create` or `create_with_action` will be called depending on the function.

```
class CustomEncoderWithAction(nn.Module):
    def __init__(self, obsevation_shape, action_size, feature_size):
        self.feature_size = feature_size
        self.fc1 = nn.Linear(observation_shape[0] + action_size, 64)
        self.fc2 = nn.Linear(64, feature_size)

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

    def get_feature_size(self):
        return self.feature_size

class CustomEncoderFactory(EncoderFactory):
    TYPE = 'custom' # this is necessary

    def __init__(self, feature_size):
        self.feature_size = feature_size

    def create(self, observation_shape):
        return CustomEncoder(observation_shape, self.feature_size)

    def create_with_action(observation_shape, action_size, discrete_action):
        return CustomEncoderWithAction(observation_shape, action_size, self.feature_size)

    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')
```

| | |
|---|---|
| *d3rlpy.models.encoders.DefaultEncoderFactory* | Default encoder factory class. |
| *d3rlpy.models.encoders.PixelEncoderFactory* | Pixel encoder factory class. |
| *d3rlpy.models.encoders.VectorEncoderFactory* | Vector encoder factory class. |
| *d3rlpy.models.encoders.DenseEncoderFactory* | DenseNet encoder factory class. |

### 4.7.1 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 enocder module.

**Parameters** **observation_shape** (*Sequence[int]*) – observation shape.

**Returns** an enocder object.

**Return type** d3rlpy.models.torch.encoders.Encoder

**create_with_action**(*observation_shape*, *action_size*, *discrete_action=False*)

Returns PyTorch's state-action enocder 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 enocder 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'**

### 4.7.2 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 enocder module.

**Parameters** **observation_shape** (*Sequence[int]*) – observation shape.

**Returns** an enocder object.

**Return type** d3rlpy.models.torch.encoders.PixelEncoder

**create_with_action**(*observation_shape*, *action_size*, *discrete_action=False*)
Returns PyTorch's state-action enocder 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 enocder 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'**

### 4.7.3 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 enocder module.

**Parameters** **observation_shape** (*Sequence[int]*) – observation shape.

**Returns** an enocder object.

**Return type** d3rlpy.models.torch.encoders.VectorEncoder

**create_with_action**(*observation_shape*, *action_size*, *discrete_action=False*)
Returns PyTorch's state-action enocder 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 enocder 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'**

## 4.7.4 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 enocder module.

**Parameters** **observation_shape** (*Sequence[int]*) – observation shape.

**Returns** an enocder object.

**Return type** d3rlpy.models.torch.encoders.VectorEncoder

**create_with_action**(*observation_shape*, *action_size*, *discrete_action=False*)
Returns PyTorch's state-action enocder 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 enocder 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'**

## 4.8 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()
```

(continues on next page)

(continued from previous page)

