d3rlpy

Takuma Seno

Jul 18, 2023

TUTORIALS

1 Tutorials **3**
- 1.1 Getting Started . . . 3
- 1.2 Data Collection . . . 6
- 1.3 Create Your Dataset . . . 7
- 1.4 Preprocess / Postprocess . . . 8
- 1.5 Customize Neural Network . . . 10
- 1.6 Online RL . . . 12
- 1.7 Finetuning . . . 13
- 1.8 Offline Policy Selection . . . 15
- 1.9 Use Distributional Q-Function . . . 19
- 1.10 After Training Policies (Save and Load) . . . 20

2 Jupyter Notebooks **23**

3 Software Design **25**
- 3.1 MDPDataset . . . 25
- 3.2 Algorithm . . . 26

4 API Reference **27**
- 4.1 Algorithms . . . 27
- 4.2 Q Functions . . . 77
- 4.3 Replay Buffer . . . 84
- 4.4 Datasets . . . 104
- 4.5 Preprocessing . . . 107
- 4.6 Optimizers . . . 141
- 4.7 Network Architectures . . . 150
- 4.8 Metrics . . . 162
- 4.9 Off-Policy Evaluation . . . 169
- 4.10 Logging . . . 188
- 4.11 Online Training . . . 197

5 Command Line Interface **201**
- 5.1 plot . . . 201
- 5.2 plot-all . . . 202
- 5.3 export . . . 203
- 5.4 record . . . 203
- 5.5 play . . . 204

6 Installation **205**
- 6.1 Recommended Platforms . . . 205
- 6.2 Install d3rlpy . . . 205

7 Tips **207**
7.1 Reproducibility . . . 207
7.2 Learning from image observation . . . 207
7.3 Improve performance beyond the original paper . . . 208

8 Paper Reproductions **209**

9 License **211**

10 Indices and tables **213**

Python Module Index **215**

Index **217**

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 3.7+. 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 *Replay Buffer*.

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

```
from d3rlpy.datasets import get_cartpole # CartPole-v1 dataset
from d3rlpy.datasets import get_pendulum # Pendulum-v1 dataset
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()
```

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 DQNConfig

# if you don't use GPU, set device=None instead.
dqn = DQNConfig().create(device="cuda:0")

# 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. d3rlpy provides `Evaluator` classes to compute evaluation metrics.

```
from d3rlpy.metrics import TDErrorEvaluator

# calculate metrics with training dataset
td_error_evaluator = TDErrorEvaluator(episodes=dataset.episodes)
```

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

```
from d3rlpy.metrics import EnvironmentEvaluator

# set environment in scorer function
env_evaluator = EnvironmentEvaluator(env)

# evaluate algorithm on the environment
rewards = env_evaluator(dqn, dataset=None)
```

See more metrics and configurations at *Metrics*.

1.1.5 Start Training

Now, you have everything to start offline training.

```
dqn.fit(
    dataset,
    n_steps=10000,
    evaluators={
        'td_error': td_error_evaluator,
        'environment': env_evaluator,
    },
)
```

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.

```
 import d3rlpy

# save full parameters and configurations in a single file.
dqn.save('dqn.d3')
# load full parameters and build algorithm
dqn2 = d3rlpy.load_learnable("dqn.d3")

# save full parameters only
dqn.save_model('dqn.pt')
# load full parameters with manual setup
dqn3 = DQN()
dqn3.build_with_dataset(dataset)
dqn3.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')
```

See more information at *After Training Policies (Save and Load)*.

1.2 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.

1.2.1 Prepare Environment

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

```
import gym

env = gym.make("CartPole-v1")
```

1.2.2 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.DiscreteRandomPolicyConfig().create()

# prepare experience replay buffer
buffer = d3rlpy.dataset.create_fifo_replay_buffer(limit=100000, env=env)

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

# save ReplayBuffer
with open("random_policy_dataset.h5", "wb") as f:
    buffer.dump(f)
```

1.2.3 Data Collection with Trained Policy

If you want to collect experiences with previously trained policy, you can still use the same set of APIs. Here, let's say a DQN model is saved as `dqn_model.d3`. This procedure corresponds to `medium` datasets in D4RL.

```
# prepare pretrained algorithm
dqn = d3rlpy.load_learnable("dqn_model.d3")

# prepare experience replay buffer
buffer = d3rlpy.dataset.create_fifo_replay_buffer(limit=100000, env=env)

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

# save ReplayBuffer
with open("trained_policy_dataset.h5", "wb") as f:
  buffer.dump(f)
```

1.2.4 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.DQNConfig().create()

# prepare experience replay buffer
buffer = d3rlpy.dataset.create_fifo_replay_buffer(limit=100000, env=env)

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

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

# save ReplayBuffer
with open("replay_dataset.h5", "wb") as f:
  buffer.dump(f)
```

1.3 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.

1.3.1 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)
```

1.3.2 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,
)
```

1.3.3 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 `timeouts` to represent this timeout states.

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

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

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

1.4 Preprocess / Postprocess

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

1.4.1 Preprocess Observations

If your dataset includes unnormalized observations, you can normalize or standardize the observations by specifying `observation_scaler` argument. 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")

# prepare scaler without initialization
```

(continues on next page)

(continued from previous page)

```
observation_scaler = d3rlpy.preprocessing.StandardObservationScaler()

sac = d3rlpy.algos.SACConfig(observation_scaler=observation_scaler).create()
```

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)
observation_scaler = d3rlpy.preprocessing.StandardObservationScaler(mean=mean, std=std)

# set as observation_scaler
sac = d3rlpy.algos.SACConfig(observation_scaler=observation_scaler).create()
```

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

1.4.2 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.

```
# prepare scaler without initialization
action_scaler = d3rlpy.preprocessing.MinMaxActionScaler()

# set as action scaler
sac = d3rlpy.algos.SACConfig(action_scaler=action_scaler).create()

# 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,
)

# set as action scaler
sac = d3rlpy.algos.SACConfig(action_scaler=action_scaler).create()
```

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

1.4.3 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.

```
# prepare scaler without initialization
reward_scaler = d3rlpy.preprocessing.StandardRewardScaler()

# set as reward scaler
sac = d3rlpy.algos.SACConfig(reward_scaler=reward_scaler).create()
```

(continues on next page)

(continued from previous page)

```
# setup manuall
mean = np.mean(dataset.rewards)
std = np.std(dataset.rewards)
reward_scaler = StandardRewardScaler(mean=mean, std=std)

# set as reward scaler
sac = d3rlpy.algos.SACConfig(reward_scaler=reward_scaler).create()
```

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

1.5 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.

1.5.1 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):
        super().__init__()
        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(h))
        return h

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

1.5.2 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.DQNConfig(encoder_factory=CustomEncoderFactory(64)).create()
```

1.5.3 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):
        super().__init__()
        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):
```

(continues on next page)

(continued from previous page)

```
        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.SACConfig(
    actor_encoder_factory=encoder_factory,
    critic_encoder_factory=encoder_factory,
).create()
```

1.6 Online RL

1.6.1 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-v1")

# for evaluation
eval_env = gym.make("CartPole-v1")
```

1.6.2 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.DQNConfig(
    batch_size=32,
    learning_rate=2.5e-4,
    target_update_interval=100,
).create(device="cuda:0")

# 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)
```

1.6.3 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.dataset.create_fifo_replay_buffer(limit=100000, env=env)

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

1.6.4 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
)
```

1.6.5 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.DiscreteSACConfig().create()
sac.fit_online(
    env,
    buffer,
    n_steps=100000,
    eval_env=eval_env,
    n_steps_per_epoch=1000,
    update_start_step=1000,
)
```

1.7 Finetuning

d3rlpy supports smooth transition from offline training to online training.

1.7.1 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")
```

1.7.2 Pretrain with Dataset

```
# setup algorithm
dqn = d3rlpy.algos.DQNConfig().create()

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

1.7.3 Finetune with Environment

```
# setup experience replay buffer
buffer = d3rlpy.dataset.create_fifo_replay_buffer(limit=100000, env=env)

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

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

1.7.4 Finetune with Saved Policy

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

```
# setup algorithm
dqn = d3rlpy.load_learnable("dqn_model.d3")

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

1.7.5 Finetune with Different Algorithm

If you want to finetune the saved policy trained offline with online RL algorithms, you can do it in an out-of-the-box way.

```
# setup offline RL algorithm
cql = d3rlpy.algos.DiscreteCQLConfig().create()

# train offline
cql.fit(dataset, n_steps=100000)
```

(continues on next page)

(continued from previous page)

```
# transfer to DQN
dqn = d3rlpy.algos.DQNConfig().create()
dqn.copy_q_function_from(cql)

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

In actor-critic cases, you should also transfer the policy function.

```
# offline RL
cql = d3rlpy.algos.CQLConfig().create()
cql.fit(dataset, n_steps=100000)

# transfer to SAC
sac = d3rlpy.algos.SACConfig().create()
sac.build_with_env(env)
sac.copy_q_function_from(cql)
sac.copy_policy_from(cql)

# online RL
sac.fit_online(env, buffer, n_steps=100000)
```

1.8 Offline Policy Selection

d3rlpy supports offline policy selection by training Fitted Q Evaluation (FQE), which is an offline on-policy RL algorithm. The use of FQE for offline policy selection is proposed by Paine et al.. The concept is that FQE trains Q-function with the trained policy in on-policy manner so that the learned Q-function reflects the expected return of the trained policy. By using the Q-value estimation of FQE, the candidate trained policies can be ranked only with offline dataset. Check *Off-Policy Evaluation* for more information.

Note: Offline policy selection with FQE is confirmed that it usually works out with discrete action-space policies. However, it seems require some hyperparameter tuning for ranking continuous action-space policies. The more techniques will be supported along with the advancement of this research domain.

1.8.1 Prepare trained policies

In this tutorial, let's train DQN with the built-in CartPole-v0 dataset.

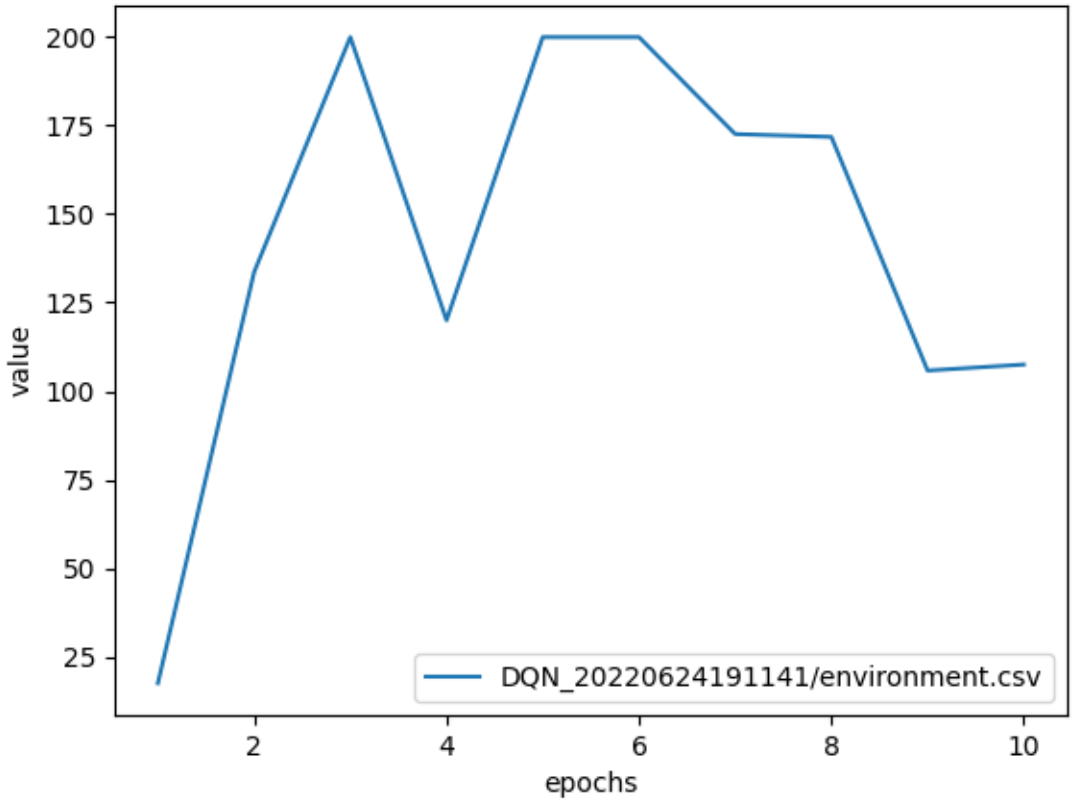
```
import d3rlpy

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

# setup algorithm
dqn = d3rlpy.algos.DQNConfig().create()

# start offline training
```

(continues on next page)

(continued from previous page)

```
dqn.fit(
    dataset,
    n_steps=100000,
    n_steps_per_epoch=10000,
    scorers={
        "environment": d3rlpy.metrics.EnvironmentEvaluator(env),
    },
)
```

Here is the example result of online evaluation.

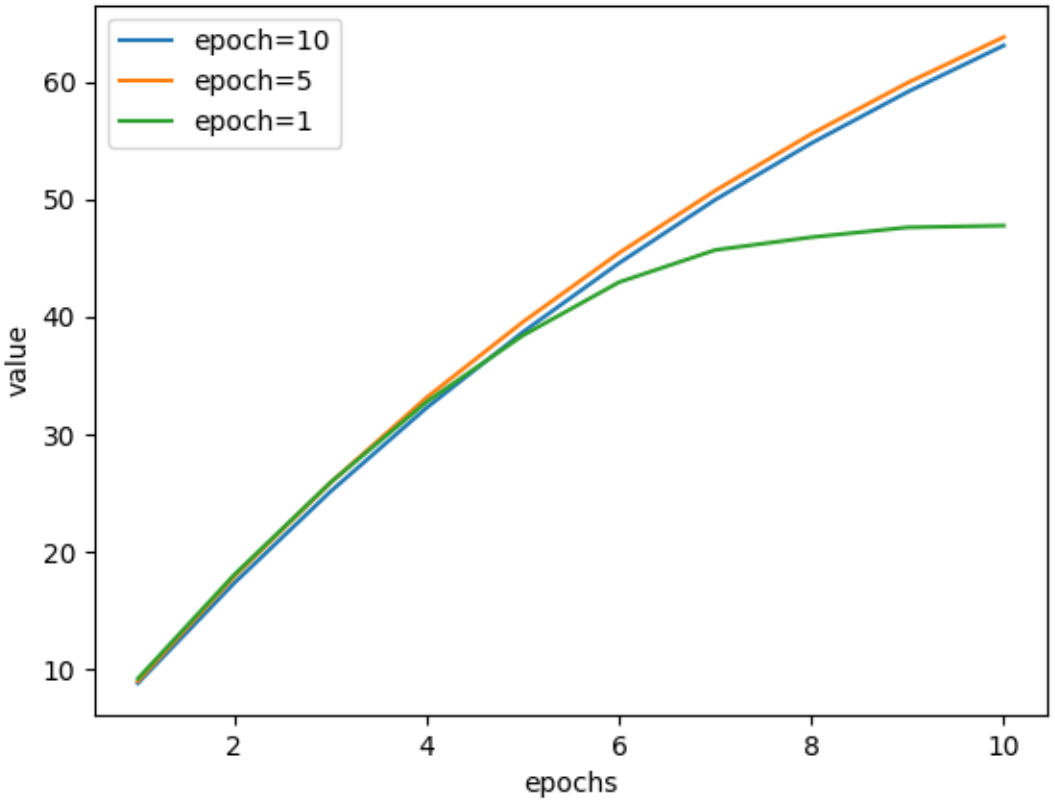


1.8.2 Train FQE with the trained policies

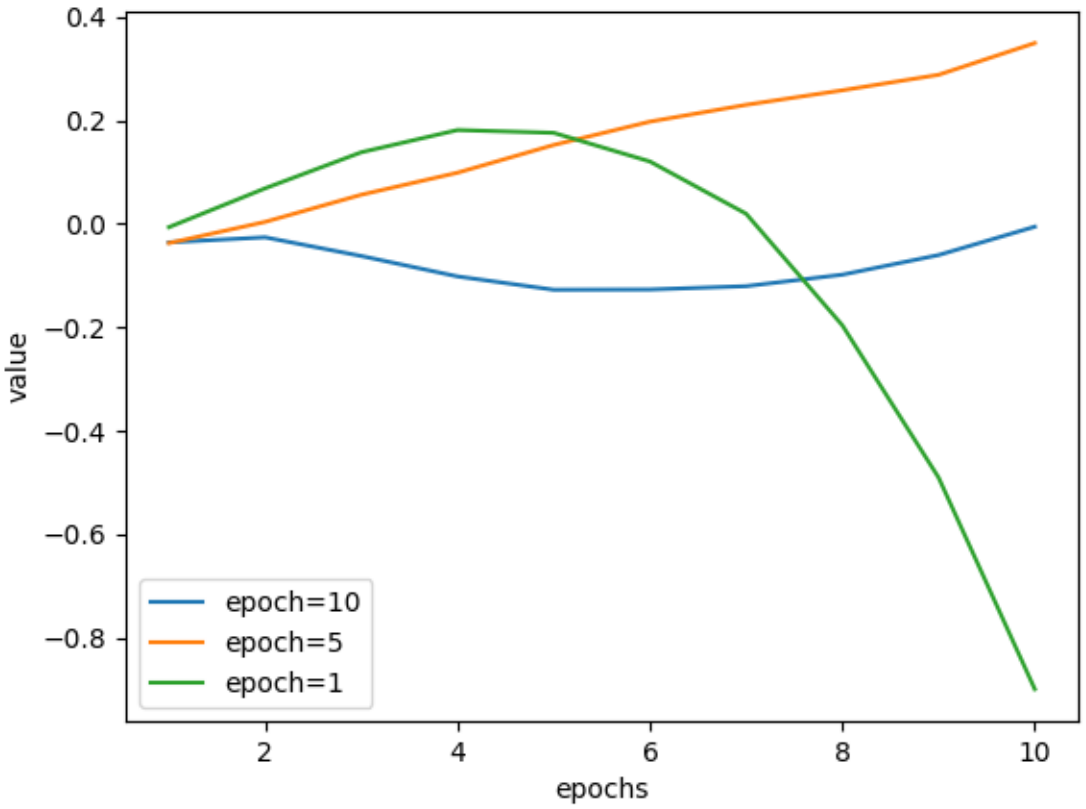
Next, we train FQE algorithm with the trained policies. Please note that we use `initial_state_value_estimation_scorer` and `soft_opc_scorer` proposed in Paine et al.. `initial_state_value_estimation_scorer` computes the mean action-value estimation at the initial states. Thus, if this value for a certain policy is bigger than others, the policy is expected to obtain the higher episode return. On the other hand, `soft_opc_scorer` computes the mean difference between the action-value estimation for the success episodes and the action-value estimation for the all episodes. If this value for a certain policy is bigger than others, the learned Q-function can clearly tell the difference between the success episodes and others.

```
import d3rlpy

# setup the same dataset used in policy training
dataset, _ = d3rlpy.datasets.get_dataset("cartpole-replay")

# load pretrained policy
dqn = d3rlpy.load_learnable("d3rlpy_logs/DQN_20220624191141/model_100000.d3")

# setup FQE algorithm
fqe = d3rlpy.ope.DiscreteFQE(algo=dqn, config=d3rlpy.ope.DiscreteFQEConfig())

# start FQE training
fqe.fit(
   dataset,
   n_steps=10000,
   n_steps_per_epoch=1000,
   scorers={
       "init_value": d3rlpy.metrics.InitialStateValueEstimationEvaluator(),
       "soft_opc": d3rlpy.metrics.SoftOPCEvaluator(180),  # set 180 for success return␣
↪threshold here
   },
)
```

In this example, the policies from epoch 10, epoch 5 and epoch 1 (evaluation episode returns of 107.5, 200.0 and 17.5 respectively) are compared. The first figure represents the `init_value` metrics during FQE training. As you can see here, the scale of `init_value` has correlation with the ranks of evaluation episode returns.



The second figure represents the `soft_opc` metrics during FQE training. These curves also have correlation with the ranks of evaluation episode returns.



Please note that there is usually no convergence in offline RL training due to the non-fixed bootstrapped target.

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.

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

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

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

(continues on next page)

(continued from previous page)

```
sac = d3rlpy.algos.SACConfig(q_func_factory=iqn_q_function).create()
```

1.10 After Training Policies (Save and Load)

This page provides answers to frequently asked questions about how to use the trained policies with your environment.

1.10.1 Prepare Pretrained Policies

```
import d3rlpy

# prepare dataset and environment
dataset, env = d3rlpy.datasets.get_dataset('pendulum-random')

# setup algorithm
cql_old = d3rlpy.algos.CQLConfig().create(device="cuda:0")

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

1.10.2 Load Trained Policies

```
# Option 1: Load d3 file

# save d3 file
cql_old.save("model.d3")
# reconstruct full setup from a d3 file
cql = d3rlpy.load_learnable("model.d3")


# Option 2: Load pt file

# save pt file
cql_old.save_model("model.pt")
# setup algorithm manually
cql = d3rlpy.algos.CQLConfig().create()

# choose one of three to build PyTorch models

# if you have MDPDataset object
cql.build_with_dataset(dataset)
# or if you have Gym-styled environment object
cql.build_with_env(env)
# or manually set observation shape and action size
cql.create_impl((3,), 1)

# load pretrained model
cql.load_model("model.pt")
```

1.10.3 Inference

Now, you can use `predict` method to infer the actions. Please note that the observation MUST have the batch dimension.

```
import numpy as np

# make sure that the observation has the batch dimension
observation = np.random.random((1, 3))

# infer the action
action = cql.predict(observation)
assert action.shape == (1, 1)
```

You can manually make the policy interact with the environment.

```
observation = env.reset()
while True:
   action = cql.predict([observation])[0]
   observation, reward, done, _ = env.step(action)
   if done:
       break
```

1.10.4 Export Policies as TorchScript

Alternatively, you can export the trained policy as TorchScript format. The advantage of the TorchScript format is that the exported policy can be used by not only Python programs, but also C++ programs, which would be useful for robotics integration. Another merit is that the trained policy depends only on PyTorch so that you don't need to install d3rlpy at production.

```
# export as TorchScript
cql.save_policy("policy.pt")


import torch

# load TorchScript policy
policy = torch.jit.load("policy.pt")

# infer the action
action = policy(torch.rand(1, 3))
assert action.shape == (1, 1)
```

1.10.5 Export Policies as ONNX

Alternatively, you can also export the trained policy as ONNX. ONNX is a widely used machine learning model format that is supported by numerous programming languages.

```
# export as ONNX
cql.save_policy("policy.onnx")


import onnxruntime as ort

# load ONNX policy via onnxruntime
ort_session = ort.InferenceSession('policy.onnx')

# observation
observation = np.random.rand(1, 3).astype(np.float32)

# returns greedy action
action = ort_session.run(None, {'input_0': observation})
assert action.shape == (1, 1)
```

CHAPTER

TWO

JUPYTER NOTEBOOKS

- CartPole
- Discrete Control with Atari

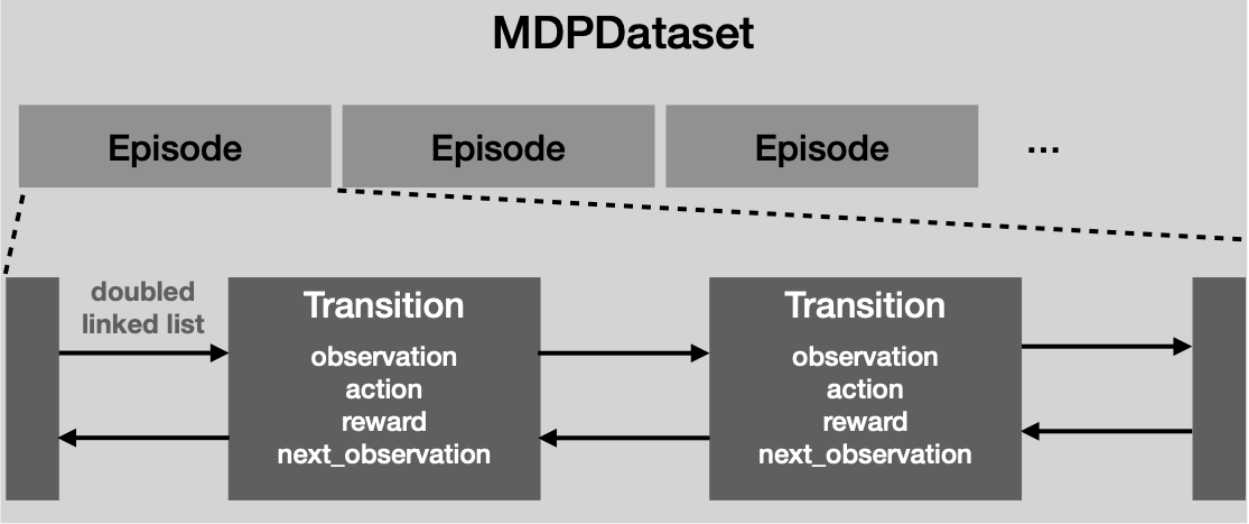
CHAPTER

THREE

SOFTWARE DESIGN

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

3.1 MDPDataset



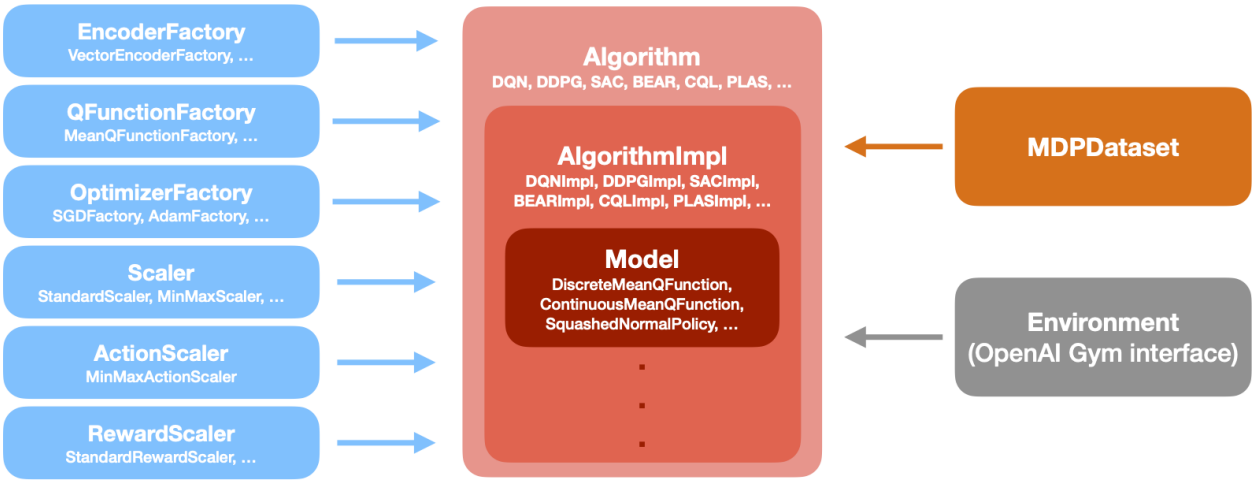
`MDPDataset` is a dedicated dataset structure for offline RL. `MDPDataset` automatically structures dataset based on `Episode` and `Transition`. `Episode` represents a single episode that includes multiple `Transition` objects collected in the episode. `Transition` represents a single tuple experience 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 tutorials/play_with_mdp_dataset for the tutorial and *Replay Buffer* 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*
ObservationScaler	*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.

Each algorithm provides its config class and you can instantiate it with specifying a device to use.

```
import d3rlpy

# instantiate algorithm with CPU
sac = d3rlpy.algos.SACConfig().create(device="cpu:0")
# instantiate algorithm with GPU
sac = d3rlpy.algos.SACConfig().create(device="cuda:0")
# instantiate algorithm with the 2nd GPU
sac = d3rlpy.algos.SACConfig().create(device="cuda:1")
```

You can also check advanced use cases at examples directory.

4.1.1 Base

LearnableBase

The base class of all algorithms.

class d3rlpy.base.**LearnableBase**(*config*, *device*, *impl=None*)

Bases: Generic[d3rlpy.base.TImpl_co, d3rlpy.base.TConfig_co]

property action_scaler: Optional[d3rlpy.preprocessing.action_scalers.ActionScaler]

Preprocessing action scaler.

Returns preprocessing action scaler.

Return type Optional[ActionScaler]

property action_size: Optional[int]

Action size.

Returns action size.

Return type Optional[int]

property batch_size: int

Batch size to train.

Returns batch size.

Return type int

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

Parameters **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – dataset.

Return type None

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

Parameters **env** (*gym.core.Env[Any, Any]*) – gym-like environment.

Return type None

property config: d3rlpy.base.TConfig_co
Config.

Returns config.

Return type LearnableConfig

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** (*Union[Sequence[int], Sequence[Sequence[int]]]*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

Return type None

classmethod from_json(*fname*, *device=False*)
Construct algorithm from params.json file.

```
from d3rlpy.algos import CQL

cql = CQL.from_json("<path-to-json>", device='cuda:0')
```

Parameters

- **fname** (*str*) – path to params.json
- **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type typing_extensions.Self

property gamma: float
Discount factor.

Returns discount factor.

Return type float

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

property grad_step: int
Total gradient step counter.

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

Returns total gradient step counter.

property impl: Optional[d3rlpy.base.TImpl_co]
Implementation object.

Returns implementation object.

Return type Optional[ImplBase]

load_model(*fname*)
Load neural network parameters.

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

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

Return type None

property observation_scaler: Optional[d3rlpy.preprocessing.observation_scalers.ObservationScaler]
Preprocessing observation scaler.

Returns preprocessing observation scaler.

Return type Optional[ObservationScaler]

property observation_shape: Optional[Union[Sequence[int], Sequence[Sequence[int]]]]
Observation shape.

Returns observation shape.

Return type Optional[Sequence[int]]

property reward_scaler: Optional[d3rlpy.preprocessing.reward_scalers.RewardScaler]
Preprocessing reward scaler.

Returns preprocessing reward scaler.

Return type Optional[RewardScaler]

save(*fname*)
Saves paired data of neural network parameters and serialized config.

```
algo.save('model.d3')

# reconstruct everything
algo2 = d3rlpy.load_learnable("model.d3", device="cuda:0")
```

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

Return type None

save_model(*fname*)
Saves neural network parameters.

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

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

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

4.1.2 Q-learning

QLearningAlgoBase

The base class of Q-learning algorithms.

class d3rlpy.algos.**QLearningAlgoBase**(*config*, *device*, *impl=None*)
Bases: Generic[d3rlpy.algos.qlearning.base.TQLearningImpl, d3rlpy.algos.qlearning.base.TQLearningConfig], *d3rlpy.base.LearnableBase*[d3rlpy.algos.qlearning.base.TQLearningImpl, d3rlpy.algos.qlearning.base.TQLearningConfig]

collect(*env*, *buffer=None*, *explorer=None*, *deterministic=False*, *n_steps=1000000*, *show_progress=True*)
Collects data via interaction with environment.