```
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)
                        })
```

### 4.8.1 Algorithms

| | |
|---|---|
| *d3rlpy.metrics.scorer.td_error_scorer* | Returns average TD error. |
| *d3rlpy.metrics.scorer.discounted_sum_of_advantage_scorer* | Returns average of discounted sum of advantage. |
| *d3rlpy.metrics.scorer.average_value_estimation_scorer* | Returns average value estimation. |
| *d3rlpy.metrics.scorer.value_estimation_std_scorer* | Returns standard deviation of value estimation. |
| *d3rlpy.metrics.scorer.initial_state_value_estimation_scorer* | Returns mean estimated action-values at the initial states. |
| *d3rlpy.metrics.scorer.soft_opc_scorer* | Returns Soft Off-Policy Classification metrics. |
| *d3rlpy.metrics.scorer.continuous_action_diff_scorer* | Returns squared difference of actions between algorithm and dataset. |
| *d3rlpy.metrics.scorer.discrete_action_match_scorer* | Returns percentage of identical actions between algorithm and dataset. |
| *d3rlpy.metrics.scorer.evaluate_on_environment* | Returns scorer function of evaluation on environment. |
| *d3rlpy.metrics.comparer.compare_continuous_action_diff* | Returns scorer function of action difference between algorithms. |
| *d3rlpy.metrics.comparer.compare_discrete_action_match* | 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 metics suggests how Q functions overfit to training sets. If the TD error is large, the Q functions are overfitting.

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

**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 *boostrap* enabled and the larger *n_critics* at algorithm. If standard deviation of value estimation is large, the Q functions are overfitting to the training set.

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

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

**Parameters**

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

**Returns** standard deviation.

**Return type** float

### d3rlpy.metrics.scorer.initial_state_value_estimation_scorer

d3rlpy.metrics.scorer.**initial_state_value_estimation_scorer**(*algo*, *episodes*)

Returns mean estimated action-values at the initial states.

This metrics suggests how much return the trained policy would get from the initial states by deploying the policy to the states. If the estimated value is large, the trained policy is expected to get higher returns.

$$\mathbb{E}_{s_0 \sim D}[Q(s_0, \pi(s_0))]$$

#### References

- Paine et al., Hyperparameter Selection for Offline Reinforcement Learning

**Parameters**

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

**Returns** mean action-value estimation at the initial states.

**Return type** float

### d3rlpy.metrics.scorer.soft_opc_scorer

d3rlpy.metrics.scorer.**soft_opc_scorer**(*return_threshold*)
Returns Soft Off-Policy Classification metrics.

This function returns scorer function, which is suitable to the standard scikit-learn scorer function style. The metrics of the scorer funciton is evaluating gaps of action-value estimation between the success episodes and the all episodes. If the learned Q-function is optimal, action-values in success episodes are expected to be higher than the others. The success episode is defined as an episode with a return above the given threshold.

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

```
from d3rlpy.datasets import get_cartpole
from d3rlpy.algos import DQN
from d3rlpy.metrics.scorer import soft_opc_scorer
from sklearn.model_selection import train_test_split

dataset, _ = get_cartpole()
train_episodes, test_episodes = train_test_split(dataset, test_size=0.2)

scorer = soft_opc_scorer(return_threshold=180)

dqn = DQN()
dqn.fit(train_episodes,
        eval_episodes=test_episodes,
        scorers={'soft_opc': scorer})
```

#### References

- Irpan et al., Off-Policy Evaluation via Off-Policy Classification.

**Parameters** **return_threshold** (*float*) – threshold of success episodes.

**Returns** scorer function.

**Return type** Callable[[d3rlpy.metrics.scorer.AlgoProtocol, List[*d3rlpy.dataset.Episode*]], float]

### d3rlpy.metrics.scorer.continuous_action_diff_scorer

d3rlpy.metrics.scorer.**continuous_action_diff_scorer**(*algo*, *episodes*)
Returns squared difference of actions between algorithm and dataset.

This metrics suggests how different the greedy-policy is from the given episodes in continuous action-space. If the given episodes are near-optimal, the small action difference would be better.

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

**Parameters**

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

**Returns** squared action difference.

**Return type** float

### d3rlpy.metrics.scorer.discrete_action_match_scorer

d3rlpy.metrics.scorer.**discrete_action_match_scorer**(*algo*, *episodes*)
Returns percentage of identical actions between algorithm and dataset.

This metrics suggests how different the greedy-policy is from the given episodes in discrete action-space. If the given episdoes are near-optimal, the large percentage would be better.

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

**Parameters**

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

**Returns** percentage of identical actions.

**Return type** float

### d3rlpy.metrics.scorer.evaluate_on_environment

d3rlpy.metrics.scorer.**evaluate_on_environment**(*env*, *n_trials=10*, *epsilon=0.0*, *render=False*)
Returns scorer function of evaluation on environment.

This function returns scorer function, which is suitable to the standard scikit-learn scorer function style. The metrics of the scorer function is ideal metrics to evaluate the resulted policies.