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

Parameters

- **env** (*gym.core.Env[Any, Any]*) – Fym-like environment.
- **buffer** (*Optional[d3rlpy.dataset.replay_buffer.ReplayBuffer]*) – Replay buffer.
- **explorer** (*Optional[d3rlpy.algos.qlearning.explorers.Explorer]*) – Action explorer.
- **deterministic** (*bool*) – Flag to collect data with the greedy policy.
- **n_steps** (*int*) – Number of total steps to train.
- **show_progress** (*bool*) – Flag to show progress bar for iterations.

Returns Replay buffer with the collected data.

Return type *d3rlpy.dataset.replay_buffer.ReplayBuffer*

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.qlearning.base.QLearningAlgoBase[d3rlpy.algos.qlearning.base.QLearningAlgoImplBase, d3rlpy.base.LearnableConfig]) – 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.qlearning.base.QLearningAlgoBase[d3rlpy.algos.qlearning.base.QLearningAlgoImplBase, d3rlpy.base.LearnableConfig]) – 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.qlearning.base.QLearningAlgoBase[d3rlpy.algos.qlearning.base.QLearningAlgoImplBase, d3rlpy.base.LearnableConfig]) – 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.qlearning.base.QLearningAlgoBase[d3rlpy.algos.qlearning.base.QLearningAlgoImplBase, d3rlpy.base.LearnableConfig]) – Algorithm object.

Return type None

fit(*dataset*, *n_steps*, *n_steps_per_epoch=10000*, *experiment_name=None*, *with_timestamp=True*, *logger_adapter=<d3rlpy.logging.file_adapter.FileAdapterFactory object>*, *show_progress=True*, *save_interval=1*, *evaluators=None*, *callback=None*, *epoch_callback=None*)
Trains with given dataset.

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

Parameters

- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – ReplayBuffer object.
- **n_steps** (*int*) – Number of steps to train.
- **n_steps_per_epoch** (*int*) – Number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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.
- **logger_adapter** (d3rlpy.logging.logger.LoggerAdapterFactory) – LoggerAdapterFactory object.
- **show_progress** (*bool*) – Flag to show progress bar for iterations.
- **save_interval** (*int*) – Interval to save parameters.
- **evaluators** (*Optional[Dict[str, d3rlpy.metrics.evaluators.EvaluatorProtocol]]*) – List of evaluators.
- **callback** (*Optional[Callable[[typing_extensions.Self, int, int], None]]*) – Callable function that takes `(algo, epoch, total_step)` , which is called every step.
- **epoch_callback** (*Optional[Callable[[typing_extensions.Self, int, int], None]]*) – Callable function that takes `(algo, epoch, total_step)`, which is called at the end of every epoch.

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_interval=1*, *experiment_name=None*, *with_timestamp=True*, *logger_adapter=<d3rlpy.logging.file_adapter.FileAdapterFactory object>*, *show_progress=True*, *callback=None*)

Start training loop of online deep reinforcement learning.

Parameters

- **env** (*gym.core.Env[Any, Any]*) – Gym-like environment.
- **buffer** (*Optional[d3rlpy.dataset.replay_buffer.ReplayBuffer]*) – Replay buffer.
- **explorer** (*Optional[d3rlpy.algos.qlearning.explorers.Explorer]*) – Action explorer.
- **n_steps** (*int*) – Number of total steps to train.
- **n_steps_per_epoch** (*int*) – Number of steps per epoch.
- **update_interval** (*int*) – Number of steps per update.
- **update_start_step** (*int*) – Steps before starting updates.
- **random_steps** (*int*) – Steps for the initial random explortion.
- **eval_env** (*Optional[gym.core.Env[Any, Any]]*) – Gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ-greedy factor during evaluation.
- **save_interval** (*int*) – 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.
- **logger_adapter** (d3rlpy.logging.logger.LoggerAdapterFactory) – LoggerAdapterFactory object.
- **show_progress** (*bool*) – Flag to show progress bar for iterations.
- **callback** (*Optional[Callable[[typing_extensions.Self, 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_steps*, *n_steps_per_epoch=10000*, *experiment_name=None*, *with_timestamp=True*, *logger_adapter=<d3rlpy.logging.file_adapter.FileAdapterFactory object>*, *show_progress=True*, *save_interval=1*, *evaluators=None*, *callback=None*, *epoch_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** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – Offline dataset to train.

- **n_steps** (*int*) – Number of steps to train.
- **n_steps_per_epoch** (*int*) – Number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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.
- **logger_adapter** (d3rlpy.logging.logger.LoggerAdapterFactory) – LoggerAdapterFactory object.
- **show_progress** (*bool*) – Flag to show progress bar for iterations.
- **save_interval** (*int*) – Interval to save parameters.
- **evaluators** (*Optional[Dict[str, d3rlpy.metrics.evaluators.EvaluatorProtocol]]*) – List of evaluators.
- **callback** (*Optional[Callable[[typing_extensions.Self, int, int], None]]*) – Callable function that takes `(algo, epoch, total_step)` , which is called every step.
- **epoch_callback** (*Optional[Callable[[typing_extensions.Self, int, int], None]]*) – Callable function that takes `(algo, epoch, total_step)`, which is called at the end of every epoch.

Returns Iterator yielding current epoch and metrics dict.

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

abstract inner_update(*batch*)

Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

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, Sequence[numpy.ndarray]]*) – Observations

Returns Greedy actions

Return type numpy.ndarray

predict_value(*x*, *action*)

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,)
```

Parameters

- **x** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observations
- **action** (*numpy.ndarray*) – Actions

Returns Predicted action-values

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*)

Returns sampled actions.

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

Parameters **x** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observations.

Returns Sampled actions.

Return type numpy.ndarray

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

update(*batch*)
Update parameters with mini-batch of data.

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

Returns Dictionary of metrics.

Return type Dict[str, float]

BC

class d3rlpy.algos.**BCConfig**(*batch_size=100*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *learning_rate=0.001*, *policy_type='deterministic'*, *optim_factory=<factory>*, *encoder_factory=<factory>*)
Bases: d3rlpy.base.LearnableConfig

Config of 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*) – Encoder factory.
- **batch_size** (*int*) – Mini-batch size.
- **policy_type** (*str*) – the policy type. Available options are ['deterministic', 'stochastic'].
- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **gamma** (*float*) –
- **reward_scaler** (*Optional[d3rlpy.preprocessing.reward_scalers.RewardScaler]*) –

Return type None

create(*device=False*)
Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, cuda:0 will be used. If the value is integer, cuda:<device> will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.bc.BC*

class d3rlpy.algos.**BC**(*config*, *device*, *impl=None*)
Bases: d3rlpy.algos.qlearning.bc._BCBase[*d3rlpy.algos.qlearning.bc.BCConfig*]

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

DiscreteBC

class d3rlpy.algos.**DiscreteBCConfig**(*batch_size=100*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *learning_rate=0.001*, *optim_factory=<factory>*, *encoder_factory=<factory>*, *beta=0.5*)
Bases: d3rlpy.base.LearnableConfig

Config of 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*) – Encoder factory.
- **batch_size** (*int*) – Mini-batch size.
- **beta** (*float*) – Reguralization factor.
- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **gamma** (*float*) –
- **action_scaler** (*Optional[d3rlpy.preprocessing.action_scalers.ActionScaler]*) –
- **reward_scaler** (*Optional[d3rlpy.preprocessing.reward_scalers.RewardScaler]*) –

Return type None

create(*device=False*)
Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, cuda:0 will be used. If the value is integer, cuda:<device> will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.bc.DiscreteBC*

class d3rlpy.algos.**DiscreteBC**(*config*, *device*, *impl=None*)
Bases: d3rlpy.algos.qlearning.bc._BCBase[*d3rlpy.algos.qlearning.bc.DiscreteBCConfig*]

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

NFQ

class d3rlpy.algos.**NFQConfig**(*batch_size=32*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *learning_rate=6.25e-05*, *optim_factory=<factory>*, *encoder_factory=<factory>*, *q_func_factory=<factory>*, *n_critics=1*)
Bases: d3rlpy.base.LearnableConfig

Config of 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 θ' 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

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **learning_rate** (*float*) – Learning rate.
- **optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – Optimizer factory.
- **encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **n_critics** (*int*) – Number of Q functions for ensemble.
- **action_scaler** (*Optional[d3rlpy.preprocessing.action_scalers.ActionScaler]*) –

Return type None

create(*device=False*)
Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, cuda:0 will be used. If the value is integer, cuda:<device> will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.nfq.NFQ*

class d3rlpy.algos.**NFQ**(*config*, *device*, *impl=None*)
Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.dqn_impl.DQNImpl, *d3rlpy.algos.qlearning.nfq.NFQConfig*]

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

DQN

class d3rlpy.algos.**DQNConfig**(*batch_size=32*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *learning_rate=6.25e-05*, *optim_factory=<factory>*, *encoder_factory=<factory>*, *q_func_factory=<factory>*, *n_critics=1*, *target_update_interval=8000*)
Bases: d3rlpy.base.LearnableConfig

Config of Deep Q-Network algorithm.

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

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

References

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

Parameters

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **learning_rate** (*float*) – Learning rate.
- **optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – Optimizer factory.

- **encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **n_critics** (*int*) – Number of Q functions for ensemble.
- **target_update_interval** (*int*) – Interval to update the target network.
- **action_scaler** (*Optional[d3rlpy.preprocessing.action_scalers.ActionScaler]*) –

Return type None

create(*device=False*)
Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.dqn.DQN*

class d3rlpy.algos.**DQN**(*config*, *device*, *impl=None*)
Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.dqn_impl.DQNImpl, *d3rlpy.algos.qlearning.dqn.DQNConfig*]

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

DoubleDQN

class d3rlpy.algos.**DoubleDQNConfig**(*batch_size=32*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *learning_rate=6.25e-05*, *optim_factory=<factory>*, *encoder_factory=<factory>*, *q_func_factory=<factory>*, *n_critics=1*, *target_update_interval=8000*)
Bases: *d3rlpy.algos.qlearning.dqn.DQNConfig*

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

References

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

Parameters

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **learning_rate** (*float*) – Learning rate.
- **optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – Optimizer factory.
- **encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **n_critics** (*int*) – Number of Q functions.
- **target_update_interval** (*int*) – Interval to synchronize the target network.
- **action_scaler** (*Optional[d3rlpy.preprocessing.action_scalers.ActionScaler]*) –

Return type None

create(*device=False*)

Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, cuda:0 will be used. If the value is integer, cuda:<device> will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.dqn.DoubleDQN*

class d3rlpy.algos.**DoubleDQN**(*config*, *device*, *impl=None*)

Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.dqn_impl.DQNImpl, *d3rlpy.algos.qlearning.dqn.DQNConfig*]

DDPG

class d3rlpy.algos.**DDPGConfig**(*batch_size=256*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *actor_learning_rate=0.0003*, *critic_learning_rate=0.0003*, *actor_optim_factory=<factory>*, *critic_optim_factory=<factory>*, *actor_encoder_factory=<factory>*, *critic_encoder_factory=<factory>*, *q_func_factory=<factory>*, *tau=0.005*, *n_critics=1*)

Bases: d3rlpy.base.LearnableConfig

Config of Deep Deterministic Policy Gradients algorithm.

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

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

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

where θ' and ϕ 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

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **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*) – Encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the critic.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.

- **tau** (*float*) – Target network synchronization coefficiency.
- **n_critics** (*int*) – Number of Q functions for ensemble.

Return type None

create(*device=False*)
Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, cuda:0 will be used. If the value is integer, cuda:<device> will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.ddpg.DDPG*

class d3rlpy.algos.**DDPG**(*config*, *device*, *impl=None*)
Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.ddpg_impl.DDPGImpl, *d3rlpy.algos.qlearning.ddpg.DDPGConfig*]

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

TD3

class d3rlpy.algos.**TD3Config**(*batch_size=256*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *actor_learning_rate=0.0003*, *critic_learning_rate=0.0003*, *actor_optim_factory=<factory>*, *critic_optim_factory=<factory>*, *actor_encoder_factory=<factory>*, *critic_encoder_factory=<factory>*, *q_func_factory=<factory>*, *tau=0.005*, *n_critics=2*, *target_smoothing_sigma=0.2*, *target_smoothing_clip=0.5*, *update_actor_interval=2*)
Bases: d3rlpy.base.LearnableConfig

Config of 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

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **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*) – Encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the critic.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **tau** (*float*) – Target network synchronization coefficiency.
- **n_critics** (*int*) – 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.

Return type None

create(*device=False*)

Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.td3.TD3*

class d3rlpy.algos.**TD3**(*config*, *device*, *impl=None*)

Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.td3_impl.TD3Impl, *d3rlpy.algos.qlearning.td3.TD3Config*]

get_action_type()

Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)

Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

SAC

class d3rlpy.algos.**SACConfig**(*batch_size=256*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *actor_learning_rate=0.0003*, *critic_learning_rate=0.0003*, *temp_learning_rate=0.0003*, *actor_optim_factory=<factory>*, *critic_optim_factory=<factory>*, *temp_optim_factory=<factory>*, *actor_encoder_factory=<factory>*, *critic_encoder_factory=<factory>*, *q_func_factory=<factory>*, *tau=0.005*, *n_critics=2*, *initial_temperature=1.0*)

Bases: d3rlpy.base.LearnableConfig

Config 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 α 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

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **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*) – Encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the critic.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **tau** (*float*) – Target network synchronization coefficiency.
- **n_critics** (*int*) – Number of Q functions for ensemble.
- **initial_temperature** (*float*) – Initial temperature value.

Return type None

create(*device=False*)

Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, cuda:0 will be used. If the value is integer, cuda:<device> will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.sac.SAC*

class d3rlpy.algos.**SAC**(*config*, *device*, *impl=None*)

Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.sac_impl.SACImpl, *d3rlpy.algos.qlearning.sac.SACConfig*]

get_action_type()

Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)

Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

DiscreteSAC

class d3rlpy.algos.**DiscreteSACConfig**(*batch_size=64*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *actor_learning_rate=0.0003*, *critic_learning_rate=0.0003*, *temp_learning_rate=0.0003*, *actor_optim_factory=<factory>*, *critic_optim_factory=<factory>*, *temp_optim_factory=<factory>*, *actor_encoder_factory=<factory>*, *critic_encoder_factory=<factory>*, *q_func_factory=<factory>*, *n_critics=2*, *initial_temperature=1.0*, *target_update_interval=8000*)

Bases: d3rlpy.base.LearnableConfig

Config of 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

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.

- **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*) – Encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the critic.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **n_critics** (*int*) – Number of Q functions for ensemble.
- **initial_temperature** (*float*) – Initial temperature value.
- **action_scaler** (*Optional[d3rlpy.preprocessing.action_scalers.ActionScaler]*) –
- **target_update_interval** (*int*) –

Return type None

create(*device=False*)

Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.sac.DiscreteSAC*

class d3rlpy.algos.**DiscreteSAC**(*config*, *device*, *impl=None*)

Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.sac_impl.DiscreteSACImpl, *d3rlpy.algos.qlearning.sac.DiscreteSACConfig*]

get_action_type()

Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)

Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

BCQ

class d3rlpy.algos.**BCQConfig**(*batch_size=100*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *actor_learning_rate=0.001*, *critic_learning_rate=0.001*, *imitator_learning_rate=0.001*, *actor_optim_factory=<factory>*, *critic_optim_factory=<factory>*, *imitator_optim_factory=<factory>*, *actor_encoder_factory=<factory>*, *critic_encoder_factory=<factory>*, *imitator_encoder_factory=<factory>*, *q_func_factory=<factory>*, *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*)

Bases: d3rlpy.base.LearnableConfig

Config of 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_ω and D_ω respectively.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **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*) – Encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the critic.
- **imitator_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the conditional VAE.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **tau** (*float*) – Target network synchronization coefficiency.
- **n_critics** (*int*) – 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*) – Number of action samples to estimate action-values.
- **action_flexibility** (*float*) – Output scale of perturbation function represented as Φ.
- **rl_start_step** (*int*) – Steps 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.

Return type None

create(*device=False*)
Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.bcq.BCQ*

class d3rlpy.algos.**BCQ**(*config*, *device*, *impl=None*)

Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.bcq_impl.BCQImpl, *d3rlpy.algos.qlearning.bcq.BCQConfig*]

get_action_type()

Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)

Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

DiscreteBCQ

class d3rlpy.algos.**DiscreteBCQConfig**(*batch_size=32*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *learning_rate=6.25e-05*, *optim_factory=<factory>*, *encoder_factory=<factory>*, *q_func_factory=<factory>*, *n_critics=1*, *action_flexibility=0.3*, *beta=0.5*, *target_update_interval=8000*)

Bases: d3rlpy.base.LearnableConfig

Config of 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 τ times smaller than the maximum one.

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

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

References

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

Parameters

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **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.
- **gamma** (*float*) – Discount factor.
- **n_critics** (*int*) – Number of Q functions for ensemble.
- **action_flexibility** (*float*) – Probability threshold represented as τ.
- **beta** (*float*) – Reguralization term for imitation function.
- **target_update_interval** (*int*) – Interval to update the target network.
- **action_scaler** (*Optional[d3rlpy.preprocessing.action_scalers.ActionScaler]*) –

Return type None

create(*device=False*)
Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, cuda:0 will be used. If the value is integer, cuda:<device> will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.bcq.DiscreteBCQ*

class d3rlpy.algos.**DiscreteBCQ**(*config*, *device*, *impl=None*)
Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.bcq_impl.DiscreteBCQImpl, *d3rlpy.algos.qlearning.bcq.DiscreteBCQConfig*]

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

BEAR

class d3rlpy.algos.**BEARConfig**(*batch_size=256*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *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=<factory>*, *critic_optim_factory=<factory>*, *imitator_optim_factory=<factory>*, *temp_optim_factory=<factory>*, *alpha_optim_factory=<factory>*, *actor_encoder_factory=<factory>*, *critic_encoder_factory=<factory>*, *imitator_encoder_factory=<factory>*, *q_func_factory=<factory>*, *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*)

Bases: d3rlpy.base.LearnableConfig

Config of 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 $k(x, y) = \exp\left((x - y)^2/(2\sigma^2)\right)$.

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

References

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

Parameters

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.

- **actor_learning_rate** (*float*) – Learning rate for policy function.
- **critic_learning_rate** (*float*) – Learning rate for Q functions.
- **imitator_learning_rate** (*float*) – Learning rate for behavior policy function.
- **temp_learning_rate** (*float*) – Learning rate for temperature parameter.
- **alpha_learning_rate** (*float*) – Learning rate for α.
- **actor_optim_factory** (d3rlpy.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 α.
- **actor_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the critic.
- **imitator_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the behavior policy.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **tau** (*float*) – Target network synchronization coefficiency.
- **n_critics** (*int*) – Number of Q functions for ensemble.
- **initial_temperature** (*float*) – Initial temperature value.
- **initial_alpha** (*float*) – Initial α value.
- **alpha_threshold** (*float*) – Threshold value described as ϵ.
- **lam** (*float*) – Weight for critic ensemble.
- **n_action_samples** (*int*) – Number of action samples to compute the best action.
- **n_target_samples** (*int*) – Number of action samples to compute BCQ-like target value.
- **n_mmd_action_samples** (*int*) – Number of action samples to compute MMD.
- **mmd_kernel** (*str*) – MMD kernel function. The available options are `['gaussian', 'laplacian']`.
- **mmd_sigma** (*float*) – σ for gaussian kernel in MMD calculation.
- **vae_kl_weight** (*float*) – Constant weight to scale KL term for behavior policy training.
- **warmup_steps** (*int*) – Number of steps to warmup the policy function.

Return type None

create(*device=False*)
Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.bear.BEAR*

class d3rlpy.algos.**BEAR**(*config*, *device*, *impl=None*)
Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.bear_impl.BEARImpl, *d3rlpy.algos.qlearning.bear.BEARConfig*]

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

CRR

class d3rlpy.algos.**CRRConfig**(*batch_size=100*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *actor_learning_rate=0.0003*, *critic_learning_rate=0.0003*, *actor_optim_factory=<factory>*, *critic_optim_factory=<factory>*, *actor_encoder_factory=<factory>*, *critic_encoder_factory=<factory>*, *q_func_factory=<factory>*, *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*)
Bases: d3rlpy.base.LearnableConfig

Config of 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

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **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*) – Encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the critic.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **beta** (*float*) – Temperature value defined as β above.
- **n_action_samples** (*int*) – 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*) – 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.
- **target_update_interval** (*int*) –

Return type None

create(*device=False*)

Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.crr.CRR*

class d3rlpy.algos.**CRR**(*config*, *device*, *impl=None*)

Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.crr_impl.CRRImpl, *d3rlpy.algos.qlearning.crr.CRRConfig*]

get_action_type()

Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)

Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

CQL

class d3rlpy.algos.**CQLConfig**(*batch_size=256*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *actor_learning_rate=0.0001*, *critic_learning_rate=0.0003*, *temp_learning_rate=0.0001*, *alpha_learning_rate=0.0001*, *actor_optim_factory=<factory>*, *critic_optim_factory=<factory>*, *temp_optim_factory=<factory>*, *alpha_optim_factory=<factory>*, *actor_encoder_factory=<factory>*, *critic_encoder_factory=<factory>*, *q_func_factory=<factory>*, *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*)

Bases: d3rlpy.base.LearnableConfig

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

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

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

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

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **actor_learning_rate** (*float*) – Learning rate for policy function.
- **critic_learning_rate** (*float*) – Learning rate for Q functions.
- **temp_learning_rate** (*float*) – Learning rate for temperature parameter of SAC.
- **alpha_learning_rate** (*float*) – Learning rate for α.
- **actor_optim_factory** (d3rlpy.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 α.
- **actor_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the critic.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.

- **gamma** (*float*) – Discount factor.
- **tau** (*float*) – Target network synchronization coefficiency.
- **n_critics** (*int*) – Number of Q functions for ensemble.
- **initial_temperature** (*float*) – Initial temperature value.
- **initial_alpha** (*float*) – Initial α value.
- **alpha_threshold** (*float*) – Threshold value described as τ.
- **conservative_weight** (*float*) – Constant weight to scale conservative loss.
- **n_action_samples** (*int*) – Number of sampled actions to compute $\log \sum_a \exp Q(s, a)$.
- **soft_q_backup** (*bool*) – Flag to use SAC-style backup.

Return type None

create(*device=False*)

Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.cql.CQL*

class d3rlpy.algos.**CQL**(*config*, *device*, *impl=None*)

Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.cql_impl.CQLImpl, *d3rlpy.algos.qlearning.cql.CQLConfig*]

get_action_type()

Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)

Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

DiscreteCQL

class d3rlpy.algos.**DiscreteCQLConfig**(*batch_size=32*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *learning_rate=6.25e-05*, *optim_factory=<factory>*, *encoder_factory=<factory>*, *q_func_factory=<factory>*, *n_critics=1*, *target_update_interval=8000*, *alpha=1.0*)

Bases: d3rlpy.base.LearnableConfig

Config of 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

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **learning_rate** (*float*) – Learning rate.
- **optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – Optimizer factory.
- **encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **n_critics** (*int*) – Number of Q functions for ensemble.
- **target_update_interval** (*int*) – Interval to synchronize the target network.
- **alpha** (*float*) – math:*alpha* value above.
- **action_scaler** (*Optional[d3rlpy.preprocessing.action_scalers.ActionScaler]*) –

Return type None

create(*device=False*)

Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, cuda:0 will be used. If the value is integer, cuda:<device> will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.cql.DiscreteCQL*

class d3rlpy.algos.**DiscreteCQL**(*config*, *device*, *impl=None*)

Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.cql_impl.DiscreteCQLImpl, *d3rlpy.algos.qlearning.cql.DiscreteCQLConfig*]

get_action_type()

Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

AWAC

class d3rlpy.algos.**AWACConfig**(*batch_size=1024*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *actor_learning_rate=0.0003*, *critic_learning_rate=0.0003*, *actor_optim_factory=<factory>*, *critic_optim_factory=<factory>*, *actor_encoder_factory=<factory>*, *critic_encoder_factory=<factory>*, *q_func_factory=<factory>*, *tau=0.005*, *lam=1.0*, *n_action_samples=1*, *n_critics=2*, *update_actor_interval=1*)

Bases: d3rlpy.base.LearnableConfig

Config of 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

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **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*) – Encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the critic.

- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **tau** (*float*) – Target network synchronization coefficiency.
- **lam** (*float*) – λ for weight calculation.
- **n_action_samples** (*int*) – Number of sampled actions to calculate $A^{\pi}(s_t, a_t)$.
- **n_critics** (*int*) – Number of Q functions for ensemble.
- **update_actor_interval** (*int*) – Interval to update policy function.

Return type None

create(*device=False*)
Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.awac.AWAC*

class d3rlpy.algos.**AWAC**(*config*, *device*, *impl=None*)
Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.awac_impl.AWACImpl, *d3rlpy.algos.qlearning.awac.AWACConfig*]

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

PLAS

class d3rlpy.algos.**PLASConfig**(*batch_size=100*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *actor_learning_rate=0.0001*, *critic_learning_rate=0.001*, *imitator_learning_rate=0.0001*, *actor_optim_factory=<factory>*, *critic_optim_factory=<factory>*, *imitator_optim_factory=<factory>*, *actor_encoder_factory=<factory>*, *critic_encoder_factory=<factory>*, *imitator_encoder_factory=<factory>*, *q_func_factory=<factory>*, *tau=0.005*, *n_critics=2*, *update_actor_interval=1*, *lam=0.75*, *warmup_steps=500000*, *beta=0.5*)
Bases: d3rlpy.base.LearnableConfig

Config of 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 β is a parameter of the decoder in Conditional VAE.

References

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

Parameters

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **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*) – Encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the critic.
- **imitator_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the conditional VAE.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **tau** (*float*) – Target network synchronization coefficiency.
- **n_critics** (*int*) – 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*) – Number of steps to warmup the VAE.
- **beta** (*float*) – KL reguralization term for Conditional VAE.

Return type None

create(*device=False*)
Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.plas.PLAS*

class d3rlpy.algos.**PLAS**(*config*, *device*, *impl=None*)
Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.plas_impl.PLASImpl, *d3rlpy.algos.qlearning.plas.PLASConfig*]

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

PLAS+P

class d3rlpy.algos.**PLASWithPerturbationConfig**(*batch_size=100*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *actor_learning_rate=0.0001*, *critic_learning_rate=0.001*, *imitator_learning_rate=0.0001*, *actor_optim_factory=<factory>*, *critic_optim_factory=<factory>*, *imitator_optim_factory=<factory>*, *actor_encoder_factory=<factory>*, *critic_encoder_factory=<factory>*, *imitator_encoder_factory=<factory>*, *q_func_factory=<factory>*, *tau=0.005*, *n_critics=2*, *update_actor_interval=1*, *lam=0.75*, *warmup_steps=500000*, *beta=0.5*, *action_flexibility=0.05*)
Bases: *d3rlpy.algos.qlearning.plas.PLASConfig*

Config of 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

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **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*) – Encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the critic.
- **imitator_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the conditional VAE.
- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **tau** (*float*) – Target network synchronization coefficiency.
- **n_critics** (*int*) – 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*) – Number of steps to warmup the VAE.
- **beta** (*float*) – KL reguralization term for Conditional VAE.

Return type None

create(*device=False*)

Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.plas.PLASWithPerturbation*

class d3rlpy.algos.**PLASWithPerturbation**(*config*, *device*, *impl=None*)

Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.plas_impl.PLASImpl, *d3rlpy.algos.qlearning.plas.PLASConfig*]

TD3+BC

class d3rlpy.algos.**TD3PlusBCConfig**(*batch_size=256*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *actor_learning_rate=0.0003*, *critic_learning_rate=0.0003*, *actor_optim_factory=<factory>*, *critic_optim_factory=<factory>*, *actor_encoder_factory=<factory>*, *critic_encoder_factory=<factory>*, *q_func_factory=<factory>*, *tau=0.005*, *n_critics=2*, *target_smoothing_sigma=0.2*, *target_smoothing_clip=0.5*, *alpha=2.5*, *update_actor_interval=2*)

Bases: d3rlpy.base.LearnableConfig

Config of 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

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **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*) – Encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the critic.

- **q_func_factory** (*d3rlpy.models.q_functions.QFunctionFactory*) – Q function factory.
- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **tau** (*float*) – Target network synchronization coefficiency.
- **n_critics** (*int*) – 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*) – α value.
- **update_actor_interval** (*int*) – Interval to update policy function described as *delayed policy update* in the paper.

Return type None

create(*device=False*)

Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.td3_plus_bc.TD3PlusBC*

class d3rlpy.algos.**TD3PlusBC**(*config*, *device*, *impl=None*)

Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.td3_plus_bc_impl.TD3PlusBCImpl, *d3rlpy.algos.qlearning.td3_plus_bc.TD3PlusBCConfig*]

get_action_type()

Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)

Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

IQL

class d3rlpy.algos.**IQLConfig**(*batch_size=256*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *actor_learning_rate=0.0003*, *critic_learning_rate=0.0003*, *actor_optim_factory=<factory>*, *critic_optim_factory=<factory>*, *actor_encoder_factory=<factory>*, *critic_encoder_factory=<factory>*, *value_encoder_factory=<factory>*, *tau=0.005*, *n_critics=2*, *expectile=0.7*, *weight_temp=3.0*, *max_weight=100.0*)

Bases: d3rlpy.base.LearnableConfig

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.

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

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

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **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*) – Encoder factory for the actor.
- **critic_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the critic.
- **value_encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory for the value function.

- **batch_size** (*int*) – Mini-batch size.
- **gamma** (*float*) – Discount factor.
- **tau** (*float*) – Target network synchronization coefficiency.
- **n_critics** (*int*) – Number of Q functions for ensemble.
- **expectile** (*float*) – Expectile value for value function training.
- **weight_temp** (*float*) – Inverse temperature value represented as β.
- **max_weight** (*float*) – Maximum advantage weight value to clip.

Return type None

create(*device=False*)
Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.iql.IQL*

class d3rlpy.algos.**IQL**(*config*, *device*, *impl=None*)
Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.torch.iql_impl.IQLImpl, *d3rlpy.algos.qlearning.iql.IQLConfig*]

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

RandomPolicy

class d3rlpy.algos.**RandomPolicyConfig**(*batch_size=256*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *distribution='uniform'*, *normal_std=1.0*)
Bases: d3rlpy.base.LearnableConfig

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

- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **distribution** (*str*) – Random distribution. Available options are `['uniform', 'normal']`.

- **normal_std** (*float*) – Standard deviation of the normal distribution. This is only used when `distribution='normal'`.
- **batch_size** (*int*) –
- **gamma** (*float*) –
- **observation_scaler** (*Optional[d3rlpy.preprocessing.observation_scalers.ObservationScaler]*) –
- **reward_scaler** (*Optional[d3rlpy.preprocessing.reward_scalers.RewardScaler]*) –

Return type None

create(*device=False*)

Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.random_policy.RandomPolicy*

class d3rlpy.algos.**RandomPolicy**(*config*)

Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[None, *d3rlpy.algos.qlearning.random_policy.RandomPolicyConfig*]

get_action_type()

Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)

Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

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, Sequence[numpy.ndarray]]*) – Observations

Returns Greedy actions

Return type numpy.ndarray

predict_value(*x*, *action*)

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,)
```

Parameters

- **x** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observations
- **action** (*numpy.ndarray*) – Actions

Returns Predicted action-values

Return type numpy.ndarray

sample_action(*x*)

Returns sampled actions.