```
import gym

from d3rlpy.algos import DQN
from d3rlpy.metrics.scorer import evaluate_on_environment


env = gym.make('CartPole-v0')

scorer = evaluate_on_environment(env)

cql = CQL()

mean_episode_return = scorer(cql)
```

**Parameters**

- **env** (*gym.core.Env*) – gym-styled environment.
- **n_trials** (*int*) – the number of trials.
- **epsilon** (*float*) – noise factor for epsilon-greedy policy.
- **render** (*bool*) – flag to render environment.

**Returns** scoerer 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 comapre 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.

$$\mathbb{E}_{s_t \sim D}[|| \{\text{argmax}_a Q_{\theta_1}(s_t, a) = \text{argmax}_a Q_{\theta_2}(s_t, a)\}]$$

```
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 comapre with.

**Returns** scorer function.

**Return type** Callable[[d3rlpy.metrics.scorer.AlgoProtocol, List[*d3rlpy.dataset.Episode*]], float]

### 4.8.2 Dynamics

| | |
|---|---|
| *d3rlpy.metrics.scorer.dynamics_observation_prediction_error_scorer* | Returns MSE of observation prediction. |
| *d3rlpy.metrics.scorer.dynamics_reward_prediction_error_scorer* | Returns MSE of reward prediction. |
| *d3rlpy.metrics.scorer.dynamics_prediction_variance_scorer* | 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

## 4.9 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.

### 4.9.1 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=100000)

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

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

**Return type** None

**copy_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 algorithmn
sac = d3rlpy.algos.SAC()
sac.create_impl(cql.observation_shape, cql.action_size)
sac.copy_q_function_from(cql)
```

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

**Return type** None

**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_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 explortion.
- **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
```

(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 inital 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 (for Python).
- 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]

### 4.9.2 For discrete control algorithms

| | |
|---|---|
| *d3rlpy.ope.DiscreteFQE* | Fitted Q Evaluation for discrete action-space. |

#### d3rlpy.ope.DiscreteFQE

**class** d3rlpy.ope.**DiscreteFQE**(*, *algo=None*, *learning_rate=0.0001*, *optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='Adam', betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)*, *encoder_factory='default'*, *q_func_factory='mean'*, *batch_size=100*, *n_frames=1*, *n_steps=1*, *gamma=0.99*, *n_critics=1*, *target_update_interval=100*, *use_gpu=False*, *scaler=None*, *action_scaler=None*, *reward_scaler=None*, *impl=None*, ***kwargs*)

Fitted Q Evaluation for discrete action-space.

FQE is an off-policy evaluation method that approximates a Q function $Q_\theta(s, a)$ with the trained policy $\pi_\phi(s)$.

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

The trained Q function in FQE will estimate evaluation metrics more accurately than learned Q function during training.

##### References

- Le et al., Batch Policy Learning under Constraints.

**Parameters**

- **algo** (*d3rlpy.algos.base.AlgoBase*) – algorithm to evaluate.
- **learning_rate** (*float*) – learning rate.
- **optim_factory** (d3rlpy.models.optimizers.OptimizerFactory *or* *str*) – optimizer factory.
- **encoder_factory** (*d3rlpy.models.encoders.EncoderFactory or str*) – encoder factory.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory or str*) – Q function factory.
- **batch_size** (*int*) – mini-batch size.
- **n_frames** (*int*) – the number of frames to stack for image observation.
- **n_steps** (*int*) – N-step TD calculation.
- **gamma** (*float*) – discount factor.
- **n_critics** (*int*) – the number of Q functions for ensemble.
- **target_update_interval** (*int*) – interval to update the target network.
- **use_gpu** (*bool, int or d3rlpy.gpu.Device*) – flag to use GPU, device ID or device.
- **scaler** (*d3rlpy.preprocessing.Scaler or str*) – preprocessor. The available options are *['pixel', 'min_max', 'standard']*

- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler or str*) – reward preprocessor. The available options are `['clip', 'min_max', 'standard']`.
- **impl** (*d3rlpy.metrics.ope.torch.FQEImpl*) – algorithm implementation.
- **action_scaler** (*Optional[Union[d3rlpy.preprocessing.action_scalers.ActionScaler, str]]*) –
- **kwargs** (*Any*) –

### Methods

**build_with_dataset**(*dataset*)
Instantiate implementation object with MDPDataset object.

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

**Return type** None

**build_with_env**(*env*)
Instantiate implementation object with OpenAI Gym object.

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

**Return type** None

**collect**(*env*, *buffer=None*, *explorer=None*, *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=100000)

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

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

**Return type** None

**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_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 explortion.
- **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 inital 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 (for Python).
- 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]

## 4.10 Save and Load

### 4.10.1 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')
```

### 4.10.2 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 beggining 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')
```

### 4.10.3 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.

## 4.11 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)
```

### 4.11.1 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')
```

## 4.12 Online Training

### 4.12.1 Standard Training

d3rlpy provides not only offline training, but also online training utilities. Despite being designed for offline training algorithms, d3rlpy is flexible enough to be trained in an online manner with a few more utilities.

```
import gym

from d3rlpy.algos import DQN
from d3rlpy.online.buffers import ReplayBuffer
from d3rlpy.online.explorers import LinearDecayEpsilonGreedy

# setup environment
env = gym.make('CartPole-v0')
eval_env = gym.make('CartPole-v0')

# setup algorithm
dqn = DQN(batch_size=32,
          learning_rate=2.5e-4,
          target_update_interval=100,
          use_gpu=True)

# setup replay buffer
buffer = ReplayBuffer(maxlen=1000000, env=env)

# setup explorers
explorer = LinearDecayEpsilonGreedy(start_epsilon=1.0,
                                    end_epsilon=0.1,
                                    duration=10000)

# start training
dqn.fit_online(env,
               buffer,
               explorer=explorer, # you don't need this with probablistic policy␣
↪algorithms
               eval_env=eval_env,
               n_steps=30000, # the number of total steps to train.
```

(continues on next page)

(continued from previous page)

```
            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*)

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.

##### 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

| | |
|---|---|
| *d3rlpy.online.explorers.ConstantEpsilonGreedy* | $\epsilon$-greedy explorer with constant $\epsilon$. |
| *d3rlpy.online.explorers.LinearDecayEpsilonGreedy* | $\epsilon$-greedy explorer with linear decay schedule. |
| *d3rlpy.online.explorers.NormalNoise* | 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

## 4.13 (experimental) Model-based Algorithms

d3rlpy provides model-based reinforcement learning algorithms.

```
from d3rlpy.datasets import get_pendulum
from d3rlpy.dynamics import ProbabilisticEnsembleDynamics
from d3rlpy.metrics.scorer import dynamics_observation_prediction_error_scorer
from d3rlpy.metrics.scorer import dynamics_reward_prediction_error_scorer
from d3rlpy.metrics.scorer import dynamics_prediction_variance_scorer
from sklearn.model_selection import train_test_split

dataset, _ = get_pendulum()

train_episodes, test_episodes = train_test_split(dataset)
```

(continues on next page)

(continued from previous page)

```
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)
```

### 4.13.1 Dynamics Model

| | |
|---|---|
| *d3rlpy.dynamics.ProbabilisticEnsembleDynamics* | Probabilistic ensemble dynamics. |

#### d3rlpy.dynamics.ProbabilisticEnsembleDynamics

**class** d3rlpy.dynamics.**ProbabilisticEnsembleDynamics**(**, learning_rate=0.001, optim_factory=d3rlpy.models.optimizers.AdamFactory(optim_cls='A 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 $N$ probablistic 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]

## 4.14 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.

### 4.14.1 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 CQL
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 = CQL()
offline_model.fit(dataset.episodes, n_epochs=100)
```