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

Parameters **x** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observations.

Returns Sampled actions.

Return type numpy.ndarray

DiscreteRandomPolicy

class d3rlpy.algos.**DiscreteRandomPolicyConfig**(*batch_size=256*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*)

Bases: d3rlpy.base.LearnableConfig

Random Policy for discrete control algorithm.

This is designed for data collection and lightweight interaction tests. `fit` and `fit_online` methods will raise exceptions.

Parameters

- **batch_size** (*int*) –
- **gamma** (*float*) –
- **observation_scaler** (*Optional[d3rlpy.preprocessing.observation_scalers.ObservationScaler]*) –
- **action_scaler** (*Optional[d3rlpy.preprocessing.action_scalers.ActionScaler]*) –

- **reward_scaler** (*Optional[d3rlpy.preprocessing.reward_scalers.RewardScaler]*) –

Return type None

create(*device=False*)
Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.qlearning.random_policy.DiscreteRandomPolicy*

class d3rlpy.algos.**DiscreteRandomPolicy**(*config*)
Bases: *d3rlpy.algos.qlearning.base.QLearningAlgoBase*[None, *d3rlpy.algos.qlearning.random_policy.DiscreteRandomPolicyConfig*]

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

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, Sequence[numpy.ndarray]]*) – Observations

Returns Greedy actions

Return type numpy.ndarray

predict_value(*x*, *action*)
Returns predicted action-values.

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

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

(continues on next page)

(continued from previous page)

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

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

Parameters

- **x** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observations
- **action** (*numpy.ndarray*) – Actions

Returns Predicted action-values

Return type numpy.ndarray

sample_action(*x*)

Returns sampled actions.

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

Parameters **x** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observations.

Returns Sampled actions.

Return type numpy.ndarray

4.1.3 Decision Transformer

Decision Transformer-based algorithms usually require tricky interaction codes for evaluation. In d3rlpy, those algorithms provide `as_stateful_wrapper` method to easily integrate the algorithm into your system.

```
import d3rlpy

dataset, env = d3rlpy.datasets.get_pendulum()

dt = d3rlpy.algos.DecisionTransformerConfig().create(device="cuda:0")

# offline training
dt.fit(
   dataset,
   n_steps=100000,
   n_steps_per_epoch=1000,
   eval_env=env,
   eval_target_return=0,  # specify target environment return
)

# wrap as stateful actor for interaction
actor = dt.as_stateful_wrapper(target_return=0)

# interaction
observation, reward = env.reset(), 0.0
while True:
```

(continues on next page)

(continued from previous page)

```
    action = actor.predict(observation, reward)
    observation, reward, done, truncated, _ = env.step(action)
    if done or truncated:
        break

# reset history
actor.reset()
```

TransformerAlgoBase

class d3rlpy.algos.**TransformerAlgoBase**(*config*, *device*, *impl=None*)

Bases: Generic[d3rlpy.algos.transformer.base.TTransformerImpl, d3rlpy.algos.transformer.base.TTransformerConfig], *d3rlpy.base.LearnableBase*[d3rlpy.algos.transformer.base.TTransformerImpl, d3rlpy.algos.transformer.base.TTransformerConfig]

as_stateful_wrapper(*target_return*)

Returns a wrapped Transformer algorithm for stateful decision making.

Parameters **target_return** (*float*) – Target environment return.

Returns StatefulTransformerWrapper object.

Return type d3rlpy.algos.transformer.base.StatefulTransformerWrapper[d3rlpy.algos.transformer.base.TTransformerIn
d3rlpy.algos.transformer.base.TTransformerConfig]

fit(*dataset*, *n_steps*, *n_steps_per_epoch=10000*, *experiment_name=None*, *with_timestamp=True*, *logger_adapter=<d3rlpy.logging.file_adapter.FileAdapterFactory object>*, *show_progress=True*, *eval_env=None*, *eval_target_return=None*, *save_interval=1*, *callback=None*)

Trains with given dataset.

Parameters

- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – Offline dataset to train.
- **n_steps** (*int*) – Number of steps to train.
- **n_steps_per_epoch** (*int*) – Number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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.
- **logger_adapter** (d3rlpy.logging.logger.LoggerAdapterFactory) – LoggerAdapterFactory object.
- **show_progress** (*bool*) – Flag to show progress bar for iterations.
- **eval_env** (*Optional[gym.core.Env[Any, Any]]*) – Evaluation environment.
- **eval_target_return** (*Optional[float]*) – Evaluation return target.
- **save_interval** (*int*) – Interval to save parameters.
- **callback** (*Optional[Callable[[typing_extensions.Self, int, int], None]]*) – Callable function that takes `(algo, epoch, total_step)` , which is called every step.

Return type None

abstract inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchTrajectoryMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

predict(*inpt*)
Returns action.

This is for internal use. For evaluation, use `StatefulTransformerWrapper` instead.

Parameters **inpt** (*d3rlpy.algos.transformer.inputs.TransformerInput*) – Sequence input.

Returns Action.

Return type numpy.ndarray

update(*batch*)
Update parameters with mini-batch of data.

Parameters **batch** (*d3rlpy.dataset.mini_batch.TrajectoryMiniBatch*) – Mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

Decision Transformer

class d3rlpy.algos.**DecisionTransformerConfig**(*batch_size=64*, *gamma=0.99*, *observation_scaler=None*, *action_scaler=None*, *reward_scaler=None*, *context_size=20*, *learning_rate=0.0001*, *encoder_factory=<factory>*, *optim_factory=<factory>*, *num_heads=1*, *max_timestep=1000*, *num_layers=3*, *attn_dropout=0.1*, *resid_dropout=0.1*, *embed_dropout=0.1*, *activation_type='relu'*, *position_encoding_type='simple'*, *warmup_steps=10000*, *clip_grad_norm=0.25*)
Bases: d3rlpy.algos.transformer.base.TransformerConfig

Config of Decision Transformer.

Decision Transformer solves decision-making problems as a sequence modeling problem.

References

- Chen at el., Decision Transformer: Reinforcement Learning via Sequence Modeling.

Parameters

- **observation_scaler** (*d3rlpy.preprocessing.ObservationScaler*) – Observation preprocessor.
- **action_scaler** (*d3rlpy.preprocessing.ActionScaler*) – Action preprocessor.
- **reward_scaler** (*d3rlpy.preprocessing.RewardScaler*) – Reward preprocessor.
- **context_size** (*int*) – Prior sequence length.
- **batch_size** (*int*) – Mini-batch size.
- **learning_rate** (*float*) – Learning rate.
- **encoder_factory** (*d3rlpy.models.encoders.EncoderFactory*) – Encoder factory.
- **optim_factory** (d3rlpy.models.optimizers.OptimizerFactory) – Optimizer factory.
- **num_heads** (*int*) – Number of attention heads.
- **max_timestep** (*int*) – Maximum environmental timestep.
- **num_layers** (*int*) – Number of attention blocks.
- **attn_dropout** (*float*) – Dropout probability for attentions.
- **resid_dropout** (*float*) – Dropout probability for residual connection.
- **embed_dropout** (*float*) – Dropout probability for embeddings.
- **activation_type** (*str*) – Type of activation function.
- **position_encoding_type** (*str*) – Type of positional encoding (`simple` or `global`).
- **warmup_steps** (*int*) – Warmup steps for learning rate scheduler.
- **clip_grad_norm** (*float*) – Norm of gradient clipping.
- **gamma** (*float*) –

Return type None

create(*device=False*)

Returns algorithm object.

Parameters **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type *d3rlpy.algos.transformer.decision_transformer.DecisionTransformer*

class d3rlpy.algos.**DecisionTransformer**(*config*, *device*, *impl=None*)

Bases: *d3rlpy.algos.transformer.base.TransformerAlgoBase*[d3rlpy.algos.transformer.torch.decision_transformer_impl.DecisionTransformerImpl, *d3rlpy.algos.transformer.decision_transformer.DecisionTransformerConfig*]

get_action_type()

Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchTrajectoryMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

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.

```
import d3rlpy

cql = d3rlpy.algos.CQLConfig(q_func_factory=d3rlpy.models.QRQFunctionFactory())
```

Also you can change hyper parameters.

```
q_func = d3rlpy.models.QRQFunctionFactory(n_quantiles=32)

cql = d3rlpy.algos.CQLConfig(q_func_factory=q_func).create()
```

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.MeanQFunctionFactory	Standard Q function factory class.
d3rlpy.models.QRQFunctionFactory	Quantile Regression Q function factory class.
d3rlpy.models.IQNQFunctionFactory	Implicit Quantile Network Q function factory class.

4.2.1 d3rlpy.models.MeanQFunctionFactory

class d3rlpy.models.**MeanQFunctionFactory**(*share_encoder=False*)
Standard Q function factory class.

This is the standard Q function factory class.

References

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

Parameters **share_encoder** (*bool*) – flag to share encoder over multiple Q functions.

Return type None

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

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()
Returns Q function type.

Returns Q function type.

Return type str

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

Attributes

share_encoder: bool = False

4.2.2 d3rlpy.models.QRQFunctionFactory

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

Return type None

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

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()
Returns Q function type.

Returns Q function type.

Return type str

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –

- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

Attributes

n_quantiles: int = 32

share_encoder: bool = False

4.2.3 d3rlpy.models.IQNQFunctionFactory

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

Implicit Quantile Network Q function factory class.

References

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

Parameters

- **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.

Return type None

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

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Returns Q function type.

Returns Q function type.

Return type str

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

`serialize_to_dict()`

Return type Dict[str, Any]

`to_dict(encode_json=False)`

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

`to_json(*, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, indent=None, separators=None, default=None, sort_keys=False, **kw)`

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

Attributes

`embed_size: int = 64`

`n_greedy_quantiles: int = 32`

`n_quantiles: int = 64`

`share_encoder: bool = False`

4.3 Replay Buffer

You can also check advanced use cases at examples directory.

4.3.1 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.

```
import d3rlpy

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

# save as HDF5
with open("dataset.h5", "wb") as f:
    dataset.dump(f)

# load from HDF5
with open("dataset.h5", "wb") as f:
    new_dataset = d3rlpy.dataset.ReplayBuffer.load(f, d3rlpy.dataset.InfiniteBuffer())
```

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)
```

`MDPDataset` is actually a shortcut of `ReplayBuffer` class.

d3rlpy.dataset.MDPDataset	Backward-compability class of MDPDataset.

d3rlpy.dataset.MDPDataset

class d3rlpy.dataset.**MDPDataset**(*observations*, *actions*, *rewards*, *terminals*, *timeouts=None*, *transition_picker=None*, *trajectory_slicer=None*)

Backward-compability class of MDPDataset.

This is a wrapper class that has a backward-compatible constructor interface.

Parameters

- **observations** (*ObservationSequence*) – Observations.
- **actions** (*np.ndarray*) – Actions.
- **rewards** (*np.ndarray*) – Rewards.
- **terminals** (*np.ndarray*) – Environmental terminal flags.
- **timeouts** (*np.ndarray*) – Timeouts.
- **transition_picker** (*Optional[TransitionPickerProtocol]*) – Transition picker implementation for Q-learning-based algorithms. If `None` is given, `BasicTransitionPicker` is used by default.

- **trajectory_slicer** (*Optional[TrajectorySlicerProtocol]*) – Trajectory slicer implementation for Transformer-based algorithms. If `None` is given, `BasicTrajectorySlicer` is used by default.

Methods

append(*observation*, *action*, *reward*)
Appends observation, action and reward to buffer.

Parameters

- **observation** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observation.
- **action** (*Union[int, numpy.ndarray]*) – Action.
- **reward** (*Union[float, numpy.ndarray]*) – Reward.

Return type None

append_episode(*episode*)
Appends episode to buffer.

Parameters **episode** (*d3rlpy.dataset.components.EpisodeBase*) – Episode.

Return type None

clip_episode(*terminated*)
Clips current episode.

Parameters **terminated** (*bool*) – Flag to represent environmental termination. This flag should be `False` if the episode is terminated by timeout.

Return type None

dump(*f*)
Dumps buffer data.

```
with open('dataset.h5', 'wb') as f:
    replay_buffer.dump(f)
```

Parameters **f** (*BinaryIO*) – IO object to write to.

Return type None

classmethod from_episode_generator(*episode_generator*, *buffer*, *transition_picker=None*, *trajectory_slicer=None*, *writer_preprocessor=None*)
Builds ReplayBuffer from episode generator.

Parameters

- **episode_generator** (*d3rlpy.dataset.episode_generator.EpisodeGeneratorProtocol*) – Episode generator implementation.
- **buffer** (*d3rlpy.dataset.buffers.BufferProtocol*) – Buffer implementation.
- **transition_picker** (*Optional[d3rlpy.dataset.transition_pickers.TransitionPickerProtocol]*) – Transition picker implementation for Q-learning-based algorithms.

- **trajectory_slicer** (*Optional[d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol]*) – Trajectory slicer implementation for Transformer-based algorithms.
- **writer_preprocessor** (*Optional[d3rlpy.dataset.writers.WriterPreprocessProtocol]*) – Writer preprocessor implementation.

Returns Replay buffer.

Return type *d3rlpy.dataset.replay_buffer.ReplayBuffer*

classmethod load(*f*, *buffer*, *episode_cls=<class 'd3rlpy.dataset.components.Episode'>*, *transition_picker=None*, *trajectory_slicer=None*, *writer_preprocessor=None*)

Builds ReplayBuffer from dumped data.

This method reconstructs replay buffer dumped by `dump` method.

```
with open('dataset.h5', 'rb') as f:
    replay_buffer = ReplayBuffer.load(f, buffer)
```

Parameters

- **f** (*BinaryIO*) – IO object to read from.
- **buffer** (d3rlpy.dataset.buffers.BufferProtocol) – Buffer implementation.
- **episode_cls** (*Type[d3rlpy.dataset.components.EpisodeBase]*) – Eisode class used to reconstruct data.
- **transition_picker** (*Optional[d3rlpy.dataset.transition_pickers.TransitionPickerProtocol]*) – Transition picker implementation for Q-learning-based algorithms.
- **trajectory_slicer** (*Optional[d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol]*) – Trajectory slicer implementation for Transformer-based algorithms.
- **writer_preprocessor** (*Optional[d3rlpy.dataset.writers.WriterPreprocessProtocol]*) – Writer preprocessor implementation.

Returns Replay buffer.

Return type *d3rlpy.dataset.replay_buffer.ReplayBuffer*

sample_trajectory(*length*)

Samples a partial trajectory.

Parameters **length** (*int*) – Length of partial trajectory.

Returns Partial trajectory.

Return type d3rlpy.dataset.components.PartialTrajectory

sample_trajectory_batch(*batch_size*, *length*)

Samples a mini-batch of partial trajectories.

Parameters

- **batch_size** (*int*) – Mini-batch size.
- **length** (*int*) – Length of partial trajectories.

Returns Mini-batch.

Return type d3rlpy.dataset.mini_batch.TrajectoryMiniBatch

sample_transition()
Samples a transition.

Returns Transition.

Return type d3rlpy.dataset.components.Transition

sample_transition_batch(*batch_size*)
Samples a mini-batch of transitions.

Parameters **batch_size** (*int*) – Mini-batch size.

Returns Mini-batch.

Return type d3rlpy.dataset.mini_batch.TransitionMiniBatch

size()
Returns number of episodes.

Returns Number of episodes.

Return type int

Attributes

buffer
Returns buffer.

Returns Buffer.

episodes
Returns sequence of episodes.

Returns Sequence of episodes.

trajectory_slicer
Returns trajectory slicer.

Returns Trajectory slicer.

transition_count
Returns number of transitions.

Returns Number of transitions.

transition_picker
Returns transition picker.

Returns Transition picker.

4.3.2 Replay Buffer

`ReplayBuffer` is a class that represents an experience replay buffer in d3rlpy. In d3rlpy, `ReplayBuffer` is a highly moduralized interface for flexibility. You can compose sub-components of `ReplayBuffer`, `Buffer`, `TransitionPicker`, *TrajectorySlicer* and `WriterPreprocess` to customize experiments.

```
import d3rlpy

# Buffer component
buffer = d3rlpy.dataset.FIFOBuffer(limit=100000)
```

(continues on next page)

(continued from previous page)

```
# TransitionPicker component
transition_picker = d3rlpy.dataset.BasicTransitionPicker()

# TrajectorySlicer component
trajectory_slicer = d3rlpy.dataset.BasicTrajectorySlicer()

# WriterPreprocess component
writer_preprocessor = d3rlpy.dataset.BasicWriterPreprocess()

# Need to specify signatures of observations, actions and rewards

# Option 1: Initialize with Gym environment
import gym
env = gym.make("Pendulum-v1")
replay_buffer = d3rlpy.dataset.ReplayBuffer(
   buffer=buffer,
   transition_picker=transition_picker,
   trajectory_slicer=trajectory_slicer,
   writer_preprocessor=writer_preprocessor,
   env=env,
)

# Option 2: Initialize with pre-collected dataset
dataset, _ = d3rlpy.datasets.get_pendulum()
replay_buffer = d3rlpy.dataset.ReplayBuffer(
   buffer=buffer,
   transition_picker=transition_picker,
   trajectory_slicer=trajectory_slicer,
   writer_preprocessor=writer_preprocessor,
   episodes=dataset.episodes,
)

# Option 3: Initialize with manually specified signatures
observation_signature = d3rlpy.dataset.Signature(shape=[(3,)], dtype=[np.float32])
action_signature = d3rlpy.dataset.Signature(shape=[(1,)], dtype=[np.float32])
reward_signature = d3rlpy.dataset.Signature(shape=[(1,)], dtype=[np.float32])
replay_buffer = d3rlpy.dataset.ReplayBuffer(
   buffer=buffer,
   transition_picker=transition_picker,
   trajectory_slicer=trajectory_slicer,
   writer_preprocessor=writer_preprocessor,
   observation_signature=observation_signature,
   action_signature=action_signature,
   reward_signature=reward_signature,
)

# shortcut
replay_buffer = d3rlpy.dataset.create_fifo_replay_buffer(limit=100000, env=env)
```

d3rlpy.dataset.ReplayBuffer	Replay buffer for experience replay.
d3rlpy.dataset.create_infinite_replay_buffer	Builds infinite replay buffer.

continues on next page

Table 3 – continued from previous page

d3rlpy.dataset.create_fifo_replay_buffer	Builds FIFO replay buffer.

d3rlpy.dataset.ReplayBuffer

class d3rlpy.dataset.**ReplayBuffer**(*buffer*, *transition_picker=None*, *trajectory_slicer=None*, *writer_preprocessor=None*, *episodes=None*, *env=None*, *observation_signature=None*, *action_signature=None*, *reward_signature=None*, *cache_size=10000*)

Replay buffer for experience replay.

This replay buffer implementation is used for both online and offline training in d3rlpy. To determine shapes of observations, actions and rewards, one of `episodes`, `env` and signatures must be provided.

```
from d3rlpy.dataset import FIFOBuffer, ReplayBuffer, Signature

buffer = FIFOBuffer(limit=1000000)

# initialize with pre-collected episodes
replay_buffer = ReplayBuffer(buffer=buffer, episodes=<episodes>)

# initialize with Gym
replay_buffer = ReplayBuffer(buffer=buffer, env=<env>)

# initialize with manually specified signatures
replay_buffer = ReplayBuffer(
    buffer=buffer,
    observation_signature=Signature(dtype=[<dtype>], shape=[<shape>]),
    action_signature=Signature(dtype=[<dtype>], shape=[<shape>]),
    reward_signature=Signature(dtype=[<dtype>], shape=[<shape>]),
)
```

Parameters

- **buffer** (d3rlpy.dataset.BufferProtocol) – Buffer implementation.
- **transition_picker** (*Optional[d3rlpy.dataset.TransitionPickerProtocol]*) – Transition picker implementation for Q-learning-based algorithms. If `None` is given, `BasicTransitionPicker` is used by default.
- **trajectory_slicer** (*Optional[d3rlpy.dataset.TrajectorySlicerProtocol]*) – Trajectory slicer implementation for Transformer-based algorithms. If `None` is given, `BasicTrajectorySlicer` is used by default.
- **writer_preprocessor** (*Optional[d3rlpy.dataset.WriterPreprocessProtocol]*) – Writer preprocessor implementation. If `None` is given, `BasicWriterPreprocess` is used by default.
- **episodes** (*Optional[Sequence[d3rlpy.dataset.EpisodeBase]]*) – List of episodes to initialize replay buffer.
- **env** (*Optional[gym.Env]*) – Gym environment to extract shapes of observations and action.
- **observation_signature** (*Optional[d3rlpy.dataset.Signature]*) – Signature of observation.

- **action_signature** (*Optional[d3rlpy.dataset.Signature]*) – Signature of action.
- **reward_signature** (*Optional[d3rlpy.dataset.Signature]*) – Signature of reward.
- **cache_size** (*int*) – Size of cache to record active episode history used for online training. `cache_size` needs to be greater than the maximum possible episode length.

Methods

append(*observation*, *action*, *reward*)
Appends observation, action and reward to buffer.

Parameters

- **observation** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observation.
- **action** (*Union[int, numpy.ndarray]*) – Action.
- **reward** (*Union[float, numpy.ndarray]*) – Reward.

Return type None

append_episode(*episode*)
Appends episode to buffer.

Parameters **episode** (*d3rlpy.dataset.components.EpisodeBase*) – Episode.

Return type None

clip_episode(*terminated*)
Clips current episode.

Parameters **terminated** (*bool*) – Flag to represent environmental termination. This flag should be `False` if the episode is terminated by timeout.

Return type None

dump(*f*)
Dumps buffer data.

```
with open('dataset.h5', 'wb') as f:
    replay_buffer.dump(f)
```

Parameters **f** (*BinaryIO*) – IO object to write to.

Return type None

classmethod from_episode_generator(*episode_generator*, *buffer*, *transition_picker=None*, *trajectory_slicer=None*, *writer_preprocessor=None*)
Builds ReplayBuffer from episode generator.

Parameters

- **episode_generator** (*d3rlpy.dataset.episode_generator.EpisodeGeneratorProtocol*) – Episode generator implementation.
- **buffer** (*d3rlpy.dataset.buffers.BufferProtocol*) – Buffer implementation.
- **transition_picker** (*Optional[d3rlpy.dataset.transition_pickers.TransitionPickerProtocol]*) – Transition picker implementation for Q-learning-based algorithms.

- **trajectory_slicer** (*Optional[d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol]*) – Trajectory slicer implementation for Transformer-based algorithms.
- **writer_preprocessor** (*Optional[d3rlpy.dataset.writers.WriterPreprocessProtocol]*) – Writer preprocessor implementation.

Returns Replay buffer.

Return type *d3rlpy.dataset.replay_buffer.ReplayBuffer*

classmethod load(*f*, *buffer*, *episode_cls=<class 'd3rlpy.dataset.components.Episode'>*, *transition_picker=None*, *trajectory_slicer=None*, *writer_preprocessor=None*)

Builds ReplayBuffer from dumped data.

This method reconstructs replay buffer dumped by `dump` method.

```
with open('dataset.h5', 'rb') as f:
    replay_buffer = ReplayBuffer.load(f, buffer)
```

Parameters

- **f** (*BinaryIO*) – IO object to read from.
- **buffer** (d3rlpy.dataset.buffers.BufferProtocol) – Buffer implementation.
- **episode_cls** (*Type[d3rlpy.dataset.components.EpisodeBase]*) – Eisode class used to reconstruct data.
- **transition_picker** (*Optional[d3rlpy.dataset.transition_pickers.TransitionPickerProtocol]*) – Transition picker implementation for Q-learning-based algorithms.
- **trajectory_slicer** (*Optional[d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol]*) – Trajectory slicer implementation for Transformer-based algorithms.
- **writer_preprocessor** (*Optional[d3rlpy.dataset.writers.WriterPreprocessProtocol]*) – Writer preprocessor implementation.

Returns Replay buffer.

Return type *d3rlpy.dataset.replay_buffer.ReplayBuffer*

sample_trajectory(*length*)

Samples a partial trajectory.

Parameters **length** (*int*) – Length of partial trajectory.

Returns Partial trajectory.

Return type d3rlpy.dataset.components.PartialTrajectory

sample_trajectory_batch(*batch_size*, *length*)

Samples a mini-batch of partial trajectories.

Parameters

- **batch_size** (*int*) – Mini-batch size.
- **length** (*int*) – Length of partial trajectories.

Returns Mini-batch.

Return type d3rlpy.dataset.mini_batch.TrajectoryMiniBatch

sample_transition()
Samples a transition.

Returns Transition.

Return type d3rlpy.dataset.components.Transition

sample_transition_batch(*batch_size*)
Samples a mini-batch of transitions.

Parameters **batch_size** (*int*) – Mini-batch size.

Returns Mini-batch.

Return type d3rlpy.dataset.mini_batch.TransitionMiniBatch

size()
Returns number of episodes.

Returns Number of episodes.

Return type int

Attributes

buffer
Returns buffer.

Returns Buffer.

episodes
Returns sequence of episodes.

Returns Sequence of episodes.

trajectory_slicer
Returns trajectory slicer.

Returns Trajectory slicer.

transition_count
Returns number of transitions.

Returns Number of transitions.

transition_picker
Returns transition picker.

Returns Transition picker.

d3rlpy.dataset.create_infinite_replay_buffer

d3rlpy.dataset.**create_infinite_replay_buffer**(*episodes=None*, *transition_picker=None*, *trajectory_slicer=None*, *writer_preprocessor=None*, *env=None*)

Builds infinite replay buffer.

This function is a shortcut alias to build replay buffer with `InfiniteBuffer`.

Parameters

- **episodes** (*Optional[Sequence[d3rlpy.dataset.components.EpisodeBase]]*) – List of episodes to initialize replay buffer.

- **transition_picker** (*Optional[d3rlpy.dataset.transition_pickers.TransitionPickerProtocol]*) – Transition picker implementation for Q-learning-based algorithms. If `None` is given, `BasicTransitionPicker` is used by default.
- **trajectory_slicer** (*Optional[d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol]*) – Trajectory slicer implementation for Transformer-based algorithms. If `None` is given, `BasicTrajectorySlicer` is used by default.
- **writer_preprocessor** (*Optional[d3rlpy.dataset.writers.WriterPreprocessProtocol]*) – Writer preprocessor implementation. If `None` is given, `BasicWriterPreprocess` is used by default.
- **env** (*Optional[gym.core.Env[numpy.ndarray, Any]]*) – Gym environment to extract shapes of observations and action.

Returns Replay buffer.

Return type *d3rlpy.dataset.replay_buffer.ReplayBuffer*

d3rlpy.dataset.create_fifo_replay_buffer

d3rlpy.dataset.**create_fifo_replay_buffer**(*limit*, *episodes=None*, *transition_picker=None*, *trajectory_slicer=None*, *writer_preprocessor=None*, *env=None*)

Builds FIFO replay buffer.

This function is a shortcut alias to build replay buffer with `FIFOBuffer`.

Parameters

- **limit** (*int*) – Maximum capacity of FIFO buffer.
- **episodes** (*Optional[Sequence[d3rlpy.dataset.components.EpisodeBase]]*) – List of episodes to initialize replay buffer.
- **transition_picker** (*Optional[d3rlpy.dataset.transition_pickers.TransitionPickerProtocol]*) – Transition picker implementation for Q-learning-based algorithms. If `None` is given, `BasicTransitionPicker` is used by default.
- **trajectory_slicer** (*Optional[d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol]*) – Trajectory slicer implementation for Transformer-based algorithms. If `None` is given, `BasicTrajectorySlicer` is used by default.
- **writer_preprocessor** (*Optional[d3rlpy.dataset.writers.WriterPreprocessProtocol]*) – Writer preprocessor implementation. If `None` is given, `BasicWriterPreprocess` is used by default.
- **env** (*Optional[gym.core.Env[numpy.ndarray, Any]]*) – Gym environment to extract shapes of observations and action.

Returns Replay buffer.

Return type *d3rlpy.dataset.replay_buffer.ReplayBuffer*

4.3.3 Buffer

`Buffer` is a list-like component that stores and drops transitions.

d3rlpy.dataset.BufferProtocol	Interface of Buffer.
d3rlpy.dataset.InfiniteBuffer	Buffer with unlimited capacity.
d3rlpy.dataset.FIFOBuffer	FIFO buffer.

d3rlpy.dataset.BufferProtocol

class d3rlpy.dataset.**BufferProtocol**(**args*, ***kwds*)
Interface of Buffer.

Methods

__getitem__(*index*)

Parameters **index** (*int*) –

Return type Tuple[d3rlpy.dataset.components.EpisodeBase, int]

append(*episode*, *index*)
Adds transition to buffer.

Parameters

- **episode** (*d3rlpy.dataset.components.EpisodeBase*) – Episode object.
- **index** (*int*) – Transition index.

Return type None

Attributes

episodes
Returns list of episodes.

Returns List of saved episodes.

transition_count
Returns the number of transitions.

Returns Number of transitions.

d3rlpy.dataset.InfiniteBuffer

class d3rlpy.dataset.**InfiniteBuffer**(**args*, ***kwds*)
Buffer with unlimited capacity.

Methods

__getitem__(*index*)

Parameters **index** (*int*) –

Return type Tuple[d3rlpy.dataset.components.EpisodeBase, int]

__len__()

Return type int

append(*episode*, *index*)
Adds transition to buffer.

Parameters

- **episode** (*d3rlpy.dataset.components.EpisodeBase*) – Episode object.
- **index** (*int*) – Transition index.

Return type None

Attributes

episodes

transition_count

d3rlpy.dataset.FIFOBuffer

class d3rlpy.dataset.**FIFOBuffer**(**args*, ***kwds*)
FIFO buffer.

Parameters **limit** (*int*) – buffer capacity.

Methods

__getitem__(*index*)

Parameters **index** (*int*) –

Return type Tuple[d3rlpy.dataset.components.EpisodeBase, int]

__len__()

Return type int

append(*episode*, *index*)
Adds transition to buffer.

Parameters

- **episode** (*d3rlpy.dataset.components.EpisodeBase*) – Episode object.
- **index** (*int*) – Transition index.

Return type None

Attributes

episodes

transition_count

4.3.4 TransitionPicker

TransitionPicker is a component that defines how to pick transition data used for Q-learning-based algorithms. You can also implement your own TransitionPicker for custom experiments.

```
import d3rlpy

# Example TransitionPicker that simply picks transition
class CustomTransitionPicker(d3rlpy.dataset.TransitionPickerProtocol):
    def __call__(self, episode: d3rlpy.dataset.EpisodeBase, index: int) -> d3rlpy.
↪dataset.Transition:
        observation = episode.observations[index]
        is_terminal = episode.terminated and index == episode.size() - 1
        if is_terminal:
            next_observation = d3rlpy.dataset.create_zero_observation(observation)
        else:
            next_observation = episode.observations[index + 1]
        return d3rlpy.dataset.Transition(
            observation=observation,
            action=episode.actions[index],
            reward=episode.rewards[index],
            next_observation=next_observation,
            terminal=float(is_terminal),
            interval=1,
        )
```

d3rlpy.dataset.TransitionPickerProtocol	Interface of TransitionPicker.
d3rlpy.dataset.BasicTransitionPicker	Standard transition picker.
d3rlpy.dataset.FrameStackTransitionPicker	Frame-stacking transition picker.
d3rlpy.dataset.MultiStepTransitionPicker	Multi-step transition picker.

d3rlpy.dataset.TransitionPickerProtocol

class d3rlpy.dataset.**TransitionPickerProtocol**(**args*, ***kwds*)

Interface of TransitionPicker.

Methods

__call__(*episode*, *index*)
Returns transition specified by `index`.

Parameters

- **episode** (*d3rlpy.dataset.components.EpisodeBase*) – Episode.
- **index** (*int*) – Index at the target transition.

Returns Transition.

Return type d3rlpy.dataset.components.Transition

d3rlpy.dataset.BasicTransitionPicker

class d3rlpy.dataset.**BasicTransitionPicker**(**args*, ***kwds*)
Standard transition picker.

This class implements a basic transition picking.

Methods

__call__(*episode*, *index*)
Returns transition specified by `index`.

Parameters

- **episode** (*d3rlpy.dataset.components.EpisodeBase*) – Episode.
- **index** (*int*) – Index at the target transition.

Returns Transition.

Return type d3rlpy.dataset.components.Transition

d3rlpy.dataset.FrameStackTransitionPicker

class d3rlpy.dataset.**FrameStackTransitionPicker**(**args*, ***kwds*)
Frame-stacking transition picker.

This class implements the frame-stacking logic. The observations are stacked with the last `n_frames-1` frames. When `index` specifies timestep below `n_frames`, those frames are padded by zeros.

```
episode = Episode(
    observations=np.random.random((100, 1, 84, 84)),
    actions=np.random.random((100, 2)),
    rewards=np.random.random((100, 1)),
    terminated=False,
)

frame_stacking_picker = FrameStackTransitionPicker(n_frames=4)
transition = frame_stacking_picker(episode, 10)

transition.observation.shape == (4, 84, 84)
```

Parameters **n_frames** – Number of frames to stack.

Methods

__call__(*episode*, *index*)
Returns transition specified by `index`.

Parameters

- **episode** (*d3rlpy.dataset.components.EpisodeBase*) – Episode.
- **index** (*int*) – Index at the target transition.

Returns Transition.

Return type d3rlpy.dataset.components.Transition

d3rlpy.dataset.MultiStepTransitionPicker

class d3rlpy.dataset.**MultiStepTransitionPicker**(**args*, ***kwds*)
Multi-step transition picker.

This class implements transition picking for the multi-step TD error. `reward` is computed as a multi-step discounted return.

Parameters

- **n_steps** – Delta timestep between `observation` and `net_observation`.
- **gamma** – Discount factor to compute a multi-step return.

Methods

__call__(*episode*, *index*)
Returns transition specified by `index`.

Parameters

- **episode** (*d3rlpy.dataset.components.EpisodeBase*) – Episode.
- **index** (*int*) – Index at the target transition.

Returns Transition.

Return type d3rlpy.dataset.components.Transition

4.3.5 TrajectorySlicer

`TrajectorySlicer` is a component that defines how to slice trajectory data used for Decision Transformer-based algorithms. You can also implement your own `TrajectorySlicer` for custom experiments.

```
import d3rlpy

class CustomTrajectorySlicer(d3rlpy.dataset.TrajectorySlicerProtocol):
    def __call__(
        self, episode: d3rlpy.dataset.EpisodeBase, end_index: int, size: int
    ) -> d3rlpy.dataset.PartialTrajectory:
        end = end_index + 1
        start = max(end - size, 0)
        actual_size = end - start
```

(continues on next page)

(continued from previous page)

```
        # prepare terminal flags
        terminals = np.zeros((actual_size, 1), dtype=np.float32)
        if episode.terminated and end_index == episode.size() - 1:
            terminals[-1][0] = 1.0

        # slice data
        observations = episode.observations[start:end]
        actions = episode.actions[start:end]
        rewards = episode.rewards[start:end]
        ret = np.sum(episode.rewards[start:])
        all_returns_to_go = ret - np.cumsum(episode.rewards[start:], axis=0)
        returns_to_go = all_returns_to_go[:actual_size].reshape((-1, 1))

        # prepare metadata
        timesteps = np.arange(start, end)
        masks = np.ones(end - start, dtype=np.float32)

        # compute backward padding size
        pad_size = size - actual_size

        if pad_size == 0:
            return d3rlpy.dataset.PartialTrajectory(
                observations=observations,
                actions=actions,
                rewards=rewards,
                returns_to_go=returns_to_go,
                terminals=terminals,
                timesteps=timesteps,
                masks=masks,
                length=size,
            )

        return d3rlpy.dataset.PartialTrajectory(
            observations=d3rlpy.dataset.batch_pad_observations(observations, pad_size),
            actions=d3rlpy.dataset.batch_pad_array(actions, pad_size),
            rewards=d3rlpy.dataset.batch_pad_array(rewards, pad_size),
            returns_to_go=d3rlpy.dataset.batch_pad_array(returns_to_go, pad_size),
            terminals=d3rlpy.dataset.batch_pad_array(terminals, pad_size),
            timesteps=d3rlpy.dataset.batch_pad_array(timesteps, pad_size),
            masks=d3rlpy.dataset.batch_pad_array(masks, pad_size),
            length=size,
        )
```

d3rlpy.dataset.TrajectorySlicerProtocol	Interface of TrajectorySlicer.
d3rlpy.dataset.BasicTrajectorySlicer	Standard trajectory slicer.

d3rlpy.dataset.TrajectorySlicerProtocol

class d3rlpy.dataset.**TrajectorySlicerProtocol**(**args*, ***kwds*)
Interface of TrajectorySlicer.

Methods

__call__(*episode*, *end_index*, *size*)
Slice trajectory.

This method returns a partial trajectory from `t=end_index-size` to `t=end_index`. If `end_index-size` is smaller than 0, those parts will be padded by zeros.

Parameters

- **episode** (*d3rlpy.dataset.components.EpisodeBase*) – Episode.
- **end_index** (*int*) – Index at the end of the sliced trajectory.
- **size** (*int*) – Length of the sliced trajectory.

Returns Sliced trajectory.

Return type d3rlpy.dataset.components.PartialTrajectory

d3rlpy.dataset.BasicTrajectorySlicer

class d3rlpy.dataset.**BasicTrajectorySlicer**(**args*, ***kwds*)
Standard trajectory slicer.

This class implements a basic trajectory slicing.

Methods

__call__(*episode*, *end_index*, *size*)
Slice trajectory.

This method returns a partial trajectory from `t=end_index-size` to `t=end_index`. If `end_index-size` is smaller than 0, those parts will be padded by zeros.

Parameters

- **episode** (*d3rlpy.dataset.components.EpisodeBase*) – Episode.
- **end_index** (*int*) – Index at the end of the sliced trajectory.
- **size** (*int*) – Length of the sliced trajectory.

Returns Sliced trajectory.

Return type d3rlpy.dataset.components.PartialTrajectory

4.3.6 WriterPreprocess

`WriterPreprocess` is a component that defines how to write experiences to an experience replay buffer. You can also implement your own `WriterPreprocess` for custom experiments.

```
import d3rlpy

class CustomWriterPreprocess(d3rlpy.dataset.WriterPreprocessProtocol):
    def process_observation(self, observation: d3rlpy.dataset.Observation) -> d3rlpy.
↪dataset.Observation:
        return observation

    def process_action(self, action: np.ndarray) -> np.ndarray:
        return action

    def process_reward(self, reward: np.ndarray) -> np.ndarray:
        return reward
```

d3rlpy.dataset.WriterPreprocessProtocol	Interface of WriterPreprocess.
d3rlpy.dataset.BasicWriterPreprocess	Stanard data writer.
d3rlpy.dataset.LastFrameWriterPreprocess	Data writer that writes the last channel of observation.

d3rlpy.dataset.WriterPreprocessProtocol

class d3rlpy.dataset.**WriterPreprocessProtocol**(**args*, ***kwds*)
Interface of WriterPreprocess.

Methods

process_action(*action*)
Processes action.

Parameters **action** (*numpy.ndarray*) – Action.

Returns Processed action.

Return type numpy.ndarray

process_observation(*observation*)
Processes observation.

Parameters **observation** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observation.

Returns Processed observation.

Return type Union[numpy.ndarray, Sequence[numpy.ndarray]]

process_reward(*reward*)
Processes reward.

Parameters **reward** (*numpy.ndarray*) – Reward.

Returns Processed reward.

Return type numpy.ndarray

d3rlpy.dataset.BasicWriterPreprocess

class d3rlpy.dataset.**BasicWriterPreprocess**(**args*, ***kwds*)
Stanard data writer.

This class implements identity preprocess.

Methods

process_action(*action*)
Processes action.

Parameters **action** (*numpy.ndarray*) – Action.

Returns Processed action.

Return type numpy.ndarray

process_observation(*observation*)
Processes observation.

Parameters **observation** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observation.

Returns Processed observation.

Return type Union[numpy.ndarray, Sequence[numpy.ndarray]]

process_reward(*reward*)
Processes reward.

Parameters **reward** (*numpy.ndarray*) – Reward.

Returns Processed reward.

Return type numpy.ndarray

d3rlpy.dataset.LastFrameWriterPreprocess

class d3rlpy.dataset.**LastFrameWriterPreprocess**(**args*, ***kwds*)
Data writer that writes the last channel of observation.

This class is designed to be used with `FrameStackTransitionPicker`.

Methods

process_action(*action*)
Processes action.

Parameters **action** (*numpy.ndarray*) – Action.

Returns Processed action.

Return type numpy.ndarray

process_observation(*observation*)
Processes observation.

Parameters **observation** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observation.

Returns Processed observation.

Return type Union[numpy.ndarray, Sequence[numpy.ndarray]]

process_reward(*reward*)
Processes reward.

Parameters **reward** (*numpy.ndarray*) – Reward.

Returns Processed reward.

Return type numpy.ndarray

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'*, *transition_picker=None*, *trajectory_slicer=None*)
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']`.
- **transition_picker** (*Optional[d3rlpy.dataset.transition_pickers.TransitionPickerProtocol]*) – TransitionPickerProtocol object.
- **trajectory_slicer** (*Optional[d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol]*) – TrajectorySlicerProtocol object.

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

Return type Tuple[*d3rlpy.dataset.replay_buffer.ReplayBuffer*, gym.core.Env[numpy.ndarray, int]]

4.4.2 d3rlpy.datasets.get_pendulum

d3rlpy.datasets.**get_pendulum**(*dataset_type='replay'*, *transition_picker=None*, *trajectory_slicer=None*)
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']`.
- **transition_picker** (*Optional[d3rlpy.dataset.transition_pickers.TransitionPickerProtocol]*) – TransitionPickerProtocol object.
- **trajectory_slicer** (*Optional[d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol]*) – TrajectorySlicerProtocol object.

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

Return type Tuple[*d3rlpy.dataset.replay_buffer.ReplayBuffer*, gym.core.Env[numpy.ndarray, numpy.ndarray]]

4.4.3 d3rlpy.datasets.get_atari

d3rlpy.datasets.**get_atari**(*env_name*, *num_stack=None*)
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.
- **num_stack** (*Optional[int]*) – the number of frames to stack (only applied to env).

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

Return type Tuple[*d3rlpy.dataset.replay_buffer.ReplayBuffer*, gym.core.Env[numpy.ndarray, int]]

4.4.4 d3rlpy.datasets.get_atari_transitions

d3rlpy.datasets.**get_atari_transitions**(*game_name*, *fraction=0.01*, *index=0*, *num_stack=None*)
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.
- **num_stack** (*Optional[int]*) – the number of frames to stack (only applied to env).

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

Return type Tuple[*d3rlpy.dataset.replay_buffer.ReplayBuffer*, gym.core.Env[numpy.ndarray, int]]

4.4.5 d3rlpy.datasets.get_d4rl

`d3rlpy.datasets.`**`get_d4rl`**(*env_name*, *transition_picker=None*, *trajectory_slicer=None*)

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.
- **transition_picker** (*Optional[d3rlpy.dataset.transition_pickers.TransitionPickerProtocol]*) – TransitionPickerProtocol object.
- **trajectory_slicer** (*Optional[d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol]*) – TrajectorySlicerProtocol object.

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

Return type Tuple[*d3rlpy.dataset.replay_buffer.ReplayBuffer*, gym.core.Env[numpy.ndarray, numpy.ndarray]]

4.4.6 d3rlpy.datasets.get_dataset

d3rlpy.datasets.**get_dataset**(*env_name*, *transition_picker=None*, *trajectory_slicer=None*)

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.
- **transition_picker** (*Optional[d3rlpy.dataset.transition_pickers.TransitionPickerProtocol]*) – TransitionPickerProtocol object.
- **trajectory_slicer** (*Optional[d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol]*) – TrajectorySlicerProtocol object.

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

Return type Tuple[*d3rlpy.dataset.replay_buffer.ReplayBuffer*, gym.core.Env[Any, Any]]

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.datasets import get_pendulum
from d3rlpy.algos import CQLConfig
from d3rlpy.preprocesing import StandardObservationScaler

dataset, _ = get_pendulum()

# choose from ['pixel', 'min_max', 'standard'] or None
cql = CQLConfig(observation_scaler=StandardObservationScaler()).create()

# observation scaler is fitted from the given dataset
cql.fit(dataset, n_steps=100000)

# 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 observation scalers by yourself.

```
from d3rlpy.preprocessing import StandardObservationScaler

observation_scaler = StandardObservationScaler(mean=..., std=...)

cql = CQLConfig(observation_scaler=observation_scaler).create()
```

d3rlpy.preprocessing.PixelObservationScaler	Pixel normalization preprocessing.
d3rlpy.preprocessing.MinMaxObservationScaler	Min-Max normalization preprocessing.
d3rlpy.preprocessing.StandardObservationScaler	Standardization preprocessing.

d3rlpy.preprocessing.PixelObservationScaler

class d3rlpy.preprocessing.**PixelObservationScaler**

Pixel normalization preprocessing.

$$x' = x/255$$

```
from d3rlpy.preprocessing import PixelObservationScaler
from d3rlpy.algos import CQLConfig

cql = CQLConfig(observation_scaler=PixelObservationScaler()).create()
```

Methods

Return type None

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

fit_with_env(*env*)
Gets scaling parameters from environment.

Parameters **env** (*gym.core.Env[Any, Any]*) – Gym environment.

Return type None

fit_with_trajectory_slicer(*episodes*, *trajectory_slicer*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **trajectory_slicer** (d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol) – Trajectory slicer to process mini-batch.

Return type None

fit_with_transition_picker(*episodes*, *transition_picker*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **transition_picker** (d3rlpy.dataset.transition_pickers.TransitionPickerProtocol) – Transition picker to process mini-batch.

Return type None

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

reverse_transform(*x*)
Returns reversely transformed output.

Parameters **x** (*torch.Tensor*) – input.

Returns Inversely transformed output.

Return type torch.Tensor

reverse_transform_numpy(*x*)
Returns reversely transformed output in numpy.

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

Returns Inversely transformed output.

Return type numpy.ndarray

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –

- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

transform(*x*)
Returns processed output.

Parameters **x** (*torch.Tensor*) – Input.

Returns Processed output.

Return type torch.Tensor

transform_numpy(*x*)
Returns processed output in numpy.

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

Returns Processed output.

Return type numpy.ndarray

Attributes

built

d3rlpy.preprocessing.MinMaxObservationScaler

class d3rlpy.preprocessing.**MinMaxObservationScaler**(*minimum=None*, *maximum=None*)
Min-Max normalization preprocessing.

Observations will be normalized in range `[-1.0, 1.0]`.

$$x' = (x - \min x)/(\max x - \min x) * 2 - 1$$

```
from d3rlpy.preprocessing import MinMaxObservationScaler
from d3rlpy.algos import CQLConfig

# normalize based on datasets or environments
cql = CQLConfig(observation_scaler=MinMaxObservationScaler()).create()

# manually initialize
minimum = observations.min(axis=0)
maximum = observations.max(axis=0)
observation_scaler = MinMaxObservationScaler(
    minimum=minimum,
    maximum=maximum,
```

(continues on next page)

(continued from previous page)

```
)
cql = CQLConfig(observation_scaler=observation_scaler).create()
```

Parameters

- **minimum** (*numpy.ndarray*) – Minimum values at each entry.
- **maximum** (*numpy.ndarray*) – Maximum values at each entry.

Return type None

Methods

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

fit_with_env(*env*)
Gets scaling parameters from environment.

Parameters **env** (*gym.core.Env[Any, Any]*) – Gym environment.

Return type None

fit_with_trajectory_slicer(*episodes*, *trajectory_slicer*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **trajectory_slicer** (d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol) – Trajectory slicer to process mini-batch.

Return type None

fit_with_transition_picker(*episodes*, *transition_picker*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.

- **transition_picker** (d3rlpy.dataset.transition_pickers.TransitionPickerProtocol) – Transition picker to process mini-batch.

Return type None

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

reverse_transform(*x*)

Returns reversely transformed output.

Parameters **x** (*torch.Tensor*) – input.

Returns Inversely transformed output.

Return type torch.Tensor

reverse_transform_numpy(*x*)

Returns reversely transformed output in numpy.

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

Returns Inversely transformed output.

Return type numpy.ndarray

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

transform(*x*)

Returns processed output.

Parameters **x** (*torch.Tensor*) – Input.

Returns Processed output.

Return type torch.Tensor

transform_numpy(*x*)

Returns processed output in numpy.

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

Returns Processed output.

Return type numpy.ndarray

Attributes

built

maximum: Optional[numpy.ndarray] = None

minimum: Optional[numpy.ndarray] = None

d3rlpy.preprocessing.StandardObservationScaler

class d3rlpy.preprocessing.**StandardObservationScaler**(*mean=None*, *std=None*, *eps=0.001*)
Standardization preprocessing.

$$x' = (x - \mu)/\sigma$$

```
from d3rlpy.preprocessing import StandardObservationScaler
from d3rlpy.algos import CQLConfig

# normalize based on datasets
cql = CQLConfig(observation_scaler=StandardObservationScaler()).create()

# manually initialize
mean = observations.mean(axis=0)
std = observations.std(axis=0)
observation_scaler = StandardObservationScaler(mean=mean, std=std)
cql = CQLConfig(observation_scaler=observation_scaler).create()
```

Parameters

- **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.

Return type None

Methods

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

fit_with_env(*env*)
Gets scaling parameters from environment.

Parameters **env** (*gym.core.Env[Any, Any]*) – Gym environment.

Return type None

fit_with_trajectory_slicer(*episodes*, *trajectory_slicer*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **trajectory_slicer** (d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol) – Trajectory slicer to process mini-batch.

Return type None

fit_with_transition_picker(*episodes*, *transition_picker*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **transition_picker** (d3rlpy.dataset.transition_pickers.TransitionPickerProtocol) – Transition picker to process mini-batch.

Return type None

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

reverse_transform(*x*)
Returns reversely transformed output.

Parameters **x** (*torch.Tensor*) – input.

Returns Inversely transformed output.

Return type torch.Tensor

reverse_transform_numpy(*x*)
Returns reversely transformed output in numpy.

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

Returns Inversely transformed output.

Return type numpy.ndarray

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, indent=None, separators=None, default=None, sort_keys=False, **kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

transform(*x*)
Returns processed output.

Parameters **x** (*torch.Tensor*) – Input.

Returns Processed output.

Return type torch.Tensor

transform_numpy(*x*)
Returns processed output in numpy.

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

Returns Processed output.

Return type numpy.ndarray

Attributes

built

eps: float = 0.001

mean: Optional[numpy.ndarray] = None

std: Optional[numpy.ndarray] = None

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.datasets import get_pendulum
from d3rlpy.algos import CQLConfig
from d3rlpy.preprocessing import MinMaxActionScaler

dataset, _ = get_pendulum()

cql = CQLConfig(action_scaler=MinMaxActionScaler()).create()

# action scaler is fitted from the given episodes
cql.fit(dataset, n_steps=100000)

# 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 = CQLConfig(action_scaler=action_scaler).create()
```

d3rlpy.preprocessing.MinMaxActionScaler	Min-Max normalization action preprocessing.

d3rlpy.preprocessing.MinMaxActionScaler

class d3rlpy.preprocessing.**MinMaxActionScaler**(*minimum=None*, *maximum=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.preprocessing import MinMaxActionScaler
from d3rlpy.algos import CQLConfig

# normalize based on datasets or environments
cql = CQLConfig(action_scaler=MinMaxActionScaler()).create()

# manually initialize
minimum = actions.min(axis=0)
maximum = actions.max(axis=0)
action_scaler = MinMaxActionScaler(minimum=minimum, maximum=maximum)
cql = CQLConfig(action_scaler=action_scaler).create()
```

Parameters

- **minimum** (*numpy.ndarray*) – Minimum values at each entry.
- **maximum** (*numpy.ndarray*) – Maximum values at each entry.

Return type None

Methods

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

fit_with_env(*env*)
Gets scaling parameters from environment.

Parameters **env** (*gym.core.Env[Any, Any]*) – Gym environment.

Return type None

fit_with_trajectory_slicer(*episodes*, *trajectory_slicer*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **trajectory_slicer** (d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol) – Trajectory slicer to process mini-batch.

Return type None

fit_with_transition_picker(*episodes*, *transition_picker*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **transition_picker** (d3rlpy.dataset.transition_pickers.TransitionPickerProtocol) – Transition picker to process mini-batch.

Return type None

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

reverse_transform(*x*)
Returns reversely transformed output.

Parameters **x** (*torch.Tensor*) – input.

Returns Inversely transformed output.

Return type torch.Tensor

reverse_transform_numpy(*x*)
Returns reversely transformed output in numpy.

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

Returns Inversely transformed output.

Return type numpy.ndarray

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

transform(*x*)

Returns processed output.

Parameters **x** (*torch.Tensor*) – Input.

Returns Processed output.

Return type torch.Tensor

transform_numpy(*x*)

Returns processed output in numpy.

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

Returns Processed output.

Return type numpy.ndarray

Attributes

built

maximum: Optional[numpy.ndarray] = None

minimum: Optional[numpy.ndarray] = None

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.datasets import get_pendulum
from d3rlpy.algos import CQLConfig
from d3rlpy.preprocessing import StandardRewardScaler

dataset, _ = get_pendulum()

cql = CQLConfig(reward_scaler=StandardRewardScaler()).create()

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

# reward scaler is also available at finetuning.
cql.fit_online(env)
```

You can also initialize scalers by yourself.

```
from d3rlpy.preprocessing import MinMaxRewardScaler

reward_scaler = MinMaxRewardScaler(minimum=..., maximum=...)

cql = CQLConfig(reward_scaler=reward_scaler).create()
```

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.ConstantShiftRewardScaler	Reward shift preprocessing.

d3rlpy.preprocessing.MinMaxRewardScaler

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

Min-Max reward normalization preprocessing.

Rewards will be normalized in range `[0.0, 1.0]`.

$$r' = (r - \min(r))/(\max(r) - \min(r))$$

```
from d3rlpy.preprocessing import MinMaxRewardScaler
from d3rlpy.algos import CQLConfig

# normalize based on datasets
cql = CQLConfig(reward_scaler=MinMaxRewardScaler()).create()

# initialize manually
```

(continues on next page)

(continued from previous page)

```
reward_scaler = MinMaxRewardScaler(minimum=0.0, maximum=10.0)
cql = CQLConfig(reward_scaler=reward_scaler).create()
```

Parameters

- **minimum** (*float*) – Minimum value.
- **maximum** (*float*) – Maximum value.
- **multiplier** (*float*) – Constant multiplication value.

Return type None

Methods

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

fit_with_env(*env*)
Gets scaling parameters from environment.

Parameters **env** (*gym.core.Env[Any, Any]*) – Gym environment.

Return type None

fit_with_trajectory_slicer(*episodes*, *trajectory_slicer*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **trajectory_slicer** (d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol) – Trajectory slicer to process mini-batch.

Return type None

fit_with_transition_picker(*episodes*, *transition_picker*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.

- **transition_picker** (d3rlpy.dataset.transition_pickers.TransitionPickerProtocol) – Transition picker to process mini-batch.

Return type None

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

reverse_transform(*x*)

Returns reversely transformed output.

Parameters **x** (*torch.Tensor*) – input.

Returns Inversely transformed output.

Return type torch.Tensor

reverse_transform_numpy(*x*)

Returns reversely transformed output in numpy.

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

Returns Inversely transformed output.

Return type numpy.ndarray

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(*, skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

transform(*x*)

Returns processed output.

Parameters **x** (*torch.Tensor*) – Input.

Returns Processed output.

Return type torch.Tensor

transform_numpy(*x*)

Returns processed output in numpy.

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

Returns Processed output.

Return type numpy.ndarray

Attributes

built

maximum: Optional[float] = None

minimum: Optional[float] = None

multiplier: float = 1.0

d3rlpy.preprocessing.StandardRewardScaler

class d3rlpy.preprocessing.**StandardRewardScaler**(*mean=None*, *std=None*, *eps=0.001*, *multiplier=1.0*)
Reward standardization preprocessing.

$$r' = (r - \mu)/\sigma$$

```
from d3rlpy.preprocessing import StandardRewardScaler
from d3rlpy.algos import CQLConfig

# normalize based on datasets
cql = CQLConfig(reward_scaler=StandardRewardScaler()).create()

# initialize manually
reward_scaler = StandardRewardScaler(mean=0.0, std=1.0)
cql = CQLConfig(reward_scaler=reward_scaler).create()
```

Parameters

- **mean** (*float*) – Mean value.
- **std** (*float*) – Standard deviation value.
- **eps** (*float*) – Constant value to avoid zero-division.
- **multiplier** (*float*) – Constant multiplication value

Return type None

Methods

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

fit_with_env(*env*)
Gets scaling parameters from environment.

Parameters **env** (*gym.core.Env[Any, Any]*) – Gym environment.

Return type None

fit_with_trajectory_slicer(*episodes*, *trajectory_slicer*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **trajectory_slicer** (d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol) – Trajectory slicer to process mini-batch.

Return type None

fit_with_transition_picker(*episodes*, *transition_picker*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **transition_picker** (d3rlpy.dataset.transition_pickers.TransitionPickerProtocol) – Transition picker to process mini-batch.

Return type None

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

reverse_transform(*x*)
Returns reversely transformed output.

Parameters **x** (*torch.Tensor*) – input.

Returns Inversely transformed output.

Return type torch.Tensor

reverse_transform_numpy(*x*)
Returns reversely transformed output in numpy.

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

Returns Inversely transformed output.

Return type numpy.ndarray

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

transform(*x*)
Returns processed output.

Parameters **x** (*torch.Tensor*) – Input.

Returns Processed output.

Return type torch.Tensor

transform_numpy(*x*)
Returns processed output in numpy.

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

Returns Processed output.

Return type numpy.ndarray

Attributes

built

eps: *float* = **0.001**

mean: *Optional[float]* = **None**

multiplier: *float* = **1.0**

std: *Optional[float]* = **None**

d3rlpy.preprocessing.ClipRewardScaler

class d3rlpy.preprocessing.**ClipRewardScaler**(*low=None*, *high=None*, *multiplier=1.0*)

Reward clipping preprocessing.

```
from d3rlpy.preprocessing import ClipRewardScaler
from d3rlpy.algos import CQLConfig

# clip rewards within [-1.0, 1.0]
reward_scaler = ClipRewardScaler(low=-1.0, high=1.0)
cql = CQLConfig(reward_scaler=reward_scaler).create()
```

Parameters

- **low** (*Optional[float]*) – Minimum value to clip.
- **high** (*Optional[float]*) – Maximum value to clip.
- **multiplier** (*float*) – Constant multiplication value.

Return type None

Methods

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

fit_with_env(*env*)

Gets scaling parameters from environment.

Parameters **env** (*gym.core.Env[Any, Any]*) – Gym environment.

Return type None

fit_with_trajectory_slicer(*episodes*, *trajectory_slicer*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **trajectory_slicer** (d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol) – Trajectory slicer to process mini-batch.

Return type None

fit_with_transition_picker(*episodes*, *transition_picker*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **transition_picker** (d3rlpy.dataset.transition_pickers.TransitionPickerProtocol) – Transition picker to process mini-batch.

Return type None

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

reverse_transform(*x*)
Returns reversely transformed output.

Parameters **x** (*torch.Tensor*) – input.

Returns Inversely transformed output.

Return type torch.Tensor

reverse_transform_numpy(*x*)
Returns reversely transformed output in numpy.

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

Returns Inversely transformed output.

Return type numpy.ndarray

classmethod schema(*, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

transform(*x*)
Returns processed output.

Parameters **x** (*torch.Tensor*) – Input.

Returns Processed output.

Return type torch.Tensor

transform_numpy(*x*)
Returns processed output in numpy.

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

Returns Processed output.

Return type numpy.ndarray

Attributes

built

high: Optional[float] = None

low: Optional[float] = None

multiplier: float = 1.0

d3rlpy.preprocessing.MultiplyRewardScaler

class d3rlpy.preprocessing.**MultiplyRewardScaler**(*multiplier=1.0*)

Multiplication reward preprocessing.

This preprocessor multiplies rewards by a constant number.

```
from d3rlpy.preprocessing import MultiplyRewardScaler
from d3rlpy.algos import CQLConfig

# multiply rewards by 10
reward_scaler = MultiplyRewardScaler(10.0)
cql = CQLConfig(reward_scaler=reward_scaler).create()
```

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

Return type None

Methods

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

fit_with_env(*env*)

Gets scaling parameters from environment.

Parameters **env** (*gym.core.Env[Any, Any]*) – Gym environment.

Return type None

fit_with_trajectory_slicer(*episodes*, *trajectory_slicer*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **trajectory_slicer** (d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol) – Trajectory slicer to process mini-batch.

Return type None

fit_with_transition_picker(*episodes*, *transition_picker*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **transition_picker** (d3rlpy.dataset.transition_pickers.TransitionPickerProtocol) – Transition picker to process mini-batch.