### 4.14.2 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
```

CHAPTER

FIVE

# COMMAND LINE INTERFACE

d3rlpy provides the convenient CLI tool.

## 5.1 plot

Plot the saved metrics by specifying paths:

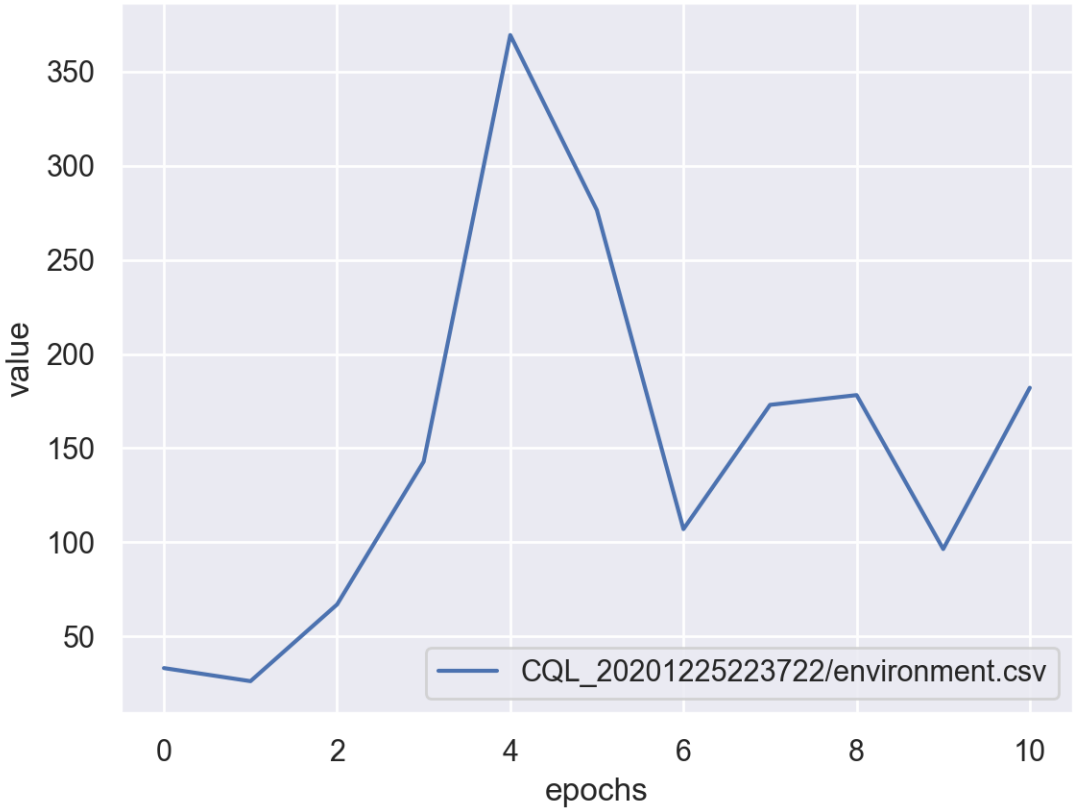
```
$ d3rlpy plot <path> [<path>...]
```

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
```