Return type None

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

reverse_transform(*x*)
Returns reversely transformed output.

Parameters **x** (*torch.Tensor*) – input.

Returns Inversely transformed output.

Return type torch.Tensor

reverse_transform_numpy(*x*)
Returns reversely transformed output in numpy.

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

Returns Inversely transformed output.

Return type numpy.ndarray

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

transform(*x*)

Returns processed output.

Parameters **x** (*torch.Tensor*) – Input.

Returns Processed output.

Return type torch.Tensor

transform_numpy(*x*)

Returns processed output in numpy.

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

Returns Processed output.

Return type numpy.ndarray

Attributes

built

multiplier: *float* = **1.0**

d3rlpy.preprocessing.ReturnBasedRewardScaler

class d3rlpy.preprocessing.**ReturnBasedRewardScaler**(*return_max=None*, *return_min=None*, *multiplier=1.0*)

Reward normalization preprocessing based on return scale.

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

```
from d3rlpy.preprocessing import ReturnBasedRewardScaler
from d3rlpy.algos import CQLConfig

# normalize based on datasets
cql = CQLConfig(reward_scaler=ReturnBasedRewardScaler()).create()

# initialize manually
reward_scaler = ReturnBasedRewardScaler(
    return_max=100.0,
    return_min=1.0,
)
cql = CQLConfig(reward_scaler=reward_scaler).create()
```

References

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

Parameters

- **return_max** (*float*) – Maximum return value.
- **return_min** (*float*) – Standard deviation value.
- **multiplier** (*float*) – Constant multiplication value

Return type None

Methods

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

fit_with_env(*env*)
Gets scaling parameters from environment.

Parameters **env** (*gym.core.Env[Any, Any]*) – Gym environment.

Return type None

fit_with_trajectory_slicer(*episodes*, *trajectory_slicer*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **trajectory_slicer** (d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol) – Trajectory slicer to process mini-batch.

Return type None

fit_with_transition_picker(*episodes*, *transition_picker*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **transition_picker** (d3rlpy.dataset.transition_pickers.TransitionPickerProtocol) – Transition picker to process mini-batch.

Return type None

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

reverse_transform(*x*)
Returns reversely transformed output.

Parameters **x** (*torch.Tensor*) – input.

Returns Inversely transformed output.

Return type torch.Tensor

reverse_transform_numpy(*x*)
Returns reversely transformed output in numpy.

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

Returns Inversely transformed output.

Return type numpy.ndarray

classmethod schema(*, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

transform(*x*)
Returns processed output.

Parameters **x** (*torch.Tensor*) – Input.

Returns Processed output.

Return type torch.Tensor

transform_numpy(*x*)

Returns processed output in numpy.

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

Returns Processed output.

Return type numpy.ndarray

Attributes

built

multiplier: float = 1.0

return_max: Optional[float] = None

return_min: Optional[float] = None

d3rlpy.preprocessing.ConstantShiftRewardScaler

class d3rlpy.preprocessing.**ConstantShiftRewardScaler**(*shift*)

Reward shift preprocessing.

$$r' = r + c$$

You need to initialize manually.

```
from d3rlpy.preprocessing import ConstantShiftRewardScaler
from d3rlpy.algos import CQLConfig

reward_scaler = ConstantShiftRewardScaler(shift=-1.0)
cql = CQLConfig(reward_scaler=reward_scaler).create()
```

References

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

Parameters **shift** (*float*) – Constant shift value

Return type None

Methods

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

fit_with_env(*env*)
Gets scaling parameters from environment.

Parameters **env** (*gym.core.Env[Any, Any]*) – Gym environment.

Return type None

fit_with_trajectory_slicer(*episodes*, *trajectory_slicer*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **trajectory_slicer** (d3rlpy.dataset.trajectory_slicers.TrajectorySlicerProtocol) – Trajectory slicer to process mini-batch.

Return type None

fit_with_transition_picker(*episodes*, *transition_picker*)
Estimates scaling parameters from dataset.

Parameters

- **episodes** (*Sequence[d3rlpy.dataset.components.EpisodeBase]*) – List of episodes.
- **transition_picker** (d3rlpy.dataset.transition_pickers.TransitionPickerProtocol) – Transition picker to process mini-batch.

Return type None

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

reverse_transform(*x*)
Returns reversely transformed output.

Parameters **x** (*torch.Tensor*) – input.

Returns Inversely transformed output.

Return type torch.Tensor

reverse_transform_numpy(*x*)
Returns reversely transformed output in numpy.

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

Returns Inversely transformed output.

Return type numpy.ndarray

classmethod schema(**, infer_missing=False, only=None, exclude=(), many=False, context=None, load_only=(), dump_only=(), partial=False, unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, indent=None, separators=None, default=None, sort_keys=False, **kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

transform(*x*)
Returns processed output.

Parameters **x** (*torch.Tensor*) – Input.

Returns Processed output.

Return type torch.Tensor

transform_numpy(*x*)
Returns processed output in numpy.

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

Returns Processed output.

Return type numpy.ndarray

Attributes

built

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.

```
import d3rlpy
from torch.optim import Adam

# modify weight decay
optim_factory = d3rlpy.models.OptimizerFactory(Adam, weight_decay=1e-4)

# set OptimizerFactory
dqn = d3rlpy.algos.DQNConfig(optim_factory=optim_factory).create()
```

There are also convenient alises.

```
# alias for Adam optimizer
optim_factory = d3rlpy.models.AdamFactory(weight_decay=1e-4)

dqn = d3rlpy.algos.DQNConfig(optim_factory=optim_factory).create()
```

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

4.6.1 d3rlpy.models.OptimizerFactory

class d3rlpy.models.**OptimizerFactory**

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

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

Methods

Return type None

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

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

classmethod schema(*, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

4.6.2 d3rlpy.models.SGDFactory

class d3rlpy.models.**SGDFactory**(*momentum=0.0*, *dampening=0.0*, *weight_decay=0.0*, *nesterov=False*)

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.

Return type None

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

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

classmethod schema(*, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

Attributes

dampening: float = 0.0

momentum: float = 0.0

nesterov: bool = False

weight_decay: float = 0.0

4.6.3 d3rlpy.models.AdamFactory

class d3rlpy.models.**AdamFactory**(*betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*)

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.

Return type None

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

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

Attributes

amsgrad: bool = False

betas: Tuple[float, float] = (0.9, 0.999)

eps: float = 1e-08

weight_decay: float = 0

4.6.4 d3rlpy.models.RMSpropFactory

class d3rlpy.models.**RMSpropFactory**(*alpha=0.95*, *eps=0.01*, *weight_decay=0.0*, *momentum=0.0*, *centered=True*)

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.

Return type None

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

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –

- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

Attributes

alpha: float = 0.95

centered: bool = True

eps: float = 0.01

momentum: float = 0.0

weight_decay: float = 0.0

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.

```
import d3rlpy

# encoder factory
encoder_factory = d3rlpy.models.VectorEncoderFactory(
    hidden_units=[300, 400],
    activation='tanh',
)

# set EncoderFactory
dqn = d3rlpy.algos.DQNConfig(encoder_factory=encoder_factory).create()
```

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)
```

(continues on next page)

(continued from previous page)

```
        self.fc2 = nn.Linear(64, feature_size)

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

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

# your own encoder factory
class CustomEncoderFactory(EncoderFactory):
    TYPE = 'custom' # this is necessary

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

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

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

dqn = d3rlpy.algos.DQNConfig(
   encoder_factory=CustomEncoderFactory(feature_size=64),
).create()
```

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
```

(continues on next page)

(continued from previous page)

```
    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}

factory = CustomEncoderFactory(feature_size=64)

sac = d3rlpy.algos.SACConfig(
   actor_encoder_factory=factory,
   critic_encoder_factory=factory,
).create()
```

If you want `load_learnable` 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 d3
dqn = d3rlpy.load_learnable("model.d3")
```

d3rlpy.models.DefaultEncoderFactory	Default encoder factory class.
d3rlpy.models.PixelEncoderFactory	Pixel encoder factory class.
d3rlpy.models.VectorEncoderFactory	Vector encoder factory class.
d3rlpy.models.DenseEncoderFactory	DenseNet encoder factory class.

4.7.1 d3rlpy.models.DefaultEncoderFactory

class d3rlpy.models.**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.

Return type None

Methods

create(*observation_shape*)
Returns PyTorch's state enocder module.

Parameters **observation_shape** (*Union[Sequence[int], Sequence[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** (*Union[Sequence[int], Sequence[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

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

classmethod schema(*, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

Attributes

activation: `str = 'relu'`

dropout_rate: `Optional[float] = None`

use_batch_norm: `bool = False`

4.7.2 d3rlpy.models.PixelEncoderFactory

class d3rlpy.models.**PixelEncoderFactory**(*filters=<factory>*, *feature_size=512*, *activation='relu'*, *use_batch_norm=False*, *dropout_rate=None*, *exclude_last_activation=False*)

Pixel encoder factory class.

This is the default encoder factory for image observation.

Parameters

- **filters** (*list*) – List of tuples consisting with `(filter_size, kernel_size, stride)`. If None, `Nature DQN`-based architecture is used.
- **feature_size** (*int*) – Last linear layer size.
- **activation** (*str*) – Activation function name.
- **use_batch_norm** (*bool*) – Flag to insert batch normalization layers.
- **dropout_rate** (*float*) – Dropout probability.
- **exclude_last_activation** (*bool*) – Flag to exclude activation function at the last layer.

Return type None

Methods

create(*observation_shape*)

Returns PyTorch's state enocder module.

Parameters **observation_shape** (*Union[Sequence[int], Sequence[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** (*Union[Sequence[int], Sequence[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

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

Attributes

activation: `str = 'relu'`

dropout_rate: `Optional[float] = None`

exclude_last_activation: `bool = False`

feature_size: `int = 512`

use_batch_norm: `bool = False`

4.7.3 d3rlpy.models.VectorEncoderFactory

class d3rlpy.models.**VectorEncoderFactory**(*hidden_units=<factory>*, *activation='relu'*, *use_batch_norm=False*, *dropout_rate=None*, *use_dense=False*, *exclude_last_activation=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.
- **exclude_last_activation** (*bool*) – Flag to exclude activation function at the last layer.

Return type None

Methods

create(*observation_shape*)
Returns PyTorch's state enocder module.

Parameters **observation_shape** (*Union[Sequence[int], Sequence[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** (*Union[Sequence[int], Sequence[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

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

classmethod schema(*, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

Attributes

activation: *str* = 'relu'

dropout_rate: *Optional[float]* = None

exclude_last_activation: *bool* = False

use_batch_norm: *bool* = False

use_dense: *bool* = False

4.7.4 d3rlpy.models.DenseEncoderFactory

class d3rlpy.models.**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.

Return type None

Methods

create(*observation_shape*)

Returns PyTorch's state enocder module.

Parameters **observation_shape** (*Union[Sequence[int], Sequence[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** (*Union[Sequence[int], Sequence[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

classmethod deserialize(*serialized_config*)

Parameters **serialized_config** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_dict(*dict_config*)

Parameters **dict_config** (*Dict[str, Any]*) –

Return type d3rlpy.serializable_config.TConfig

classmethod deserialize_from_file(*path*)

Parameters **path** (*str*) –

Return type d3rlpy.serializable_config.TConfig

classmethod from_dict(*kvs*, ***, *infer_missing=False*)

Parameters **kvs** (*Optional[Union[dict, list, str, int, float, bool]]*) –

Return type dataclasses_json.api.A

classmethod from_json(*s*, ***, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *infer_missing=False*, ***kw*)

Parameters **s** (*Union[str, bytes, bytearray]*) –

Return type dataclasses_json.api.A

static get_type()

Return type str

classmethod schema(***, *infer_missing=False*, *only=None*, *exclude=()*, *many=False*, *context=None*, *load_only=()*, *dump_only=()*, *partial=False*, *unknown=None*)

Parameters

- **infer_missing** (*bool*) –
- **many** (*bool*) –
- **partial** (*bool*) –

Return type dataclasses_json.mm.SchemaF[dataclasses_json.api.A]

serialize()

Return type str

serialize_to_dict()

Return type Dict[str, Any]

to_dict(*encode_json=False*)

Return type Dict[str, Optional[Union[dict, list, str, int, float, bool]]]

to_json(*, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, indent=None, separators=None, default=None, sort_keys=False, **kw*)

Parameters

- **skipkeys** (*bool*) –
- **ensure_ascii** (*bool*) –
- **check_circular** (*bool*) –
- **allow_nan** (*bool*) –
- **indent** (*Optional[Union[int, str]]*) –
- **separators** (*Optional[Tuple[str, str]]*) –
- **default** (*Optional[Callable]*) –
- **sort_keys** (*bool*) –

Return type str

Attributes

activation: str = 'relu'

dropout_rate: Optional[float] = None

use_batch_norm: bool = False

4.8 Metrics

d3rlpy provides scoring functions for offline Q-learning-based training. You can also check *Logging* to understand how to write metrics to files.

```
import d3rlpy

dataset, env = d3rlpy.datasets.get_cartpole()
# use partial episodes as test data
test_episodes = dataset.episodes[:10]

dqn = d3rlpy.algos.DQNConfig().create()

dqn.fit(
    dataset,
    n_steps=100000,
    evaluators={
```

(continues on next page)

(continued from previous page)

```
        'td_error': d3rlpy.metrics.TDErrorEvaluator(test_episodes),
        'value_scale': d3rlpy.metrics.AverageValueEstimationEvaluator(test_episodes),
        'environment': d3rlpy.metrics.EnvironmentEvaluator(env),
    },
)
```

You can also implement your own metrics.

```
class CustomEvaluator(d3rlpy.metrics.EvaluatorProtocol):
    def __call__(self, algo: d3rlpy.algos.QLearningAlgoBase, dataset: ReplayBuffer) ->␣
↪float:
        # do some evaluation
```

d3rlpy.metrics.TDErrorEvaluator	Returns average TD error.
d3rlpy.metrics.DiscountedSumOfAdvantageEvaluator	Returns average of discounted sum of advantage.
d3rlpy.metrics.AverageValueEstimationEvaluator	Returns average value estimation.
d3rlpy.metrics.InitialStateValueEstimationEvaluator	Returns mean estimated action-values at the initial states.
d3rlpy.metrics.SoftOPCEvaluator	Returns Soft Off-Policy Classification metrics.
d3rlpy.metrics.ContinuousActionDiffEvaluator	Returns squared difference of actions between algorithm and dataset.
d3rlpy.metrics.DiscreteActionMatchEvaluator	Returns percentage of identical actions between algorithm and dataset.
d3rlpy.metrics.EnvironmentEvaluator	Action matches between algorithms.
d3rlpy.metrics.CompareContinuousActionDiffEvaluator	Action difference between algorithms.
d3rlpy.metrics.CompareDiscreteActionMatchEvaluator	Action matches between algorithms.

4.8.1 d3rlpy.metrics.TDErrorEvaluator

class d3rlpy.metrics.**TDErrorEvaluator**(**args*, ***kwds*)

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 **episodes** – Optional evaluation episodes. If it's not given, dataset used in training will be used.

Methods

__call__(*algo*, *dataset*)

Computes metrics.

Parameters

- **algo** (*d3rlpy.interface.QLearningAlgoProtocol*) – Q-learning algorithm.
- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – ReplayBuffer.

Returns Computed metrics.

Return type float

4.8.2 d3rlpy.metrics.DiscountedSumOfAdvantageEvaluator

class d3rlpy.metrics.**DiscountedSumOfAdvantageEvaluator**(**args*, ***kwds*)

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 **episodes** – Optional evaluation episodes. If it's not given, dataset used in training will be used.

Methods

__call__(*algo*, *dataset*)

Computes metrics.

Parameters

- **algo** (*d3rlpy.interface.QLearningAlgoProtocol*) – Q-learning algorithm.
- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – ReplayBuffer.

Returns Computed metrics.

Return type float

4.8.3 d3rlpy.metrics.AverageValueEstimationEvaluator

class d3rlpy.metrics.**AverageValueEstimationEvaluator**(**args*, ***kwds*)

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 **episodes** – Optional evaluation episodes. If it's not given, dataset used in training will be used.

Methods

__call__(*algo*, *dataset*)
Computes metrics.

Parameters

- **algo** (*d3rlpy.interface.QLearningAlgoProtocol*) – Q-learning algorithm.
- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – ReplayBuffer.

Returns Computed metrics.

Return type float

4.8.4 d3rlpy.metrics.InitialStateValueEstimationEvaluator

class d3rlpy.metrics.**InitialStateValueEstimationEvaluator**(**args*, ***kwds*)
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 **episodes** – Optional evaluation episodes. If it's not given, dataset used in training will be used.

Methods

__call__(*algo*, *dataset*)
Computes metrics.

Parameters

- **algo** (*d3rlpy.interface.QLearningAlgoProtocol*) – Q-learning algorithm.
- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – ReplayBuffer.

Returns Computed metrics.

Return type float

4.8.5 d3rlpy.metrics.SoftOPCEvaluator

class d3rlpy.metrics.**SoftOPCEvaluator**(**args*, ***kwds*)
Returns Soft Off-Policy Classification metrics.

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.

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

References

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

Parameters

- **return_threshold** – Return threshold of success episodes.
- **episodes** – Optional evaluation episodes. If it's not given, dataset used in training will be used.

Methods

__call__(*algo*, *dataset*)
Computes metrics.

Parameters

- **algo** (*d3rlpy.interface.QLearningAlgoProtocol*) – Q-learning algorithm.
- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – ReplayBuffer.

Returns Computed metrics.

Return type float

4.8.6 d3rlpy.metrics.ContinuousActionDiffEvaluator

class d3rlpy.metrics.**ContinuousActionDiffEvaluator**(**args*, ***kwds*)
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 **episodes** – Optional evaluation episodes. If it's not given, dataset used in training will be used.

Methods

__call__(*algo*, *dataset*)
Computes metrics.

Parameters

- **algo** (*d3rlpy.interface.QLearningAlgoProtocol*) – Q-learning algorithm.
- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – ReplayBuffer.

Returns Computed metrics.

Return type float

4.8.7 d3rlpy.metrics.DiscreteActionMatchEvaluator

class d3rlpy.metrics.**DiscreteActionMatchEvaluator**(**args*, ***kwds*)
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 **episodes** – Optional evaluation episodes. If it's not given, dataset used in training will be used.

Methods

__call__(*algo*, *dataset*)
Computes metrics.

Parameters

- **algo** (*d3rlpy.interface.QLearningAlgoProtocol*) – Q-learning algorithm.
- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – ReplayBuffer.

Returns Computed metrics.

Return type float

4.8.8 d3rlpy.metrics.EnvironmentEvaluator

class d3rlpy.metrics.**EnvironmentEvaluator**(**args*, ***kwds*)
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.

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

Parameters

- **env** – Gym environment.
- **n_trials** – Number of episodes to evaluate.

- **epsilon** – Probability of random action.
- **render** – Flag to turn on rendering.

Methods

__call__(*algo*, *dataset*)
Computes metrics.

Parameters

- **algo** (*d3rlpy.interface.QLearningAlgoProtocol*) – Q-learning algorithm.
- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – ReplayBuffer.

Returns Computed metrics.

Return type float

4.8.9 d3rlpy.metrics.CompareContinuousActionDiffEvaluator

class d3rlpy.metrics.**CompareContinuousActionDiffEvaluator**(**args*, ***kwds*)
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.

$$\mathbb{E}_{s_t \sim D}[(\pi_{\phi_1}(s_t) - \pi_{\phi_2}(s_t))^2]$$

Parameters

- **base_algo** – Target algorithm to comapre with.
- **episodes** – Optional evaluation episodes. If it's not given, dataset used in training will be used.

Methods

__call__(*algo*, *dataset*)
Computes metrics.

Parameters

- **algo** (*d3rlpy.interface.QLearningAlgoProtocol*) – Q-learning algorithm.
- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – ReplayBuffer.

Returns Computed metrics.

Return type float

4.8.10 d3rlpy.metrics.CompareDiscreteActionMatchEvaluator

class d3rlpy.metrics.**CompareDiscreteActionMatchEvaluator**(**args*, ***kwds*)

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.

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

Parameters

- **base_algo** – Target algorithm to comapre with.
- **episodes** – Optional evaluation episodes. If it's not given, dataset used in training will be used.

Methods

__call__(*algo*, *dataset*)

Computes metrics.

Parameters

- **algo** (*d3rlpy.interface.QLearningAlgoProtocol*) – Q-learning algorithm.
- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – ReplayBuffer.

Returns Computed metrics.

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.

```
import d3rlpy

# prepare the trained algorithm
cql = d3rlpy.load_learnable("model.d3")

# dataset to evaluate with
dataset, env = d3rlpy.datasets.get_pendulum()

# off-policy evaluation algorithm
fqe = d3rlpy.ope.FQE(algo=cql, config=d3rlpy.ope.FQEConfig())

# train estimators to evaluate the trained policy
fqe.fit(
   dataset,
   n_steps=100000,
   scorers={
      'init_value': d3rlpy.metrics.InitialStateValueEstimationEvaluator(),
      'soft_opc': d3rlpy.metrics.SoftOPCEvaluator(return_threshold=-300),
   },
)
```

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*, *config*, *device=False*, *impl=None*)

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.
- **config** (*d3rlpy.ope.FQEConfig*) – FQE config.
- **device** (*bool, int or str*) – Flag to use GPU, device ID or PyTorch device identifier.
- **impl** (*d3rlpy.metrics.ope.torch.FQEImpl*) – Algorithm implementation.

Methods

build_with_dataset(*dataset*)

Instantiate implementation object with ReplayBuffer object.

Parameters **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – dataset.

Return type None

build_with_env(*env*)

Instantiate implementation object with OpenAI Gym object.

Parameters **env** (*gym.core.Env[Any, Any]*) – gym-like environment.

Return type None

collect(*env*, *buffer=None*, *explorer=None*, *deterministic=False*, *n_steps=1000000*, *show_progress=True*)

Collects data via interaction with environment.

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

Parameters

- **env** (*gym.core.Env[Any, Any]*) – Fym-like environment.

- **buffer** (*Optional[d3rlpy.dataset.replay_buffer.ReplayBuffer]*) – Replay buffer.
- **explorer** (*Optional[d3rlpy.algos.qlearning.explorers.Explorer]*) – Action explorer.
- **deterministic** (*bool*) – Flag to collect data with the greedy policy.
- **n_steps** (*int*) – Number of total steps to train.
- **show_progress** (*bool*) – Flag to show progress bar for iterations.

Returns Replay buffer with the collected data.

Return type *d3rlpy.dataset.replay_buffer.ReplayBuffer*

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.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.base.QLearningAlgoImplBase, d3rlpy.base.LearnableConfig]*) – 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.qlearning.base.QLearningAlgoBase*[d3rlpy.algos.qlearning.base.QLearningAlgoImplBase, d3rlpy.base.LearnableConfig]*) – 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()
```

(continues on next page)

(continued from previous page)

```
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.qlearning.base.QLearningAlgoBase[d3rlpy.algos.qlearning.base.QLearningAlgoImplBase, d3rlpy.base.LearnableConfig]) – 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.qlearning.base.QLearningAlgoBase[d3rlpy.algos.qlearning.base.QLearningAlgoImplBase, d3rlpy.base.LearnableConfig]) – 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** (*Union[Sequence[int], Sequence[Sequence[int]]]*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

Return type None

fit(*dataset*, *n_steps*, *n_steps_per_epoch=10000*, *experiment_name=None*, *with_timestamp=True*, *logger_adapter=<d3rlpy.logging.file_adapter.FileAdapterFactory object>*, *show_progress=True*, *save_interval=1*, *evaluators=None*, *callback=None*, *epoch_callback=None*)
Trains with given dataset.

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

Parameters

- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – ReplayBuffer object.
- **n_steps** (*int*) – Number of steps to train.

- **n_steps_per_epoch** (*int*) – Number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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.
- **logger_adapter** (d3rlpy.logging.logger.LoggerAdapterFactory) – LoggerAdapterFactory object.
- **show_progress** (*bool*) – Flag to show progress bar for iterations.
- **save_interval** (*int*) – Interval to save parameters.
- **evaluators** (*Optional[Dict[str, d3rlpy.metrics.evaluators.EvaluatorProtocol]]*) – List of evaluators.
- **callback** (*Optional[Callable[[typing_extensions.Self, int, int], None]]*) – Callable function that takes `(algo, epoch, total_step)` , which is called every step.
- **epoch_callback** (*Optional[Callable[[typing_extensions.Self, int, int], None]]*) – Callable function that takes `(algo, epoch, total_step)`, which is called at the end of every epoch.

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_interval=1*, *experiment_name=None*, *with_timestamp=True*, *logger_adapter=<d3rlpy.logging.file_adapter.FileAdapterFactory object>*, *show_progress=True*, *callback=None*)

Start training loop of online deep reinforcement learning.

Parameters

- **env** (*gym.core.Env[Any, Any]*) – Gym-like environment.
- **buffer** (*Optional[*d3rlpy.dataset.replay_buffer.ReplayBuffer*]*) – Replay buffer.
- **explorer** (*Optional[d3rlpy.algos.qlearning.explorers.Explorer]*) – Action explorer.
- **n_steps** (*int*) – Number of total steps to train.
- **n_steps_per_epoch** (*int*) – Number of steps per epoch.
- **update_interval** (*int*) – Number of steps per update.
- **update_start_step** (*int*) – Steps before starting updates.
- **random_steps** (*int*) – Steps for the initial random explortion.
- **eval_env** (*Optional[gym.core.Env[Any, Any]]*) – Gym-like environment. If None, evaluation is skipped.
- **eval_epsilon** (*float*) – ϵ-greedy factor during evaluation.
- **save_interval** (*int*) – 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.
- **logger_adapter** (d3rlpy.logging.logger.LoggerAdapterFactory) – LoggerAdapterFactory object.
- **show_progress** (*bool*) – Flag to show progress bar for iterations.
- **callback** (*Optional[Callable[[typing_extensions.Self, 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_steps*, *n_steps_per_epoch=10000*, *experiment_name=None*, *with_timestamp=True*, *logger_adapter=<d3rlpy.logging.file_adapter.FileAdapterFactory object>*, *show_progress=True*, *save_interval=1*, *evaluators=None*, *callback=None*, *epoch_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** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – Offline dataset to train.
- **n_steps** (*int*) – Number of steps to train.
- **n_steps_per_epoch** (*int*) – Number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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.
- **logger_adapter** (d3rlpy.logging.logger.LoggerAdapterFactory) – LoggerAdapterFactory object.
- **show_progress** (*bool*) – Flag to show progress bar for iterations.
- **save_interval** (*int*) – Interval to save parameters.
- **evaluators** (*Optional[Dict[str, d3rlpy.metrics.evaluators.EvaluatorProtocol]]*) – List of evaluators.
- **callback** (*Optional[Callable[[typing_extensions.Self, int, int], None]]*) – Callable function that takes `(algo, epoch, total_step)` , which is called every step.
- **epoch_callback** (*Optional[Callable[[typing_extensions.Self, int, int], None]]*) – Callable function that takes `(algo, epoch, total_step)`, which is called at the end of every epoch.

Returns Iterator yielding current epoch and metrics dict.

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

classmethod from_json(*fname*, *device=False*)
Construct algorithm from params.json file.

```
from d3rlpy.algos import CQL

cql = CQL.from_json("<path-to-json>", device='cuda:0')
```

Parameters

- **fname** (*str*) – path to params.json
- **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type typing_extensions.Self

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_create_impl(*observation_shape*, *action_size*)

Parameters

- **observation_shape** (*Union[Sequence[int], Sequence[Sequence[int]]]*) –
- **action_size** (*int*) –

Return type None

inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

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))
```

(continues on next page)

(continued from previous page)

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

Parameters **x** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observations

Returns Greedy actions

Return type numpy.ndarray

predict_value(*x*, *action*)

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,)
```

Parameters

- **x** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observations
- **action** (*numpy.ndarray*) – Actions

Returns Predicted action-values

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*)

Returns sampled actions.

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

Parameters **x** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observations.

Returns Sampled actions.

Return type numpy.ndarray

save(*fname*)

Saves paired data of neural network parameters and serialized config.