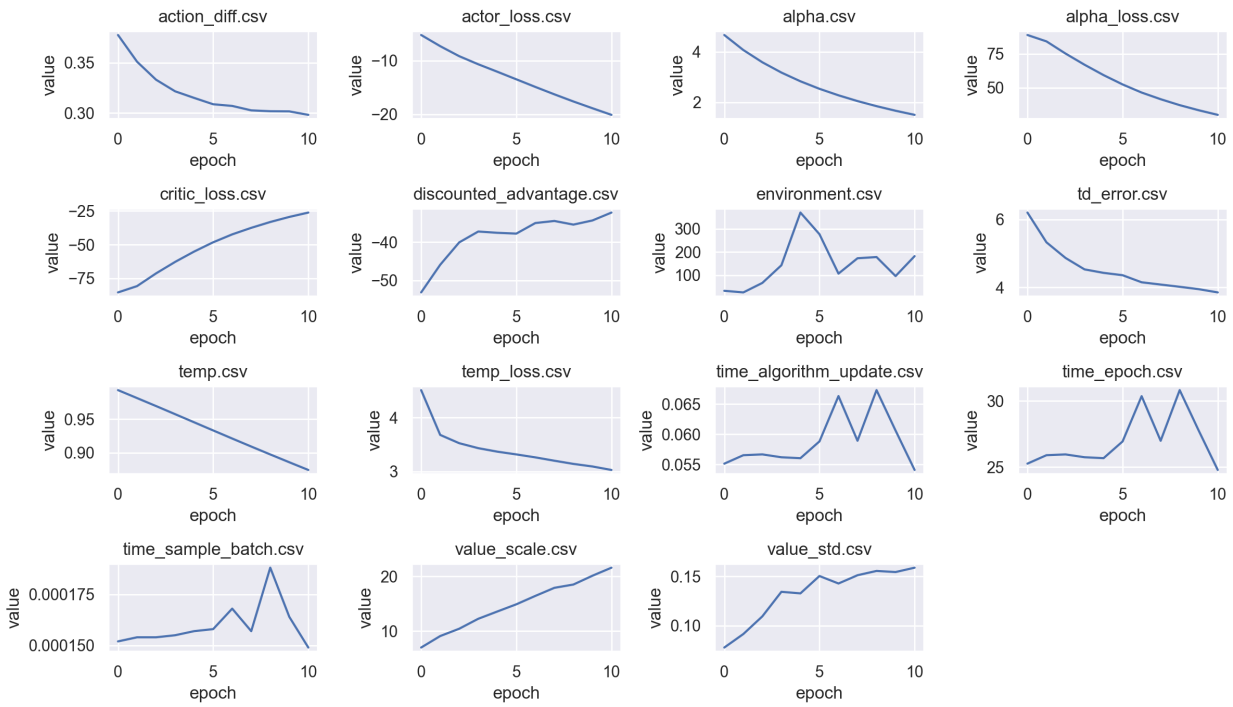

## 5.2 plot-all

Plot the all metrics saved in the directory:

```
$ d3rlpy plot-all <path>
```

example:

```
$ d3rlpy plot-all d3rlpy_logs/CQL_20201224224314
```




## 5.3 export

Export the saved model to the inference format, `onnx` and `torchscript`:

```
$ d3rlpy export <path>
```

Table 2: options

| option | description |
|---|---|
| `--format` | model format (torchscript, onnx). |
| `--params-json` | explicitly specify params.json. |
| `--out` | output path. |

example:

```
$ d3rlpy export d3rlpy_logs/CQL_20201224224314/model_100.pt
```

## 5.4 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)'
```

## 5.5 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)'
```

CHAPTER

SIX

# INSTALLATION

## 6.1 Recommended Platforms

d3rlpy supports Linux, macOS and also Windows.

## 6.2 Install d3rlpy

### 6.2.1 Install via PyPI

*pip* is a recommended way to install d3rlpy:

```
$ pip install d3rlpy
```

### 6.2.2 Install via Anaconda

d3rlpy is also available on *conda-forge*:

```
$ conda install -c conda-forge d3rlpy
```

### 6.2.3 Install via Docker

d3rlpy is also available on Docker Hub:

```
$ docker run -it --gpus all --name d3rlpy takuseno/d3rlpy:latest bash
```

### 6.2.4 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 .
```

CHAPTER

SEVEN

# TIPS

## 7.1 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)
```

## 7.2 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.randomint(4, size=100000)
rewards = np.random.random(100000)
terminals = np.random.randint(2, size=100000)

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
```

## 7.3 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)
```

CHAPTER

EIGHT

# PAPER REPRODUCTIONS

For the experiment code, please take a look at reproductions directory.

All the experimental results are available in d3rlpy-benchmarks repository.

CHAPTER

NINE

# LICENSE

CHAPTER

TEN

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## d

# INDEX

## Symbols



## A

## B

## C

## D

## E



## F

## G

## I



## L

## M



## N

## O



## P

## Q



## R

## S

## T

## U



## V