```
algo.save('model.d3')

# reconstruct everything
algo2 = d3rlpy.load_learnable("model.d3", device="cuda:0")
```

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

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_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_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

update(*batch*)

Update parameters with mini-batch of data.

Parameters **batch** (*d3rlpy.dataset.mini_batch.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]

`algo`

`batch_size`
Batch size to train.

Returns batch size.

Return type int

`config`
Config.

Returns config.

Return type LearnableConfig

`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]

`observation_scaler`
Preprocessing observation scaler.

Returns preprocessing observation scaler.

Return type Optional[ObservationScaler]

`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]

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*, *config*, *device=False*, *impl=None*)
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.qlearning.base.QLearningAlgoBase) – Algorithm to evaluate.
- **config** (*d3rlpy.ope.FQEConfig*) – FQE config.
- **device** (*bool, int or str*) – Flag to use GPU, device ID or PyTorch device identifier.
- **impl** (*d3rlpy.metrics.ope.torch.DiscreteFQEImpl*) – Algorithm implementation.

Methods

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

Parameters **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – dataset.

Return type None

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

Parameters **env** (*gym.core.Env[Any, Any]*) – gym-like environment.

Return type None

collect(*env*, *buffer=None*, *explorer=None*, *deterministic=False*, *n_steps=1000000*, *show_progress=True*)
Collects data via interaction with environment.

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

Parameters

- **env** (*gym.core.Env[Any, Any]*) – Fym-like environment.
- **buffer** (*Optional[d3rlpy.dataset.replay_buffer.ReplayBuffer]*) – Replay buffer.
- **explorer** (*Optional[d3rlpy.algos.qlearning.explorers.Explorer]*) – Action explorer.
- **deterministic** (*bool*) – Flag to collect data with the greedy policy.
- **n_steps** (*int*) – Number of total steps to train.
- **show_progress** (*bool*) – Flag to show progress bar for iterations.

Returns Replay buffer with the collected data.

Return type *d3rlpy.dataset.replay_buffer.ReplayBuffer*

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.qlearning.base.QLearningAlgoBase[d3rlpy.algos.qlearning.base.QLearningAlgoImplBase, d3rlpy.base.LearnableConfig]) – 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.qlearning.base.QLearningAlgoBase[d3rlpy.algos.qlearning.base.QLearningAlgoImplBase, d3rlpy.base.LearnableConfig]) – 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.qlearning.base.QLearningAlgoBase[d3rlpy.algos.qlearning.base.QLearningAlgoImplBase, d3rlpy.base.LearnableConfig]*) – 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.qlearning.base.QLearningAlgoBase[d3rlpy.algos.qlearning.base.QLearningAlgoImplBase, d3rlpy.base.LearnableConfig]*) – 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** (*Union[Sequence[int], Sequence[Sequence[int]]]*) – observation shape.
- **action_size** (*int*) – dimension of action-space.

Return type None

fit(*dataset*, *n_steps*, *n_steps_per_epoch=10000*, *experiment_name=None*, *with_timestamp=True*, *logger_adapter=<d3rlpy.logging.file_adapter.FileAdapterFactory object>*, *show_progress=True*, *save_interval=1*, *evaluators=None*, *callback=None*, *epoch_callback=None*)
Trains with given dataset.

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

Parameters

- **dataset** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – ReplayBuffer object.
- **n_steps** (*int*) – Number of steps to train.
- **n_steps_per_epoch** (*int*) – Number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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.
- **logger_adapter** (d3rlpy.logging.logger.LoggerAdapterFactory) – LoggerAdapterFactory object.
- **show_progress** (*bool*) – Flag to show progress bar for iterations.
- **save_interval** (*int*) – Interval to save parameters.
- **evaluators** (*Optional[Dict[str, d3rlpy.metrics.evaluators.EvaluatorProtocol]]*) – List of evaluators.
- **callback** (*Optional[Callable[[typing_extensions.Self, int, int], None]]*) – Callable function that takes `(algo, epoch, total_step)` , which is called every step.
- **epoch_callback** (*Optional[Callable[[typing_extensions.Self, int, int], None]]*) – Callable function that takes `(algo, epoch, total_step)`, which is called at the end of every epoch.

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_interval=1*, *experiment_name=None*, *with_timestamp=True*, *logger_adapter=<d3rlpy.logging.file_adapter.FileAdapterFactory object>*, *show_progress=True*, *callback=None*)

Start training loop of online deep reinforcement learning.

Parameters

- **env** (*gym.core.Env[Any, Any]*) – Gym-like environment.
- **buffer** (*Optional[d3rlpy.dataset.replay_buffer.ReplayBuffer]*) – Replay buffer.
- **explorer** (*Optional[d3rlpy.algos.qlearning.explorers.Explorer]*) – Action explorer.
- **n_steps** (*int*) – Number of total steps to train.
- **n_steps_per_epoch** (*int*) – Number of steps per epoch.
- **update_interval** (*int*) – Number of steps per update.
- **update_start_step** (*int*) – Steps before starting updates.
- **random_steps** (*int*) – Steps for the initial random explortion.
- **eval_env** (*Optional[gym.core.Env[Any, Any]]*) – Gym-like environment. If None, evaluation is skipped.

- **eval_epsilon** (*float*) – ϵ-greedy factor during evaluation.
- **save_interval** (*int*) – 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.
- **logger_adapter** (d3rlpy.logging.logger.LoggerAdapterFactory) – LoggerAdapterFactory object.
- **show_progress** (*bool*) – Flag to show progress bar for iterations.
- **callback** (*Optional[Callable[[typing_extensions.Self, 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_steps*, *n_steps_per_epoch=10000*, *experiment_name=None*, *with_timestamp=True*, *logger_adapter=<d3rlpy.logging.file_adapter.FileAdapterFactory object>*, *show_progress=True*, *save_interval=1*, *evaluators=None*, *callback=None*, *epoch_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** (d3rlpy.dataset.replay_buffer.ReplayBuffer) – Offline dataset to train.
- **n_steps** (*int*) – Number of steps to train.
- **n_steps_per_epoch** (*int*) – Number of steps per epoch. This value will be ignored when `n_steps` is `None`.
- **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.
- **logger_adapter** (d3rlpy.logging.logger.LoggerAdapterFactory) – LoggerAdapterFactory object.
- **show_progress** (*bool*) – Flag to show progress bar for iterations.
- **save_interval** (*int*) – Interval to save parameters.
- **evaluators** (*Optional[Dict[str, d3rlpy.metrics.evaluators.EvaluatorProtocol]]*) – List of evaluators.
- **callback** (*Optional[Callable[[typing_extensions.Self, int, int], None]]*) – Callable function that takes `(algo, epoch, total_step)` , which is called every step.
- **epoch_callback** (*Optional[Callable[[typing_extensions.Self, int, int], None]]*) – Callable function that takes `(algo, epoch, total_step)`, which is called at the end of every epoch.

Returns Iterator yielding current epoch and metrics dict.

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

classmethod from_json(*fname*, *device=False*)
Construct algorithm from params.json file.

```
from d3rlpy.algos import CQL

cql = CQL.from_json("<path-to-json>", device='cuda:0')
```

Parameters

- **fname** (*str*) – path to params.json
- **device** (*Union[int, str, bool]*) – device option. If the value is boolean and True, `cuda:0` will be used. If the value is integer, `cuda:<device>` will be used. If the value is string in torch device style, the specified device will be used.

Returns algorithm object.

Return type typing_extensions.Self

get_action_type()
Returns action type (continuous or discrete).

Returns action type.

Return type d3rlpy.constants.ActionSpace

inner_create_impl(*observation_shape*, *action_size*)

Parameters

- **observation_shape** (*Union[Sequence[int], Sequence[Sequence[int]]]*) –
- **action_size** (*int*) –

Return type None

inner_update(*batch*)
Update parameters with PyTorch mini-batch.

Parameters **batch** (*d3rlpy.torch_utility.TorchMiniBatch*) – PyTorch mini-batch data.

Returns Dictionary of metrics.

Return type Dict[str, float]

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, Sequence[numpy.ndarray]]*) – Observations

Returns Greedy actions

Return type numpy.ndarray

predict_value(*x*, *action*)

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,)
```

Parameters

- **x** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observations
- **action** (*numpy.ndarray*) – Actions

Returns Predicted action-values

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*)

Returns sampled actions.

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

Parameters **x** (*Union[numpy.ndarray, Sequence[numpy.ndarray]]*) – Observations.

Returns Sampled actions.

Return type numpy.ndarray

save(*fname*)

Saves paired data of neural network parameters and serialized config.

```
algo.save('model.d3')

# reconstruct everything
algo2 = d3rlpy.load_learnable("model.d3", device="cuda:0")
```

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

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_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_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

update(*batch*)
Update parameters with mini-batch of data.

Parameters **batch** (*d3rlpy.dataset.mini_batch.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]

algo

batch_size
Batch size to train.

Returns batch size.

Return type int

config
Config.

Returns config.

Return type LearnableConfig

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]

observation_scaler
Preprocessing observation scaler.

Returns preprocessing observation scaler.

Return type Optional[ObservationScaler]

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]

4.10 Logging

d3rlpy provides a customizable interface for logging metrics, `LoggerAdapter` and `LoggerAdapterFactory`.

```
import d3rlpy

dataset, env = d3rlpy.datasets.get_cartpole()

dqn = d3rlpy.algos.DQNConfig().create()

dqn.fit(
   dataset=dataset,
   n_steps=100000,
   # set FileAdapterFactory to save metrics as CSV files
   logger_adapter=d3rlpy.logging.FileAdapterFactory(root_dir="d3rlpy_logs"),
)
```

`LoggerAdapterFactory` is a parent interface that instantiates `LoggerAdapter` at the beginning of training. You can also use `CombineAdapter` to combine multiple `LoggerAdapter` in the same training.

```
# combine FileAdapterFactory and TensorboardAdapterFactory
logger_adapter = d3rlpy.logging.CombineAdapterFactory([
   d3rlpy.logging.FileAdapterFactory(root_dir="d3rlpy_logs"),
   d3rlpy.logging.TensorboardAdapterFactory(root_dir="tensorboard_logs"),
])

dqn.fit(dataset=dataset, n_steps=100000, logger_adapter=logger_adapter)
```

4.10.1 LoggerAdapter

`LoggerAdapter` is an inner interface of `LoggerAdapterFactory`. You can implement your own `LoggerAdapter` for 3rd-party visualizers.

```
import d3rlpy

class CustomAdapter(d3rlpy.logging.LoggerAdapter):
    def write_params(self, params: Dict[str, Any]) -> None:
        # save dictionary as json file
        with open("params.json", "w") as f:
            f.write(json.dumps(params, default=default_json_encoder, indent=2))

    def before_write_metric(self, epoch: int, step: int) -> None:
```

(continues on next page)

(continued from previous page)

```
        pass

    def write_metric(
        self, epoch: int, step: int, name: str, value: float
    ) -> None:
        with open(f"{name}.csv", "a") as f:
            print(f"{epoch},{step},{value}", file=f)

    def after_write_metric(self, epoch: int, step: int) -> None:
        pass

    def save_model(self, epoch: int, algo: Any) -> None:
        algo.save(f"model_{epoch}.d3")

    def close(self) -> None:
        pass
```

d3rlpy.logging.LoggerAdapter	Interface of LoggerAdapter.
d3rlpy.logging.FileAdapter	FileAdapter class.
d3rlpy.logging.TensorboardAdapter	TensorboardAdapter class.
d3rlpy.logging.NoopAdapter	NoopAdapter class.
d3rlpy.logging.CombineAdapter	CombineAdapter class.

d3rlpy.logging.LoggerAdapter

class d3rlpy.logging.**LoggerAdapter**(**args*, ***kwds*)

Interface of LoggerAdapter.

Methods

after_write_metric(*epoch*, *step*)

Callback executed after write_metric method.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.

Return type None

before_write_metric(*epoch*, *step*)

Callback executed before write_metric method.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.

Return type None

close()

Closes this LoggerAdapter.

Return type None

save_model(*epoch*, *algo*)
Saves models.

Parameters

- **epoch** (*int*) – Epoch.
- **algo** (*d3rlpy.logging.logger.SaveProtocol*) – Algorithm that provides `save` method.

Return type None

write_metric(*epoch*, *step*, *name*, *value*)
Writes metric.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.
- **name** (*str*) – Metric name.
- **value** (*float*) – Metric value.

Return type None

write_params(*params*)
Writes hyperparameters.

Parameters **params** (*Dict[str, Any]*) – Dictionary of hyperparameters.

Return type None

d3rlpy.logging.FileAdapter

class d3rlpy.logging.**FileAdapter**(**args*, ***kwds*)
FileAdapter class.

This class saves metrics as CSV files, hyperparameters as json file and models as d3 files.

Parameters **logdir** (*str*) – Log directory.

Methods

after_write_metric(*epoch*, *step*)
Callback executed after write_metric method.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.

Return type None

before_write_metric(*epoch*, *step*)
Callback executed before write_metric method.

Parameters

- **epoch** (*int*) – Epoch.

- **step** (*int*) – Training step.

Return type None

close()
Closes this LoggerAdapter.

Return type None

save_model(*epoch*, *algo*)
Saves models.

Parameters

- **epoch** (*int*) – Epoch.
- **algo** (*d3rlpy.logging.logger.SaveProtocol*) – Algorithm that provides `save` method.

Return type None

write_metric(*epoch*, *step*, *name*, *value*)
Writes metric.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.
- **name** (*str*) – Metric name.
- **value** (*float*) – Metric value.

Return type None

write_params(*params*)
Writes hyperparameters.

Parameters **params** (*Dict[str, Any]*) – Dictionary of hyperparameters.

Return type None

Attributes

logdir

d3rlpy.logging.TensorboardAdapter

class d3rlpy.logging.**TensorboardAdapter**(**args*, ***kwds*)
TensorboardAdapter class.

This class saves metrics for Tensorboard visualization, powered by tensorboardX.

Note that this class does not save models. If you want to save models during training, consider `FileAdapter` as well.

Parameters

- **root_dir** (*str*) – Top-level log directory.
- **experiment_name** (*str*) – Experiment name.

Methods

after_write_metric(*epoch*, *step*)
Callback executed after write_metric method.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.

Return type None

before_write_metric(*epoch*, *step*)
Callback executed before write_metric method.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.

Return type None

close()
Closes this LoggerAdapter.

Return type None

save_model(*epoch*, *algo*)
Saves models.

Parameters

- **epoch** (*int*) – Epoch.
- **algo** (*d3rlpy.logging.logger.SaveProtocol*) – Algorithm that provides `save` method.

Return type None

write_metric(*epoch*, *step*, *name*, *value*)
Writes metric.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.
- **name** (*str*) – Metric name.
- **value** (*float*) – Metric value.

Return type None

write_params(*params*)
Writes hyperparameters.

Parameters **params** (*Dict[str, Any]*) – Dictionary of hyperparameters.

Return type None

d3rlpy.logging.NoopAdapter

class d3rlpy.logging.**NoopAdapter**(**args*, ***kwds*)
NoopAdapter class.

This class does not save anything. This can be used especially when programs are not allowed to write things to disks.

Methods

after_write_metric(*epoch*, *step*)
Callback executed after write_metric method.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.

Return type None

before_write_metric(*epoch*, *step*)
Callback executed before write_metric method.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.

Return type None

close()
Closes this LoggerAdapter.

Return type None

save_model(*epoch*, *algo*)
Saves models.

Parameters

- **epoch** (*int*) – Epoch.
- **algo** (*d3rlpy.logging.logger.SaveProtocol*) – Algorithm that provides `save` method.

Return type None

write_metric(*epoch*, *step*, *name*, *value*)
Writes metric.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.
- **name** (*str*) – Metric name.
- **value** (*float*) – Metric value.

Return type None

write_params(*params*)
Writes hyperparameters.

Parameters **params** (*Dict[str, Any]*) – Dictionary of hyperparameters.

Return type None

d3rlpy.logging.CombineAdapter

class d3rlpy.logging.**CombineAdapter**(**args*, ***kwds*)

CombineAdapter class.

This class combines multiple LoggerAdapter to write metrics through different adapters at the same time.

Parameters **adapters** (*Sequence[LoggerAdapter]*) – List of LoggerAdapter.

Methods

after_write_metric(*epoch*, *step*)

Callback executed after write_metric method.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.

Return type None

before_write_metric(*epoch*, *step*)

Callback executed before write_metric method.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.

Return type None

close()

Closes this LoggerAdapter.

Return type None

save_model(*epoch*, *algo*)

Saves models.

Parameters

- **epoch** (*int*) – Epoch.
- **algo** (*d3rlpy.logging.logger.SaveProtocol*) – Algorithm that provides `save` method.

Return type None

write_metric(*epoch*, *step*, *name*, *value*)

Writes metric.

Parameters

- **epoch** (*int*) – Epoch.
- **step** (*int*) – Training step.
- **name** (*str*) – Metric name.

- **value** (*float*) – Metric value.

Return type None

write_params(*params*)
Writes hyperparameters.

Parameters **params** (*Dict[str, Any]*) – Dictionary of hyperparameters.

Return type None

4.10.2 LoggerAdapterFactory

LoggerAdapterFactory is an interface that instantiates LoggerAdapter at the beginning of training. You can implement your own LoggerAdapterFactory for 3rd-party visualizers.

```
import d3rlpy

class CustomAdapterFactory(d3rlpy.logging.LoggerAdapterFactory):
    def create(self, experiment_name: str) -> d3rlpy.logging.FileAdapter:
        return CustomAdapter()
```

d3rlpy.logging.LoggerAdapterFactory	Interface of LoggerAdapterFactory.
d3rlpy.logging.FileAdapterFactory	FileAdapterFactory class.
d3rlpy.logging.TensorboardAdapterFactory	TensorboardAdapterFactory class.
d3rlpy.logging.NoopAdapterFactory	NoopAdapterFactory class.
d3rlpy.logging.CombineAdapterFactory	CombineAdapterFactory class.

d3rlpy.logging.LoggerAdapterFactory

class d3rlpy.logging.**LoggerAdapterFactory**(**args*, ***kwds*)
Interface of LoggerAdapterFactory.

Methods

create(*experiment_name*)
Creates LoggerAdapter.

This method instantiates LoggerAdapter with a given experiment_name. This method is usually called at the beginning of training.

Parameters **experiment_name** (*str*) – Experiment name.

Return type *d3rlpy.logging.logger.LoggerAdapter*

d3rlpy.logging.FileAdapterFactory

class d3rlpy.logging.**FileAdapterFactory**(**args*, ***kwds*)
FileAdapterFactory class.

This class instantiates `FileAdapter` object. Log directory will be created at `<root_dir>/<experiment_name>`.

Parameters **root_dir** (*str*) – Top-level log directory.

Methods

create(*experiment_name*)
Creates LoggerAdapter.

This method instantiates `LoggerAdapter` with a given `experiment_name`. This method is usually called at the beginning of training.

Parameters **experiment_name** (*str*) – Experiment name.

Return type *d3rlpy.logging.file_adapter.FileAdapter*

d3rlpy.logging.TensorboardAdapterFactory

class d3rlpy.logging.**TensorboardAdapterFactory**(**args*, ***kwds*)
TensorboardAdapterFactory class.

This class instantiates `TensorboardAdapter` object.

Parameters **root_dir** (*str*) – Top-level log directory.

Methods

create(*experiment_name*)
Creates LoggerAdapter.

This method instantiates `LoggerAdapter` with a given `experiment_name`. This method is usually called at the beginning of training.

Parameters **experiment_name** (*str*) – Experiment name.

Return type *d3rlpy.logging.tensorboard_adapter.TensorboardAdapter*

d3rlpy.logging.NoopAdapterFactory

class d3rlpy.logging.**NoopAdapterFactory**(**args*, ***kwds*)
NoopAdapterFactory class.

This class instantiates `NoopAdapter` object.

Methods

create(*experiment_name*)
Creates LoggerAdapter.

This method instantiates `LoggerAdapter` with a given `experiment_name`. This method is usually called at the beginning of training.

Parameters **experiment_name** (*str*) – Experiment name.

Return type *d3rlpy.logging.noop_adapter.NoopAdapter*

d3rlpy.logging.CombineAdapterFactory

class d3rlpy.logging.**CombineAdapterFactory**(**args*, ***kwds*)
CombineAdapterFactory class.

This class instantiates `CombineAdapter` object.

Parameters **adapter_factories** (*Sequence[LoggerAdapterFactory]*) – List of LoggerAdapterFactory.

Methods

create(*experiment_name*)
Creates LoggerAdapter.

This method instantiates `LoggerAdapter` with a given `experiment_name`. This method is usually called at the beginning of training.

Parameters **experiment_name** (*str*) – Experiment name.

Return type *d3rlpy.logging.utils.CombineAdapter*

4.11 Online Training

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

```
import d3lpy
import gym

# setup environment
env = gym.make('CartPole-v1')
eval_env = gym.make('CartPole-v1')

# setup algorithm
dqn = d3rlpy.algos.DQN(
    batch_size=32,
    learning_rate=2.5e-4,
    target_update_interval=100,
).create(device="cuda:0")

# setup replay buffer
```

(continues on next page)

(continued from previous page)

```
buffer = d3rlpy.dataset.create_fifo_replay_buffer(limit=100000, env=env)

# setup explorers
explorer = d3rlpy.algos.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.
    n_steps_per_epoch=1000,
    update_interval=10, # update parameters every 10 steps.
)
```

4.11.1 Explorers

d3rlpy.algos.ConstantEpsilonGreedy	ϵ-greedy explorer with constant ϵ.
d3rlpy.algos.LinearDecayEpsilonGreedy	ϵ-greedy explorer with linear decay schedule.
d3rlpy.algos.NormalNoise	Normal noise explorer.

d3rlpy.algos.ConstantEpsilonGreedy

class d3rlpy.algos.**ConstantEpsilonGreedy**(*epsilon*)

ϵ-greedy explorer with constant ϵ.

Parameters **epsilon** (*float*) – the constant ϵ.

Methods

sample(*algo*, *x*, *step*)

Parameters

- **algo** (*d3rlpy.algos.qlearning.explorers._ActionProtocol*) –
- **x** (*numpy.ndarray*) –
- **step** (*int*) –

Return type numpy.ndarray

d3rlpy.algos.LinearDecayEpsilonGreedy

class d3rlpy.algos.**LinearDecayEpsilonGreedy**(*start_epsilon=1.0*, *end_epsilon=0.1*, *duration=1000000*)
ϵ-greedy explorer with linear decay schedule.

Parameters

- **start_epsilon** (*float*) – Initial ϵ.
- **end_epsilon** (*float*) – Final ϵ.
- **duration** (*int*) – Scheduling duration.

Methods

compute_epsilon(*step*)
Returns decayed ϵ.

Returns ϵ.

Parameters **step** (*int*) –

Return type float

sample(*algo*, *x*, *step*)
Returns ϵ-greedy action.

Parameters

- **algo** (*d3rlpy.algos.qlearning.explorers._ActionProtocol*) – Algorithm.
- **x** (*numpy.ndarray*) – Observation.
- **step** (*int*) – Current environment step.

Returns ϵ-greedy action.

Return type numpy.ndarray

d3rlpy.algos.NormalNoise

class d3rlpy.algos.**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.algos.qlearning.explorers._ActionProtocol*) – Algorithm.
- **x** (*numpy.ndarray*) – Observation.
- **step** (*int*) –

Returns Action with noise injection.

Return type numpy.ndarray

CHAPTER

FIVE

COMMAND LINE INTERFACE

d3rlpy provides the convenient CLI tool.

5.1 plot

Plot the saved metrics by specifying paths:

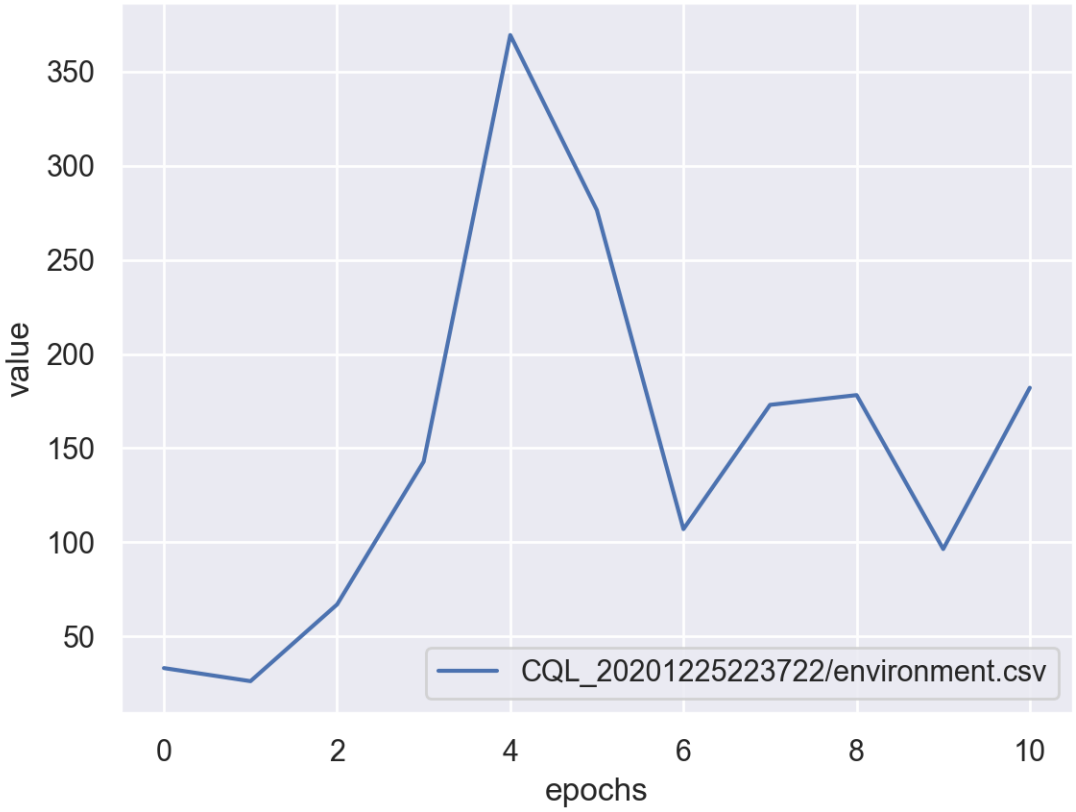
```
$ d3rlpy plot <path> [<path>...]
```

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
```



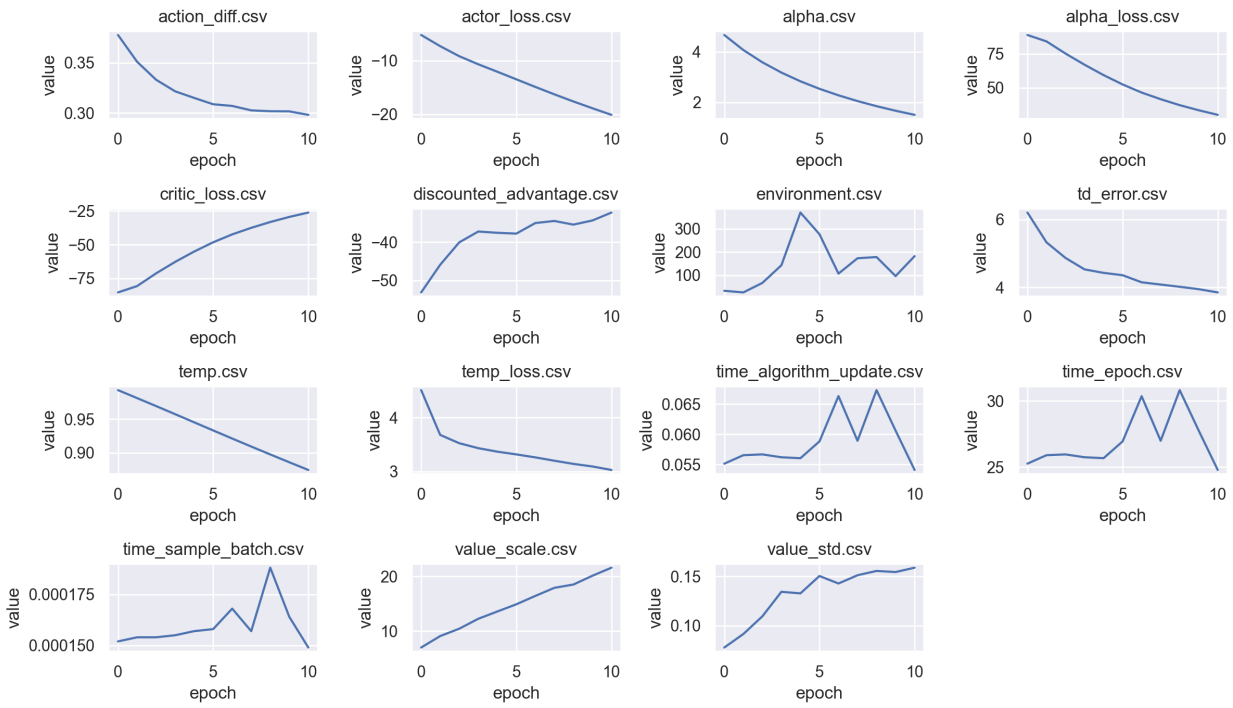
5.2 plot-all

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 (`.onnx`) and TorchScript (`.pt`):

```
$ d3rlpy export <model_path> <out_path>
```

example:

```
$ d3rlpy export d3rlpy_logs/CQL_20201224224314/model_100.d3 policy.onnx
```

5.4 record

Record evaluation episodes as videos with the saved model:

```
$ d3rlpy record <path> --env-id <environment id>
```

Table 2: options

option	description
`--env-id`	Gym environment id.
`--env-header`	Arbitrary Python code to define environment to evaluate.
`--out`	Output directory.
`--n-episodes`	The number of episodes to record.
`--frame-rate`	Video frame rate.
`--record-rate`	Images are recored every `record-rate` frames.
`--epsilon`	ϵ-greedy evaluation.
`--target-return`	The target environment return for Decision Transformer algorithms.

example:

```
# record simple environment
$ d3rlpy record d3rlpy_logs/CQL_20201224224314/model_100.d3 --env-id HopperBulletEnv-v0

# record wrapped environment
$ d3rlpy record d3rlpy_logs/Discrete_CQL_20201224224314/model_100.d3 \
    --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 3: options

option	description
`--env-id`	Gym environment id.
`--env-header`	Arbitrary Python code to define environment to evaluate.
`--n-episodes`	The number of episodes to run.
`--target-return`	The target environment return for Decision Transformer algorithms.

example:

```
# record simple environment
$ d3rlpy play d3rlpy_logs/CQL_20201224224314/model_100.d3 --env-id HopperBulletEnv-v0

# record wrapped environment
$ d3rlpy play d3rlpy_logs/Discrete_CQL_20201224224314/model_100.d3 \
    --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 -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')
d3rlpy.envs.seed_env(env, 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.

```
import d3rlpy

# 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 = d3rlpy.dataset.MDPDataset(
    observations=observations,
    actions=actions,
    rewards=rewards,
    terminals=terminals,
    # stack last 4 frames (stacked shape is [4, 84, 84])
    transition_picker=d3rlpy.dataset.FrameStackTransitionPicker(n_frames=4),
)

dqn = DQNConfig(
```

(continues on next page)

(continued from previous page)

```
    observation_scaler=d3rlpy.preprocessing.PixelObservationScaler(),  # pixels are␣
↪devided by 255
).create()
```

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.

```
import d3rlpy

# use batch normalization
# this seems to improve performance with discrete action-spaces
encoder = d3rlpy.models.DefaultEncoderFactory(use_batch_norm=True)
# use distributional Q function leading to robust improvement
q_func = d3rlpy.models.QRQFunctionFactory()
dqn = d3rlpy.algos.DQNConfig(
    encoder_factory=encoder,
    q_func_factory=q_func,
).create()

# use dropout
# this could dramatically improve performance
encoder = d3rlpy.models.DefaultEncoderFactory(dropout_rate=0.2)
sac = d3rlpy.algos.SACConfig(actor_encoder_factory=encoder).create()

# multi-step transition sampling
transition_picker = d3rlpy.dataset.MultiStepTransitionPicker(
    n_steps=3,
    gamma=0.99,
)
# replay buffer for experience replay
buffer = d3rlpy.dataset.create_fifo_replay_buffer(
    limit=100000,
    env=env,
    transition_picker=transition_picker,
)
```

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

d3rlpy, 27
d3rlpy.algos, 197
d3rlpy.dataset, 84
d3rlpy.datasets, 104
d3rlpy.logging, 188
d3rlpy.metrics, 162
d3rlpy.models, 77
d3rlpy.ope, 169
d3rlpy.preprocessing, 107

INDEX

Symbols

__call__() (*d3rlpy.dataset.BasicTrajectorySlicer method*), 101
__call__() (*d3rlpy.dataset.BasicTransitionPicker method*), 98
__call__() (*d3rlpy.dataset.FrameStackTransitionPicker method*), 99
__call__() (*d3rlpy.dataset.MultiStepTransitionPicker method*), 99
__call__() (*d3rlpy.dataset.TrajectorySlicerProtocol method*), 101
__call__() (*d3rlpy.dataset.TransitionPickerProtocol method*), 98
__call__() (*d3rlpy.metrics.AverageValueEstimationEvaluator method*), 165
__call__() (*d3rlpy.metrics.CompareContinuousActionDiffEvaluator method*), 168
__call__() (*d3rlpy.metrics.CompareDiscreteActionMatchEvaluator method*), 169
__call__() (*d3rlpy.metrics.ContinuousActionDiffEvaluator method*), 167
__call__() (*d3rlpy.metrics.DiscountedSumOfAdvantageEvaluator method*), 164
__call__() (*d3rlpy.metrics.DiscreteActionMatchEvaluator method*), 167
__call__() (*d3rlpy.metrics.EnvironmentEvaluator method*), 168
__call__() (*d3rlpy.metrics.InitialStateValueEstimationEvaluator method*), 165
__call__() (*d3rlpy.metrics.SoftOPCEvaluator method*), 166
__call__() (*d3rlpy.metrics.TDErrorEvaluator method*), 163
__getitem__() (*d3rlpy.dataset.BufferProtocol method*), 95
__getitem__() (*d3rlpy.dataset.FIFOBuffer method*), 96
__getitem__() (*d3rlpy.dataset.InfiniteBuffer method*), 96
__len__() (*d3rlpy.dataset.FIFOBuffer method*), 96
__len__() (*d3rlpy.dataset.InfiniteBuffer method*), 96

A

action_scaler (*d3rlpy.base.LearnableBase property*), 27
action_scaler (*d3rlpy.ope.DiscreteFQE attribute*), 187
action_scaler (*d3rlpy.ope.FQE attribute*), 178
action_size (*d3rlpy.base.LearnableBase property*), 27
action_size (*d3rlpy.ope.DiscreteFQE attribute*), 187
action_size (*d3rlpy.ope.FQE attribute*), 178
activation (*d3rlpy.models.DefaultEncoderFactory attribute*), 155
activation (*d3rlpy.models.DenseEncoderFactory attribute*), 162
activation (*d3rlpy.models.PixelEncoderFactory attribute*), 157
activation (*d3rlpy.models.VectorEncoderFactory attribute*), 160
AdamFactory (*class in d3rlpy.models*), 146
after_write_metric() (*d3rlpy.logging.CombineAdapter method*), 194
after_write_metric() (*d3rlpy.logging.FileAdapter method*), 190
after_write_metric() (*d3rlpy.logging.LoggerAdapter method*), 189
after_write_metric() (*d3rlpy.logging.NoopAdapter method*), 193
after_write_metric() (*d3rlpy.logging.TensorboardAdapter method*), 192
algo (*d3rlpy.ope.DiscreteFQE attribute*), 187
algo (*d3rlpy.ope.FQE attribute*), 178
alpha (*d3rlpy.models.RMSpropFactory attribute*), 150
amsgrad (*d3rlpy.models.AdamFactory attribute*), 148
append() (*d3rlpy.dataset.BufferProtocol method*), 95
append() (*d3rlpy.dataset.FIFOBuffer method*), 96
append() (*d3rlpy.dataset.InfiniteBuffer method*), 96
append() (*d3rlpy.dataset.MDPDataset method*), 86
append() (*d3rlpy.dataset.ReplayBuffer method*), 91
append_episode() (*d3rlpy.dataset.MDPDataset method*), 86
append_episode() (*d3rlpy.dataset.ReplayBuffer*

method), 91
as_stateful_wrapper() (*d3rlpy.algos.TransformerAlgoBase method*), 74
AverageValueEstimationEvaluator (*class in d3rlpy.metrics*), 164
AWAC (*class in d3rlpy.algos*), 62
AWACConfig (*class in d3rlpy.algos*), 61

B

BasicTrajectorySlicer (*class in d3rlpy.dataset*), 101
BasicTransitionPicker (*class in d3rlpy.dataset*), 98
BasicWriterPreprocess (*class in d3rlpy.dataset*), 103
batch_size (*d3rlpy.base.LearnableBase property*), 27
batch_size (*d3rlpy.ope.DiscreteFQE attribute*), 187
batch_size (*d3rlpy.ope.FQE attribute*), 178
BC (*class in d3rlpy.algos*), 37
BCConfig (*class in d3rlpy.algos*), 36
BCQ (*class in d3rlpy.algos*), 51
BCQConfig (*class in d3rlpy.algos*), 49
BEAR (*class in d3rlpy.algos*), 55
BEARConfig (*class in d3rlpy.algos*), 53
before_write_metric() (*d3rlpy.logging.CombineAdapter method*), 194
before_write_metric() (*d3rlpy.logging.FileAdapter method*), 190
before_write_metric() (*d3rlpy.logging.LoggerAdapter method*), 189
before_write_metric() (*d3rlpy.logging.NoopAdapter method*), 193
before_write_metric() (*d3rlpy.logging.TensorboardAdapter method*), 192
betas (*d3rlpy.models.AdamFactory attribute*), 148
buffer (*d3rlpy.dataset.MDPDataset attribute*), 88
buffer (*d3rlpy.dataset.ReplayBuffer attribute*), 93
BufferProtocol (*class in d3rlpy.dataset*), 95
build_with_dataset() (*d3rlpy.base.LearnableBase method*), 28
build_with_dataset() (*d3rlpy.ope.DiscreteFQE method*), 179
build_with_dataset() (*d3rlpy.ope.FQE method*), 170
build_with_env() (*d3rlpy.base.LearnableBase method*), 28
build_with_env() (*d3rlpy.ope.DiscreteFQE method*), 179
build_with_env() (*d3rlpy.ope.FQE method*), 170
built (*d3rlpy.preprocessing.ClipRewardScaler attribute*), 132
built (*d3rlpy.preprocessing.ConstantShiftRewardScaler attribute*), 141
built (*d3rlpy.preprocessing.MinMaxActionScaler attribute*), 121
built (*d3rlpy.preprocessing.MinMaxObservationScaler attribute*), 114
built (*d3rlpy.preprocessing.MinMaxRewardScaler attribute*), 125
built (*d3rlpy.preprocessing.MultiplyRewardScaler attribute*), 135
built (*d3rlpy.preprocessing.PixelObservationScaler attribute*), 111
built (*d3rlpy.preprocessing.ReturnBasedRewardScaler attribute*), 138
built (*d3rlpy.preprocessing.StandardObservationScaler attribute*), 118
built (*d3rlpy.preprocessing.StandardRewardScaler attribute*), 129

C

centered (*d3rlpy.models.RMSpropFactory attribute*), 150
clip_episode() (*d3rlpy.dataset.MDPDataset method*), 86
clip_episode() (*d3rlpy.dataset.ReplayBuffer method*), 91
ClipRewardScaler (*class in d3rlpy.preprocessing*), 129
close() (*d3rlpy.logging.CombineAdapter method*), 194
close() (*d3rlpy.logging.FileAdapter method*), 191
close() (*d3rlpy.logging.LoggerAdapter method*), 189
close() (*d3rlpy.logging.NoopAdapter method*), 193
close() (*d3rlpy.logging.TensorboardAdapter method*), 192
collect() (*d3rlpy.algos.QLearningAlgoBase method*), 30
collect() (*d3rlpy.ope.DiscreteFQE method*), 179
collect() (*d3rlpy.ope.FQE method*), 170
CombineAdapter (*class in d3rlpy.logging*), 194
CombineAdapterFactory (*class in d3rlpy.logging*), 197
CompareContinuousActionDiffEvaluator (*class in d3rlpy.metrics*), 168
CompareDiscreteActionMatchEvaluator (*class in d3rlpy.metrics*), 169
compute_epsilon() (*d3rlpy.algos.LinearDecayEpsilonGreedy method*), 199
config (*d3rlpy.base.LearnableBase property*), 28
config (*d3rlpy.ope.DiscreteFQE attribute*), 187
config (*d3rlpy.ope.FQE attribute*), 178
ConstantEpsilonGreedy (*class in d3rlpy.algos*), 198
ConstantShiftRewardScaler (*class in d3rlpy.preprocessing*), 138
ContinuousActionDiffEvaluator (*class in d3rlpy.metrics*), 166
copy_policy_from() (*d3rlpy.algos.QLearningAlgoBase method*), 30

copy_policy_from() (*d3rlpy.ope.DiscreteFQE method*), 180
copy_policy_from() (*d3rlpy.ope.FQE method*), 171
copy_policy_optim_from() (*d3rlpy.algos.QLearningAlgoBase method*), 31
copy_policy_optim_from() (*d3rlpy.ope.DiscreteFQE method*), 180
copy_policy_optim_from() (*d3rlpy.ope.FQE method*), 171
copy_q_function_from() (*d3rlpy.algos.QLearningAlgoBase method*), 31
copy_q_function_from() (*d3rlpy.ope.DiscreteFQE method*), 181
copy_q_function_from() (*d3rlpy.ope.FQE method*), 171
copy_q_function_optim_from() (*d3rlpy.algos.QLearningAlgoBase method*), 31
copy_q_function_optim_from() (*d3rlpy.ope.DiscreteFQE method*), 181
copy_q_function_optim_from() (*d3rlpy.ope.FQE method*), 172
CQL (*class in d3rlpy.algos*), 59
CQLConfig (*class in d3rlpy.algos*), 57
create() (*d3rlpy.algos.AWACConfig method*), 62
create() (*d3rlpy.algos.BCConfig method*), 36
create() (*d3rlpy.algos.BCQConfig method*), 50
create() (*d3rlpy.algos.BEARConfig method*), 55
create() (*d3rlpy.algos.CQLConfig method*), 59
create() (*d3rlpy.algos.CRRConfig method*), 57
create() (*d3rlpy.algos.DDPGConfig method*), 43
create() (*d3rlpy.algos.DecisionTransformerConfig method*), 76
create() (*d3rlpy.algos.DiscreteBCConfig method*), 37
create() (*d3rlpy.algos.DiscreteBCQConfig method*), 52
create() (*d3rlpy.algos.DiscreteCQLConfig method*), 60
create() (*d3rlpy.algos.DiscreteRandomPolicyConfig method*), 72
create() (*d3rlpy.algos.DiscreteSACConfig method*), 48
create() (*d3rlpy.algos.DoubleDQNConfig method*), 41
create() (*d3rlpy.algos.DQNConfig method*), 40
create() (*d3rlpy.algos.IQLConfig method*), 69
create() (*d3rlpy.algos.NFQConfig method*), 39
create() (*d3rlpy.algos.PLASConfig method*), 64
create() (*d3rlpy.algos.PLASWithPerturbationConfig method*), 65
create() (*d3rlpy.algos.RandomPolicyConfig method*), 70
create() (*d3rlpy.algos.SACConfig method*), 46
create() (*d3rlpy.algos.TD3Config method*), 44
create() (*d3rlpy.algos.TD3PlusBCConfig method*), 67
create() (*d3rlpy.logging.CombineAdapterFactory method*), 197
create() (*d3rlpy.logging.FileAdapterFactory method*), 196
create() (*d3rlpy.logging.LoggerAdapterFactory method*), 195
create() (*d3rlpy.logging.NoopAdapterFactory method*), 197
create() (*d3rlpy.logging.TensorboardAdapterFactory method*), 196
create() (*d3rlpy.models.AdamFactory method*), 146
create() (*d3rlpy.models.DefaultEncoderFactory method*), 153
create() (*d3rlpy.models.DenseEncoderFactory method*), 160
create() (*d3rlpy.models.OptimizerFactory method*), 142
create() (*d3rlpy.models.PixelEncoderFactory method*), 155
create() (*d3rlpy.models.RMSpropFactory method*), 148
create() (*d3rlpy.models.SGDFactory method*), 144
create() (*d3rlpy.models.VectorEncoderFactory method*), 158
create_continuous() (*d3rlpy.models.IQNQFunctionFactory method*), 82
create_continuous() (*d3rlpy.models.MeanQFunctionFactory method*), 78
create_continuous() (*d3rlpy.models.QRQFunctionFactory method*), 80
create_discrete() (*d3rlpy.models.IQNQFunctionFactory method*), 82
create_discrete() (*d3rlpy.models.MeanQFunctionFactory method*), 78
create_discrete() (*d3rlpy.models.QRQFunctionFactory method*), 80
create_fifo_replay_buffer() (*in module d3rlpy.dataset*), 94
create_impl() (*d3rlpy.base.LearnableBase method*), 28
create_impl() (*d3rlpy.ope.DiscreteFQE method*), 181
create_impl() (*d3rlpy.ope.FQE method*), 172
create_infinite_replay_buffer() (*in module d3rlpy.dataset*), 93
create_with_action() (*d3rlpy.models.DefaultEncoderFactory method*), 153
create_with_action() (*d3rlpy.models.DenseEncoderFactory method*), 160
create_with_action() (*d3rlpy.models.PixelEncoderFactory method*), 155
create_with_action() (*d3rlpy.models.VectorEncoderFactory method*), 158

CRR (*class in d3rlpy.algos*), 57
CRRConfig (*class in d3rlpy.algos*), 55

D

d3rlpy
    module, 27
d3rlpy.algos
    module, 27, 197
d3rlpy.dataset
    module, 84
d3rlpy.datasets
    module, 104
d3rlpy.logging
    module, 188
d3rlpy.metrics
    module, 162
d3rlpy.models
    module, 77, 141, 150
d3rlpy.ope
    module, 169
d3rlpy.preprocessing
    module, 107
dampening (*d3rlpy.models.SGDFactory attribute*), 145
DDPG (*class in d3rlpy.algos*), 43
DDPGConfig (*class in d3rlpy.algos*), 42
DecisionTransformer (*class in d3rlpy.algos*), 76
DecisionTransformerConfig (*class in d3rlpy.algos*), 75
DefaultEncoderFactory (*class in d3rlpy.models*), 152
DenseEncoderFactory (*class in d3rlpy.models*), 160
deserialize() (*d3rlpy.models.AdamFactory class method*), 146
deserialize() (*d3rlpy.models.DefaultEncoderFactory class method*), 153
deserialize() (*d3rlpy.models.DenseEncoderFactory class method*), 161
deserialize() (*d3rlpy.models.IQNQFunctionFactory class method*), 83
deserialize() (*d3rlpy.models.MeanQFunctionFactory class method*), 78
deserialize() (*d3rlpy.models.OptimizerFactory class method*), 142
deserialize() (*d3rlpy.models.PixelEncoderFactory class method*), 155
deserialize() (*d3rlpy.models.QRQFunctionFactory class method*), 80
deserialize() (*d3rlpy.models.RMSpropFactory class method*), 148
deserialize() (*d3rlpy.models.SGDFactory class method*), 144
deserialize() (*d3rlpy.models.VectorEncoderFactory class method*), 158
deserialize() (*d3rlpy.preprocessing.ClipRewardScaler class method*), 129
deserialize() (*d3rlpy.preprocessing.ConstantShiftRewardScaler class method*), 138
deserialize() (*d3rlpy.preprocessing.MinMaxActionScaler class method*), 119
deserialize() (*d3rlpy.preprocessing.MinMaxObservationScaler class method*), 112
deserialize() (*d3rlpy.preprocessing.MinMaxRewardScaler class method*), 123
deserialize() (*d3rlpy.preprocessing.MultiplyRewardScaler class method*), 132
deserialize() (*d3rlpy.preprocessing.PixelObservationScaler class method*), 109
deserialize() (*d3rlpy.preprocessing.ReturnBasedRewardScaler class method*), 135
deserialize() (*d3rlpy.preprocessing.StandardObservationScaler class method*), 115
deserialize() (*d3rlpy.preprocessing.StandardRewardScaler class method*), 126
deserialize_from_dict() (*d3rlpy.models.AdamFactory class method*), 146
deserialize_from_dict() (*d3rlpy.models.DefaultEncoderFactory class method*), 153
deserialize_from_dict() (*d3rlpy.models.DenseEncoderFactory class method*), 161
deserialize_from_dict() (*d3rlpy.models.IQNQFunctionFactory class method*), 83
deserialize_from_dict() (*d3rlpy.models.MeanQFunctionFactory class method*), 78
deserialize_from_dict() (*d3rlpy.models.OptimizerFactory class method*), 142
deserialize_from_dict() (*d3rlpy.models.PixelEncoderFactory class method*), 156
deserialize_from_dict() (*d3rlpy.models.QRQFunctionFactory class method*), 80
deserialize_from_dict() (*d3rlpy.models.RMSpropFactory class method*), 148
deserialize_from_dict() (*d3rlpy.models.SGDFactory class method*), 144
deserialize_from_dict() (*d3rlpy.models.VectorEncoderFactory class method*), 158
deserialize_from_dict() (*d3rlpy.preprocessing.ClipRewardScaler class method*), 129

deserialize_from_dict() (*d3rlpy.preprocessing.ConstantShiftRewardScaler class method*), 138
deserialize_from_dict() (*d3rlpy.preprocessing.MinMaxActionScaler class method*), 119
deserialize_from_dict() (*d3rlpy.preprocessing.MinMaxObservationScaler class method*), 112
deserialize_from_dict() (*d3rlpy.preprocessing.MinMaxRewardScaler class method*), 123
deserialize_from_dict() (*d3rlpy.preprocessing.MultiplyRewardScaler class method*), 132
deserialize_from_dict() (*d3rlpy.preprocessing.PixelObservationScaler class method*), 109
deserialize_from_dict() (*d3rlpy.preprocessing.ReturnBasedRewardScaler class method*), 135
deserialize_from_dict() (*d3rlpy.preprocessing.StandardObservationScaler class method*), 115
deserialize_from_dict() (*d3rlpy.preprocessing.StandardRewardScaler class method*), 126
deserialize_from_file() (*d3rlpy.models.AdamFactory class method*), 146
deserialize_from_file() (*d3rlpy.models.DefaultEncoderFactory class method*), 153
deserialize_from_file() (*d3rlpy.models.DenseEncoderFactory class method*), 161
deserialize_from_file() (*d3rlpy.models.IQNQFunctionFactory class method*), 83
deserialize_from_file() (*d3rlpy.models.MeanQFunctionFactory class method*), 78
deserialize_from_file() (*d3rlpy.models.OptimizerFactory class method*), 142
deserialize_from_file() (*d3rlpy.models.PixelEncoderFactory class method*), 156
deserialize_from_file() (*d3rlpy.models.QRQFunctionFactory class method*), 80
deserialize_from_file() (*d3rlpy.models.RMSpropFactory class method*), 149
deserialize_from_file() (*d3rlpy.models.SGDFactory class method*), 144
deserialize_from_file() (*d3rlpy.models.VectorEncoderFactory class method*), 158
deserialize_from_file() (*d3rlpy.preprocessing.ClipRewardScaler class method*), 129
deserialize_from_file() (*d3rlpy.preprocessing.ConstantShiftRewardScaler class method*), 139
deserialize_from_file() (*d3rlpy.preprocessing.MinMaxActionScaler class method*), 119
deserialize_from_file() (*d3rlpy.preprocessing.MinMaxObservationScaler class method*), 112
deserialize_from_file() (*d3rlpy.preprocessing.MinMaxRewardScaler class method*), 123
deserialize_from_file() (*d3rlpy.preprocessing.MultiplyRewardScaler class method*), 132
deserialize_from_file() (*d3rlpy.preprocessing.PixelObservationScaler class method*), 109
deserialize_from_file() (*d3rlpy.preprocessing.ReturnBasedRewardScaler class method*), 136
deserialize_from_file() (*d3rlpy.preprocessing.StandardObservationScaler class method*), 115
deserialize_from_file() (*d3rlpy.preprocessing.StandardRewardScaler class method*), 126
DiscountedSumOfAdvantageEvaluator (*class in d3rlpy.metrics*), 164
DiscreteActionMatchEvaluator (*class in d3rlpy.metrics*), 167
DiscreteBC (*class in d3rlpy.algos*), 38
DiscreteBCConfig (*class in d3rlpy.algos*), 37
DiscreteBCQ (*class in d3rlpy.algos*), 52
DiscreteBCQConfig (*class in d3rlpy.algos*), 51
DiscreteCQL (*class in d3rlpy.algos*), 60
DiscreteCQLConfig (*class in d3rlpy.algos*), 59
DiscreteFQE (*class in d3rlpy.ope*), 179
DiscreteRandomPolicy (*class in d3rlpy.algos*), 72
DiscreteRandomPolicyConfig (*class in d3rlpy.algos*), 71
DiscreteSAC (*class in d3rlpy.algos*), 48
DiscreteSACConfig (*class in d3rlpy.algos*), 47
DoubleDQN (*class in d3rlpy.algos*), 41
DoubleDQNConfig (*class in d3rlpy.algos*), 40

DQN (*class in d3rlpy.algos*), 40
DQNConfig (*class in d3rlpy.algos*), 39
dropout_rate (*d3rlpy.models.DefaultEncoderFactory attribute*), 155
dropout_rate (*d3rlpy.models.DenseEncoderFactory attribute*), 162
dropout_rate (*d3rlpy.models.PixelEncoderFactory attribute*), 157
dropout_rate (*d3rlpy.models.VectorEncoderFactory attribute*), 160
dump() (*d3rlpy.dataset.MDPDataset method*), 86
dump() (*d3rlpy.dataset.ReplayBuffer method*), 91

E

embed_size (*d3rlpy.models.IQNQFunctionFactory attribute*), 84
EnvironmentEvaluator (*class in d3rlpy.metrics*), 167
episodes (*d3rlpy.dataset.BufferProtocol attribute*), 95
episodes (*d3rlpy.dataset.FIFOBuffer attribute*), 97
episodes (*d3rlpy.dataset.InfiniteBuffer attribute*), 96
episodes (*d3rlpy.dataset.MDPDataset attribute*), 88
episodes (*d3rlpy.dataset.ReplayBuffer attribute*), 93
eps (*d3rlpy.models.AdamFactory attribute*), 148
eps (*d3rlpy.models.RMSpropFactory attribute*), 150
eps (*d3rlpy.preprocessing.StandardObservationScaler attribute*), 118
eps (*d3rlpy.preprocessing.StandardRewardScaler attribute*), 129
exclude_last_activation (*d3rlpy.models.PixelEncoderFactory attribute*), 157
exclude_last_activation (*d3rlpy.models.VectorEncoderFactory attribute*), 160

F

feature_size (*d3rlpy.models.PixelEncoderFactory attribute*), 157
FIFOBuffer (*class in d3rlpy.dataset*), 96
FileAdapter (*class in d3rlpy.logging*), 190
FileAdapterFactory (*class in d3rlpy.logging*), 196
fit() (*d3rlpy.algos.QLearningAlgoBase method*), 32
fit() (*d3rlpy.algos.TransformerAlgoBase method*), 74
fit() (*d3rlpy.ope.DiscreteFQE method*), 181
fit() (*d3rlpy.ope.FQE method*), 172
fit_online() (*d3rlpy.algos.QLearningAlgoBase method*), 32
fit_online() (*d3rlpy.ope.DiscreteFQE method*), 182
fit_online() (*d3rlpy.ope.FQE method*), 173
fit_with_env() (*d3rlpy.preprocessing.ClipRewardScaler method*), 129
fit_with_env() (*d3rlpy.preprocessing.ConstantShiftRewardScaler method*), 139
fit_with_env() (*d3rlpy.preprocessing.MinMaxActionScaler method*), 119
fit_with_env() (*d3rlpy.preprocessing.MinMaxObservationScaler method*), 112
fit_with_env() (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 123
fit_with_env() (*d3rlpy.preprocessing.MultiplyRewardScaler method*), 132
fit_with_env() (*d3rlpy.preprocessing.PixelObservationScaler method*), 109
fit_with_env() (*d3rlpy.preprocessing.ReturnBasedRewardScaler method*), 136
fit_with_env() (*d3rlpy.preprocessing.StandardObservationScaler method*), 115
fit_with_env() (*d3rlpy.preprocessing.StandardRewardScaler method*), 126
fit_with_trajectory_slicer() (*d3rlpy.preprocessing.ClipRewardScaler method*), 130
fit_with_trajectory_slicer() (*d3rlpy.preprocessing.ConstantShiftRewardScaler method*), 139
fit_with_trajectory_slicer() (*d3rlpy.preprocessing.MinMaxActionScaler method*), 119
fit_with_trajectory_slicer() (*d3rlpy.preprocessing.MinMaxObservationScaler method*), 112
fit_with_trajectory_slicer() (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 123
fit_with_trajectory_slicer() (*d3rlpy.preprocessing.MultiplyRewardScaler method*), 132
fit_with_trajectory_slicer() (*d3rlpy.preprocessing.PixelObservationScaler method*), 109
fit_with_trajectory_slicer() (*d3rlpy.preprocessing.ReturnBasedRewardScaler method*), 136
fit_with_trajectory_slicer() (*d3rlpy.preprocessing.StandardObservationScaler method*), 115
fit_with_trajectory_slicer() (*d3rlpy.preprocessing.StandardRewardScaler method*), 126
fit_with_transition_picker() (*d3rlpy.preprocessing.ClipRewardScaler method*), 130
fit_with_transition_picker() (*d3rlpy.preprocessing.ConstantShiftRewardScaler method*), 139
fit_with_transition_picker() (*d3rlpy.preprocessing.MinMaxActionScaler*

method), 120
fit_with_transition_picker() (*d3rlpy.preprocessing.MinMaxObservationScaler method*), 112
fit_with_transition_picker() (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 123
fit_with_transition_picker() (*d3rlpy.preprocessing.MultiplyRewardScaler method*), 133
fit_with_transition_picker() (*d3rlpy.preprocessing.PixelObservationScaler method*), 109
fit_with_transition_picker() (*d3rlpy.preprocessing.ReturnBasedRewardScaler method*), 136
fit_with_transition_picker() (*d3rlpy.preprocessing.StandardObservationScaler method*), 116
fit_with_transition_picker() (*d3rlpy.preprocessing.StandardRewardScaler method*), 127
fitter() (*d3rlpy.algos.QLearningAlgoBase method*), 33
fitter() (*d3rlpy.ope.DiscreteFQE method*), 183
fitter() (*d3rlpy.ope.FQE method*), 174
FQE (*class in d3rlpy.ope*), 170
FrameStackTransitionPicker (*class in d3rlpy.dataset*), 98
from_dict() (*d3rlpy.models.AdamFactory class method*), 146
from_dict() (*d3rlpy.models.DefaultEncoderFactory class method*), 153
from_dict() (*d3rlpy.models.DenseEncoderFactory class method*), 161
from_dict() (*d3rlpy.models.IQNQFunctionFactory class method*), 83
from_dict() (*d3rlpy.models.MeanQFunctionFactory class method*), 78
from_dict() (*d3rlpy.models.OptimizerFactory class method*), 142
from_dict() (*d3rlpy.models.PixelEncoderFactory class method*), 156
from_dict() (*d3rlpy.models.QRQFunctionFactory class method*), 81
from_dict() (*d3rlpy.models.RMSpropFactory class method*), 149
from_dict() (*d3rlpy.models.SGDFactory class method*), 144
from_dict() (*d3rlpy.models.VectorEncoderFactory class method*), 158
from_dict() (*d3rlpy.preprocessing.ClipRewardScaler class method*), 130
from_dict() (*d3rlpy.preprocessing.ConstantShiftRewardScaler class method*), 139
from_dict() (*d3rlpy.preprocessing.MinMaxActionScaler class method*), 120
from_dict() (*d3rlpy.preprocessing.MinMaxObservationScaler class method*), 113
from_dict() (*d3rlpy.preprocessing.MinMaxRewardScaler class method*), 124
from_dict() (*d3rlpy.preprocessing.MultiplyRewardScaler class method*), 133
from_dict() (*d3rlpy.preprocessing.PixelObservationScaler class method*), 109
from_dict() (*d3rlpy.preprocessing.ReturnBasedRewardScaler class method*), 136
from_dict() (*d3rlpy.preprocessing.StandardObservationScaler class method*), 116
from_dict() (*d3rlpy.preprocessing.StandardRewardScaler class method*), 127
from_episode_generator() (*d3rlpy.dataset.MDPDataset class method*), 86
from_episode_generator() (*d3rlpy.dataset.ReplayBuffer class method*), 91
from_json() (*d3rlpy.base.LearnableBase class method*), 28
from_json() (*d3rlpy.models.AdamFactory class method*), 146
from_json() (*d3rlpy.models.DefaultEncoderFactory class method*), 153
from_json() (*d3rlpy.models.DenseEncoderFactory class method*), 161
from_json() (*d3rlpy.models.IQNQFunctionFactory class method*), 83
from_json() (*d3rlpy.models.MeanQFunctionFactory class method*), 78
from_json() (*d3rlpy.models.OptimizerFactory class method*), 142
from_json() (*d3rlpy.models.PixelEncoderFactory class method*), 156
from_json() (*d3rlpy.models.QRQFunctionFactory class method*), 81
from_json() (*d3rlpy.models.RMSpropFactory class method*), 149
from_json() (*d3rlpy.models.SGDFactory class method*), 144
from_json() (*d3rlpy.models.VectorEncoderFactory class method*), 158
from_json() (*d3rlpy.ope.DiscreteFQE class method*), 184
from_json() (*d3rlpy.ope.FQE class method*), 174
from_json() (*d3rlpy.preprocessing.ClipRewardScaler class method*), 130
from_json() (*d3rlpy.preprocessing.ConstantShiftRewardScaler class method*), 139
from_json() (*d3rlpy.preprocessing.MinMaxActionScaler class method*), 120

from_json() (*d3rlpy.preprocessing.MinMaxObservationScaler class method*), 113
from_json() (*d3rlpy.preprocessing.MinMaxRewardScaler class method*), 124
from_json() (*d3rlpy.preprocessing.MultiplyRewardScaler class method*), 133
from_json() (*d3rlpy.preprocessing.PixelObservationScaler class method*), 109
from_json() (*d3rlpy.preprocessing.ReturnBasedRewardScaler class method*), 136
from_json() (*d3rlpy.preprocessing.StandardObservationScaler class method*), 116
from_json() (*d3rlpy.preprocessing.StandardRewardScaler class method*), 127

G

gamma (*d3rlpy.base.LearnableBase property*), 28
gamma (*d3rlpy.ope.DiscreteFQE attribute*), 187
gamma (*d3rlpy.ope.FQE attribute*), 178
get_action_type() (*d3rlpy.algos.AWAC method*), 62
get_action_type() (*d3rlpy.algos.BC method*), 37
get_action_type() (*d3rlpy.algos.BCQ method*), 51
get_action_type() (*d3rlpy.algos.BEAR method*), 55
get_action_type() (*d3rlpy.algos.CQL method*), 59
get_action_type() (*d3rlpy.algos.CRR method*), 57
get_action_type() (*d3rlpy.algos.DDPG method*), 43
get_action_type() (*d3rlpy.algos.DecisionTransformer method*), 76
get_action_type() (*d3rlpy.algos.DiscreteBC method*), 38
get_action_type() (*d3rlpy.algos.DiscreteBCQ method*), 52
get_action_type() (*d3rlpy.algos.DiscreteCQL method*), 60
get_action_type() (*d3rlpy.algos.DiscreteRandomPolicy method*), 72
get_action_type() (*d3rlpy.algos.DiscreteSAC method*), 48
get_action_type() (*d3rlpy.algos.DQN method*), 40
get_action_type() (*d3rlpy.algos.IQL method*), 69
get_action_type() (*d3rlpy.algos.NFQ method*), 39
get_action_type() (*d3rlpy.algos.PLAS method*), 64
get_action_type() (*d3rlpy.algos.RandomPolicy method*), 70
get_action_type() (*d3rlpy.algos.SAC method*), 47
get_action_type() (*d3rlpy.algos.TD3 method*), 45
get_action_type() (*d3rlpy.algos.TD3PlusBC method*), 67
get_action_type() (*d3rlpy.base.LearnableBase method*), 28
get_action_type() (*d3rlpy.ope.DiscreteFQE method*), 184
get_action_type() (*d3rlpy.ope.FQE method*), 175
get_atari() (*in module d3rlpy.datasets*), 105
get_atari_transitions() (*in module d3rlpy.datasets*), 105
get_cartpole() (*in module d3rlpy.datasets*), 104
get_d4rl() (*in module d3rlpy.datasets*), 106
get_dataset() (*in module d3rlpy.datasets*), 107
get_pendulum() (*in module d3rlpy.datasets*), 105
get_type() (*d3rlpy.models.AdamFactory static method*), 147
get_type() (*d3rlpy.models.DefaultEncoderFactory static method*), 153
get_type() (*d3rlpy.models.DenseEncoderFactory static method*), 161
get_type() (*d3rlpy.models.IQNQFunctionFactory static method*), 83
get_type() (*d3rlpy.models.MeanQFunctionFactory static method*), 79
get_type() (*d3rlpy.models.OptimizerFactory static method*), 142
get_type() (*d3rlpy.models.PixelEncoderFactory static method*), 156
get_type() (*d3rlpy.models.QRQFunctionFactory static method*), 81
get_type() (*d3rlpy.models.RMSpropFactory static method*), 149
get_type() (*d3rlpy.models.SGDFactory static method*), 144
get_type() (*d3rlpy.models.VectorEncoderFactory static method*), 158
get_type() (*d3rlpy.preprocessing.ClipRewardScaler static method*), 130
get_type() (*d3rlpy.preprocessing.ConstantShiftRewardScaler static method*), 139
get_type() (*d3rlpy.preprocessing.MinMaxActionScaler static method*), 120
get_type() (*d3rlpy.preprocessing.MinMaxObservationScaler static method*), 113
get_type() (*d3rlpy.preprocessing.MinMaxRewardScaler static method*), 124
get_type() (*d3rlpy.preprocessing.MultiplyRewardScaler static method*), 133
get_type() (*d3rlpy.preprocessing.PixelObservationScaler static method*), 110
get_type() (*d3rlpy.preprocessing.ReturnBasedRewardScaler static method*), 136
get_type() (*d3rlpy.preprocessing.StandardObservationScaler static method*), 116
get_type() (*d3rlpy.preprocessing.StandardRewardScaler static method*), 127
grad_step (*d3rlpy.base.LearnableBase property*), 29
grad_step (*d3rlpy.ope.DiscreteFQE attribute*), 187
grad_step (*d3rlpy.ope.FQE attribute*), 178

H

high (*d3rlpy.preprocessing.ClipRewardScaler attribute*),

132

I

impl (*d3rlpy.base.LearnableBase property*), 29
impl (*d3rlpy.ope.DiscreteFQE attribute*), 187
impl (*d3rlpy.ope.FQE attribute*), 178
InfiniteBuffer (*class in d3rlpy.dataset*), 95
InitialStateValueEstimationEvaluator (*class in d3rlpy.metrics*), 165
inner_create_impl() (*d3rlpy.ope.DiscreteFQE method*), 184
inner_create_impl() (*d3rlpy.ope.FQE method*), 175
inner_update() (*d3rlpy.algos.AWAC method*), 62
inner_update() (*d3rlpy.algos.BCQ method*), 51
inner_update() (*d3rlpy.algos.BEAR method*), 55
inner_update() (*d3rlpy.algos.CQL method*), 59
inner_update() (*d3rlpy.algos.CRR method*), 57
inner_update() (*d3rlpy.algos.DDPG method*), 43
inner_update() (*d3rlpy.algos.DecisionTransformer method*), 77
inner_update() (*d3rlpy.algos.DiscreteBCQ method*), 52
inner_update() (*d3rlpy.algos.DiscreteCQL method*), 60
inner_update() (*d3rlpy.algos.DiscreteRandomPolicy method*), 72
inner_update() (*d3rlpy.algos.DiscreteSAC method*), 48
inner_update() (*d3rlpy.algos.DQN method*), 40
inner_update() (*d3rlpy.algos.IQL method*), 69
inner_update() (*d3rlpy.algos.NFQ method*), 39
inner_update() (*d3rlpy.algos.PLAS method*), 64
inner_update() (*d3rlpy.algos.QLearningAlgoBase method*), 34
inner_update() (*d3rlpy.algos.RandomPolicy method*), 70
inner_update() (*d3rlpy.algos.SAC method*), 47
inner_update() (*d3rlpy.algos.TD3 method*), 45
inner_update() (*d3rlpy.algos.TD3PlusBC method*), 67
inner_update() (*d3rlpy.algos.TransformerAlgoBase method*), 75
inner_update() (*d3rlpy.ope.DiscreteFQE method*), 184
inner_update() (*d3rlpy.ope.FQE method*), 175
IQL (*class in d3rlpy.algos*), 69
IQLConfig (*class in d3rlpy.algos*), 68
IQNQFunctionFactory (*class in d3rlpy.models*), 82

L

LastFrameWriterPreprocess (*class in d3rlpy.dataset*), 103
LearnableBase (*class in d3rlpy.base*), 27
LinearDecayEpsilonGreedy (*class in d3rlpy.algos*), 199
load() (*d3rlpy.dataset.MDPDataset class method*), 87
load() (*d3rlpy.dataset.ReplayBuffer class method*), 92
load_model() (*d3rlpy.base.LearnableBase method*), 29
load_model() (*d3rlpy.ope.DiscreteFQE method*), 184
load_model() (*d3rlpy.ope.FQE method*), 175
logdir (*d3rlpy.logging.FileAdapter attribute*), 191
LoggerAdapter (*class in d3rlpy.logging*), 189
LoggerAdapterFactory (*class in d3rlpy.logging*), 195
low (*d3rlpy.preprocessing.ClipRewardScaler attribute*), 132

M

maximum (*d3rlpy.preprocessing.MinMaxActionScaler attribute*), 121
maximum (*d3rlpy.preprocessing.MinMaxObservationScaler attribute*), 114
maximum (*d3rlpy.preprocessing.MinMaxRewardScaler attribute*), 125
MDPDataset (*class in d3rlpy.dataset*), 85
mean (*d3rlpy.preprocessing.StandardObservationScaler attribute*), 118
mean (*d3rlpy.preprocessing.StandardRewardScaler attribute*), 129
MeanQFunctionFactory (*class in d3rlpy.models*), 77
minimum (*d3rlpy.preprocessing.MinMaxActionScaler attribute*), 121
minimum (*d3rlpy.preprocessing.MinMaxObservationScaler attribute*), 114
minimum (*d3rlpy.preprocessing.MinMaxRewardScaler attribute*), 125
MinMaxActionScaler (*class in d3rlpy.preprocessing*), 118
MinMaxObservationScaler (*class in d3rlpy.preprocessing*), 111
MinMaxRewardScaler (*class in d3rlpy.preprocessing*), 122
module
    d3rlpy, 27
    d3rlpy.algos, 27, 197
    d3rlpy.dataset, 84
    d3rlpy.datasets, 104
    d3rlpy.logging, 188
    d3rlpy.metrics, 162
    d3rlpy.models, 77, 141, 150
    d3rlpy.ope, 169
    d3rlpy.preprocessing, 107
momentum (*d3rlpy.models.RMSpropFactory attribute*), 150
momentum (*d3rlpy.models.SGDFactory attribute*), 145
multiplier (*d3rlpy.preprocessing.ClipRewardScaler attribute*), 132
multiplier (*d3rlpy.preprocessing.MinMaxRewardScaler attribute*), 125

multiplier (*d3rlpy.preprocessing.MultiplyRewardScaler attribute*), 135
multiplier (*d3rlpy.preprocessing.ReturnBasedRewardScaler attribute*), 138
multiplier (*d3rlpy.preprocessing.StandardRewardScaler attribute*), 129
MultiplyRewardScaler (*class in d3rlpy.preprocessing*), 132
MultiStepTransitionPicker (*class in d3rlpy.dataset*), 99

N

n_greedy_quantiles (*d3rlpy.models.IQNQFunctionFactory attribute*), 84
n_quantiles (*d3rlpy.models.IQNQFunctionFactory attribute*), 84
n_quantiles (*d3rlpy.models.QRQFunctionFactory attribute*), 82
nesterov (*d3rlpy.models.SGDFactory attribute*), 145
NFQ (*class in d3rlpy.algos*), 39
NFQConfig (*class in d3rlpy.algos*), 38
NoopAdapter (*class in d3rlpy.logging*), 193
NoopAdapterFactory (*class in d3rlpy.logging*), 196
NormalNoise (*class in d3rlpy.algos*), 199

O

observation_scaler (*d3rlpy.base.LearnableBase property*), 29
observation_scaler (*d3rlpy.ope.DiscreteFQE attribute*), 187
observation_scaler (*d3rlpy.ope.FQE attribute*), 178
observation_shape (*d3rlpy.base.LearnableBase property*), 29
observation_shape (*d3rlpy.ope.DiscreteFQE attribute*), 187
observation_shape (*d3rlpy.ope.FQE attribute*), 178
OptimizerFactory (*class in d3rlpy.models*), 142

P

PixelEncoderFactory (*class in d3rlpy.models*), 155
PixelObservationScaler (*class in d3rlpy.preprocessing*), 108
PLAS (*class in d3rlpy.algos*), 64
PLASConfig (*class in d3rlpy.algos*), 62
PLASWithPerturbation (*class in d3rlpy.algos*), 66
PLASWithPerturbationConfig (*class in d3rlpy.algos*), 64
predict() (*d3rlpy.algos.DiscreteRandomPolicy method*), 72
predict() (*d3rlpy.algos.QLearningAlgoBase method*), 34
predict() (*d3rlpy.algos.RandomPolicy method*), 70
predict() (*d3rlpy.algos.TransformerAlgoBase method*), 75
predict() (*d3rlpy.ope.DiscreteFQE method*), 184
predict() (*d3rlpy.ope.FQE method*), 175
predict_value() (*d3rlpy.algos.DiscreteRandomPolicy method*), 72
predict_value() (*d3rlpy.algos.QLearningAlgoBase method*), 34
predict_value() (*d3rlpy.algos.RandomPolicy method*), 70
predict_value() (*d3rlpy.ope.DiscreteFQE method*), 185
predict_value() (*d3rlpy.ope.FQE method*), 176
process_action() (*d3rlpy.dataset.BasicWriterPreprocess method*), 103
process_action() (*d3rlpy.dataset.LastFrameWriterPreprocess method*), 103
process_action() (*d3rlpy.dataset.WriterPreprocessProtocol method*), 102
process_observation() (*d3rlpy.dataset.BasicWriterPreprocess method*), 103
process_observation() (*d3rlpy.dataset.LastFrameWriterPreprocess method*), 103
process_observation() (*d3rlpy.dataset.WriterPreprocessProtocol method*), 102
process_reward() (*d3rlpy.dataset.BasicWriterPreprocess method*), 103
process_reward() (*d3rlpy.dataset.LastFrameWriterPreprocess method*), 104
process_reward() (*d3rlpy.dataset.WriterPreprocessProtocol method*), 102

Q

QLearningAlgoBase (*class in d3rlpy.algos*), 30
QRQFunctionFactory (*class in d3rlpy.models*), 80

R

RandomPolicy (*class in d3rlpy.algos*), 70
RandomPolicyConfig (*class in d3rlpy.algos*), 69
ReplayBuffer (*class in d3rlpy.dataset*), 90
reset_optimizer_states() (*d3rlpy.algos.QLearningAlgoBase method*), 35
reset_optimizer_states() (*d3rlpy.ope.DiscreteFQE method*), 185
reset_optimizer_states() (*d3rlpy.ope.FQE method*), 176
return_max (*d3rlpy.preprocessing.ReturnBasedRewardScaler attribute*), 138
return_min (*d3rlpy.preprocessing.ReturnBasedRewardScaler attribute*), 138
ReturnBasedRewardScaler (*class in d3rlpy.preprocessing*), 135

reverse_transform() (*d3rlpy.preprocessing.ClipRewardScaler method*), 130
reverse_transform() (*d3rlpy.preprocessing.ConstantShiftRewardScaler method*), 139
reverse_transform() (*d3rlpy.preprocessing.MinMaxActionScaler method*), 120
reverse_transform() (*d3rlpy.preprocessing.MinMaxObservationScaler method*), 113
reverse_transform() (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 124
reverse_transform() (*d3rlpy.preprocessing.MultiplyRewardScaler method*), 133
reverse_transform() (*d3rlpy.preprocessing.PixelObservationScaler method*), 110
reverse_transform() (*d3rlpy.preprocessing.ReturnBasedRewardScaler method*), 136
reverse_transform() (*d3rlpy.preprocessing.StandardObservationScaler method*), 116
reverse_transform() (*d3rlpy.preprocessing.StandardRewardScaler method*), 127
reverse_transform_numpy() (*d3rlpy.preprocessing.ClipRewardScaler method*), 130
reverse_transform_numpy() (*d3rlpy.preprocessing.ConstantShiftRewardScaler method*), 140
reverse_transform_numpy() (*d3rlpy.preprocessing.MinMaxActionScaler method*), 120
reverse_transform_numpy() (*d3rlpy.preprocessing.MinMaxObservationScaler method*), 113
reverse_transform_numpy() (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 124
reverse_transform_numpy() (*d3rlpy.preprocessing.MultiplyRewardScaler method*), 133
reverse_transform_numpy() (*d3rlpy.preprocessing.PixelObservationScaler method*), 110
reverse_transform_numpy() (*d3rlpy.preprocessing.ReturnBasedRewardScaler method*), 137
reverse_transform_numpy() (*d3rlpy.preprocessing.StandardObservationScaler method*), 116
reverse_transform_numpy() (*d3rlpy.preprocessing.StandardRewardScaler method*), 127
reward_scaler (*d3rlpy.base.LearnableBase property*), 29
reward_scaler (*d3rlpy.ope.DiscreteFQE attribute*), 188
reward_scaler (*d3rlpy.ope.FQE attribute*), 179
RMSpropFactory (*class in d3rlpy.models*), 148

S

SAC (*class in d3rlpy.algos*), 46
SACConfig (*class in d3rlpy.algos*), 45
sample() (*d3rlpy.algos.ConstantEpsilonGreedy method*), 198
sample() (*d3rlpy.algos.LinearDecayEpsilonGreedy method*), 199
sample() (*d3rlpy.algos.NormalNoise method*), 199
sample_action() (*d3rlpy.algos.DiscreteRandomPolicy method*), 73
sample_action() (*d3rlpy.algos.QLearningAlgoBase method*), 35
sample_action() (*d3rlpy.algos.RandomPolicy method*), 71
sample_action() (*d3rlpy.ope.DiscreteFQE method*), 185
sample_action() (*d3rlpy.ope.FQE method*), 176
sample_trajectory() (*d3rlpy.dataset.MDPDataset method*), 87
sample_trajectory() (*d3rlpy.dataset.ReplayBuffer method*), 92
sample_trajectory_batch() (*d3rlpy.dataset.MDPDataset method*), 87
sample_trajectory_batch() (*d3rlpy.dataset.ReplayBuffer method*), 92
sample_transition() (*d3rlpy.dataset.MDPDataset method*), 88
sample_transition() (*d3rlpy.dataset.ReplayBuffer method*), 93
sample_transition_batch() (*d3rlpy.dataset.MDPDataset method*), 88
sample_transition_batch() (*d3rlpy.dataset.ReplayBuffer method*), 93
save() (*d3rlpy.base.LearnableBase method*), 29
save() (*d3rlpy.ope.DiscreteFQE method*), 185
save() (*d3rlpy.ope.FQE method*), 176
save_model() (*d3rlpy.base.LearnableBase method*), 29
save_model() (*d3rlpy.logging.CombineAdapter method*), 194
save_model() (*d3rlpy.logging.FileAdapter method*), 191

save_model() (*d3rlpy.logging.LoggerAdapter method*), 190
save_model() (*d3rlpy.logging.NoopAdapter method*), 193
save_model() (*d3rlpy.logging.TensorboardAdapter method*), 192
save_model() (*d3rlpy.ope.DiscreteFQE method*), 186
save_model() (*d3rlpy.ope.FQE method*), 177
save_policy() (*d3rlpy.algos.QLearningAlgoBase method*), 35
save_policy() (*d3rlpy.ope.DiscreteFQE method*), 186
save_policy() (*d3rlpy.ope.FQE method*), 177
schema() (*d3rlpy.models.AdamFactory class method*), 147
schema() (*d3rlpy.models.DefaultEncoderFactory class method*), 154
schema() (*d3rlpy.models.DenseEncoderFactory class method*), 161
schema() (*d3rlpy.models.IQNQFunctionFactory class method*), 83
schema() (*d3rlpy.models.MeanQFunctionFactory class method*), 79
schema() (*d3rlpy.models.OptimizerFactory class method*), 142
schema() (*d3rlpy.models.PixelEncoderFactory class method*), 156
schema() (*d3rlpy.models.QRQFunctionFactory class method*), 81
schema() (*d3rlpy.models.RMSpropFactory class method*), 149
schema() (*d3rlpy.models.SGDFactory class method*), 144
schema() (*d3rlpy.models.VectorEncoderFactory class method*), 159
schema() (*d3rlpy.preprocessing.ClipRewardScaler class method*), 130
schema() (*d3rlpy.preprocessing.ConstantShiftRewardScaler class method*), 140
schema() (*d3rlpy.preprocessing.MinMaxActionScaler class method*), 120
schema() (*d3rlpy.preprocessing.MinMaxObservationScaler class method*), 113
schema() (*d3rlpy.preprocessing.MinMaxRewardScaler class method*), 124
schema() (*d3rlpy.preprocessing.MultiplyRewardScaler class method*), 133
schema() (*d3rlpy.preprocessing.PixelObservationScaler class method*), 110
schema() (*d3rlpy.preprocessing.ReturnBasedRewardScaler class method*), 137
schema() (*d3rlpy.preprocessing.StandardObservationScaler class method*), 116
schema() (*d3rlpy.preprocessing.StandardRewardScaler class method*), 127
serialize() (*d3rlpy.models.AdamFactory method*), 147
serialize() (*d3rlpy.models.DefaultEncoderFactory method*), 154
serialize() (*d3rlpy.models.DenseEncoderFactory method*), 161
serialize() (*d3rlpy.models.IQNQFunctionFactory method*), 83
serialize() (*d3rlpy.models.MeanQFunctionFactory method*), 79
serialize() (*d3rlpy.models.OptimizerFactory method*), 143
serialize() (*d3rlpy.models.PixelEncoderFactory method*), 156
serialize() (*d3rlpy.models.QRQFunctionFactory method*), 81
serialize() (*d3rlpy.models.RMSpropFactory method*), 149
serialize() (*d3rlpy.models.SGDFactory method*), 145
serialize() (*d3rlpy.models.VectorEncoderFactory method*), 159
serialize() (*d3rlpy.preprocessing.ClipRewardScaler method*), 131
serialize() (*d3rlpy.preprocessing.ConstantShiftRewardScaler method*), 140
serialize() (*d3rlpy.preprocessing.MinMaxActionScaler method*), 120
serialize() (*d3rlpy.preprocessing.MinMaxObservationScaler method*), 113
serialize() (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 124
serialize() (*d3rlpy.preprocessing.MultiplyRewardScaler method*), 134
serialize() (*d3rlpy.preprocessing.PixelObservationScaler method*), 110
serialize() (*d3rlpy.preprocessing.ReturnBasedRewardScaler method*), 137
serialize() (*d3rlpy.preprocessing.StandardObservationScaler method*), 117
serialize() (*d3rlpy.preprocessing.StandardRewardScaler method*), 128
serialize_to_dict() (*d3rlpy.models.AdamFactory method*), 147
serialize_to_dict() (*d3rlpy.models.DefaultEncoderFactory method*), 154
serialize_to_dict() (*d3rlpy.models.DenseEncoderFactory method*), 161
serialize_to_dict() (*d3rlpy.models.IQNQFunctionFactory method*), 84
serialize_to_dict() (*d3rlpy.models.MeanQFunctionFactory*

method), 79
serialize_to_dict() (*d3rlpy.models.OptimizerFactory method*), 143
serialize_to_dict() (*d3rlpy.models.PixelEncoderFactory method*), 156
serialize_to_dict() (*d3rlpy.models.QRQFunctionFactory method*), 81
serialize_to_dict() (*d3rlpy.models.RMSpropFactory method*), 149
serialize_to_dict() (*d3rlpy.models.SGDFactory method*), 145
serialize_to_dict() (*d3rlpy.models.VectorEncoderFactory method*), 159
serialize_to_dict() (*d3rlpy.preprocessing.ClipRewardScaler method*), 131
serialize_to_dict() (*d3rlpy.preprocessing.ConstantShiftRewardScaler method*), 140
serialize_to_dict() (*d3rlpy.preprocessing.MinMaxActionScaler method*), 121
serialize_to_dict() (*d3rlpy.preprocessing.MinMaxObservationScaler method*), 113
serialize_to_dict() (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 124
serialize_to_dict() (*d3rlpy.preprocessing.MultiplyRewardScaler method*), 134
serialize_to_dict() (*d3rlpy.preprocessing.PixelObservationScaler method*), 110
serialize_to_dict() (*d3rlpy.preprocessing.ReturnBasedRewardScaler method*), 137
serialize_to_dict() (*d3rlpy.preprocessing.StandardObservationScaler method*), 117
serialize_to_dict() (*d3rlpy.preprocessing.StandardRewardScaler method*), 128
set_grad_step() (*d3rlpy.base.LearnableBase method*), 30
set_grad_step() (*d3rlpy.ope.DiscreteFQE method*), 186
set_grad_step() (*d3rlpy.ope.FQE method*), 177
SGDFactory (*class in d3rlpy.models*), 143
share_encoder (*d3rlpy.models.IQNQFunctionFactory attribute*), 84
share_encoder (*d3rlpy.models.MeanQFunctionFactory attribute*), 80
share_encoder (*d3rlpy.models.QRQFunctionFactory attribute*), 82
size() (*d3rlpy.dataset.MDPDataset method*), 88
size() (*d3rlpy.dataset.ReplayBuffer method*), 93
SoftOPCEvaluator (*class in d3rlpy.metrics*), 166
StandardObservationScaler (*class in d3rlpy.preprocessing*), 115
StandardRewardScaler (*class in d3rlpy.preprocessing*), 126
std (*d3rlpy.preprocessing.StandardObservationScaler attribute*), 118
std (*d3rlpy.preprocessing.StandardRewardScaler attribute*), 129

T

TD3 (*class in d3rlpy.algos*), 44
TD3Config (*class in d3rlpy.algos*), 43
TD3PlusBC (*class in d3rlpy.algos*), 67
TD3PlusBCConfig (*class in d3rlpy.algos*), 66
TDErrorEvaluator (*class in d3rlpy.metrics*), 163
TensorboardAdapter (*class in d3rlpy.logging*), 191
TensorboardAdapterFactory (*class in d3rlpy.logging*), 196
to_dict() (*d3rlpy.models.AdamFactory method*), 147
to_dict() (*d3rlpy.models.DefaultEncoderFactory method*), 154
to_dict() (*d3rlpy.models.DenseEncoderFactory method*), 162
to_dict() (*d3rlpy.models.IQNQFunctionFactory method*), 84
to_dict() (*d3rlpy.models.MeanQFunctionFactory method*), 79
to_dict() (*d3rlpy.models.OptimizerFactory method*), 143
to_dict() (*d3rlpy.models.PixelEncoderFactory method*), 156
to_dict() (*d3rlpy.models.QRQFunctionFactory method*), 81
to_dict() (*d3rlpy.models.RMSpropFactory method*), 149
to_dict() (*d3rlpy.models.SGDFactory method*), 145
to_dict() (*d3rlpy.models.VectorEncoderFactory method*), 159
to_dict() (*d3rlpy.preprocessing.ClipRewardScaler method*), 131
to_dict() (*d3rlpy.preprocessing.ConstantShiftRewardScaler method*), 140
to_dict() (*d3rlpy.preprocessing.MinMaxActionScaler method*), 121

to_dict() (*d3rlpy.preprocessing.MinMaxObservationScaler method*), 113
to_dict() (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 124
to_dict() (*d3rlpy.preprocessing.MultiplyRewardScaler method*), 134
to_dict() (*d3rlpy.preprocessing.PixelObservationScaler method*), 110
to_dict() (*d3rlpy.preprocessing.ReturnBasedRewardScaler method*), 137
to_dict() (*d3rlpy.preprocessing.StandardObservationScaler method*), 117
to_dict() (*d3rlpy.preprocessing.StandardRewardScaler method*), 128
to_json() (*d3rlpy.models.AdamFactory method*), 147
to_json() (*d3rlpy.models.DefaultEncoderFactory method*), 154
to_json() (*d3rlpy.models.DenseEncoderFactory method*), 162
to_json() (*d3rlpy.models.IQNQFunctionFactory method*), 84
to_json() (*d3rlpy.models.MeanQFunctionFactory method*), 79
to_json() (*d3rlpy.models.OptimizerFactory method*), 143
to_json() (*d3rlpy.models.PixelEncoderFactory method*), 157
to_json() (*d3rlpy.models.QRQFunctionFactory method*), 81
to_json() (*d3rlpy.models.RMSpropFactory method*), 149
to_json() (*d3rlpy.models.SGDFactory method*), 145
to_json() (*d3rlpy.models.VectorEncoderFactory method*), 159
to_json() (*d3rlpy.preprocessing.ClipRewardScaler method*), 131
to_json() (*d3rlpy.preprocessing.ConstantShiftRewardScaler method*), 140
to_json() (*d3rlpy.preprocessing.MinMaxActionScaler method*), 121
to_json() (*d3rlpy.preprocessing.MinMaxObservationScaler method*), 114
to_json() (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 125
to_json() (*d3rlpy.preprocessing.MultiplyRewardScaler method*), 134
to_json() (*d3rlpy.preprocessing.PixelObservationScaler method*), 110
to_json() (*d3rlpy.preprocessing.ReturnBasedRewardScaler method*), 137
to_json() (*d3rlpy.preprocessing.StandardObservationScaler method*), 117
to_json() (*d3rlpy.preprocessing.StandardRewardScaler method*), 128
trajectory_slicer (*d3rlpy.dataset.MDPDataset attribute*), 88
trajectory_slicer (*d3rlpy.dataset.ReplayBuffer attribute*), 93
TrajectorySlicerProtocol (*class in d3rlpy.dataset*), 101
transform() (*d3rlpy.preprocessing.ClipRewardScaler method*), 131
transform() (*d3rlpy.preprocessing.ConstantShiftRewardScaler method*), 140
transform() (*d3rlpy.preprocessing.MinMaxActionScaler method*), 121
transform() (*d3rlpy.preprocessing.MinMaxObservationScaler method*), 114
transform() (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 125
transform() (*d3rlpy.preprocessing.MultiplyRewardScaler method*), 134
transform() (*d3rlpy.preprocessing.PixelObservationScaler method*), 111
transform() (*d3rlpy.preprocessing.ReturnBasedRewardScaler method*), 137
transform() (*d3rlpy.preprocessing.StandardObservationScaler method*), 117
transform() (*d3rlpy.preprocessing.StandardRewardScaler method*), 128
transform_numpy() (*d3rlpy.preprocessing.ClipRewardScaler method*), 131
transform_numpy() (*d3rlpy.preprocessing.ConstantShiftRewardScaler method*), 141
transform_numpy() (*d3rlpy.preprocessing.MinMaxActionScaler method*), 121
transform_numpy() (*d3rlpy.preprocessing.MinMaxObservationScaler method*), 114
transform_numpy() (*d3rlpy.preprocessing.MinMaxRewardScaler method*), 125
transform_numpy() (*d3rlpy.preprocessing.MultiplyRewardScaler method*), 134
transform_numpy() (*d3rlpy.preprocessing.PixelObservationScaler method*), 111
transform_numpy() (*d3rlpy.preprocessing.ReturnBasedRewardScaler method*), 138
transform_numpy() (*d3rlpy.preprocessing.StandardObservationScaler method*), 117
transform_numpy() (*d3rlpy.preprocessing.StandardRewardScaler method*), 128
TransformerAlgoBase (*class in d3rlpy.algos*), 74
transition_count (*d3rlpy.dataset.BufferProtocol attribute*), 95
transition_count (*d3rlpy.dataset.FIFOBuffer attribute*), 97
transition_count (*d3rlpy.dataset.InfiniteBuffer attribute*), 96
transition_count (*d3rlpy.dataset.MDPDataset*

attribute), 88
transition_count (*d3rlpy.dataset.ReplayBuffer attribute*), 93
transition_picker (*d3rlpy.dataset.MDPDataset attribute*), 88
transition_picker (*d3rlpy.dataset.ReplayBuffer attribute*), 93
TransitionPickerProtocol (*class in d3rlpy.dataset*), 97

U

update() (*d3rlpy.algos.QLearningAlgoBase method*), 36
update() (*d3rlpy.algos.TransformerAlgoBase method*), 75
update() (*d3rlpy.ope.DiscreteFQE method*), 186
update() (*d3rlpy.ope.FQE method*), 177
use_batch_norm (*d3rlpy.models.DefaultEncoderFactory attribute*), 155
use_batch_norm (*d3rlpy.models.DenseEncoderFactory attribute*), 162
use_batch_norm (*d3rlpy.models.PixelEncoderFactory attribute*), 157
use_batch_norm (*d3rlpy.models.VectorEncoderFactory attribute*), 160
use_dense (*d3rlpy.models.VectorEncoderFactory attribute*), 160

V

VectorEncoderFactory (*class in d3rlpy.models*), 157

W

weight_decay (*d3rlpy.models.AdamFactory attribute*), 148
weight_decay (*d3rlpy.models.RMSpropFactory attribute*), 150
weight_decay (*d3rlpy.models.SGDFactory attribute*), 145
write_metric() (*d3rlpy.logging.CombineAdapter method*), 194
write_metric() (*d3rlpy.logging.FileAdapter method*), 191
write_metric() (*d3rlpy.logging.LoggerAdapter method*), 190
write_metric() (*d3rlpy.logging.NoopAdapter method*), 193
write_metric() (*d3rlpy.logging.TensorboardAdapter method*), 192
write_params() (*d3rlpy.logging.CombineAdapter method*), 195
write_params() (*d3rlpy.logging.FileAdapter method*), 191
write_params() (*d3rlpy.logging.LoggerAdapter method*), 190
write_params() (*d3rlpy.logging.NoopAdapter method*), 193
write_params() (*d3rlpy.logging.TensorboardAdapter method*), 192
WriterPreprocessProtocol (*class in d3rlpy.dataset*), 102